SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

SORTING AND SEARCHING IN MULTISETS*

IAN MUNRO† AND PHILIP M. SPIRA‡

Abstract. In this paper the problem of sorting multisets is considered. An information theoretic lower bound on the number of three branch comparisons is obtained, and it is shown that this bound is asymptotically attainable. It is shown that the multiplicities of a set can only be obtained by comparisons if the total order is discovered in the process. A lower bound on finding the mode of a multiset as a function of the actual multiplicity is given, and it is demonstrated that the bound can be achieved to within a multiplicative constant. The determination of the intersection of two multisets is also discussed, and partial results, including a generalization of Reingold's result for determining whether or not two sets have a nonempty intersection, are obtained.

Key words. sorting, multiset, lower bounds, mode, multiplicity set

1. Introduction. For many years sorting has been extensively studied by numerous investigators. This work is well-documented by Knuth [4]. In this paper we consider the problem of sorting a multiset. In this case, the familiar $n \log n$[1] information theory lower bound no longer is valid, and is replaced by a bound which depends upon the multiplicities of the numbers in the set being sorted. We show that this bound is attainable to within at most $O(n \log \log n)$, where the set has cardinality n. We also study the problem of obtaining the multiplicities—or spectrum—of a set, and show that the set must effectively be sorted to do this. Related results concerned with obtaining the mode of a set are also given. We also discuss the problem of determining whether two sets have an element in common.

2. Definitions and preliminaries. Let $S = \{x_1, \cdots, x_n\}$ be a set of not necessarily distinct real numbers. Knuth [4] and others call such a set a multiset.

Assume the values occurring are $y_1 < y_2 \cdots < y_k$. Then we say that $M = \{m_1, \cdots, m_k\}$ is the multiplicity set of S where each y_i occurs m_i times, $1 \leqq i \leqq k$, and of course,

$$\sum_{i=1}^{k} m_i = n.$$

We shall be interested in sorting and in determining the multiplicity set of a multiset. We shall also study the complexity of finding the mode—or most frequent value—of such a set. A number of related questions will be addressed as well. We will make no attempt to perform stable sorts; that is, in the case that $x_i = x_j$ it shall be immaterial which occurs first in the sorted sequence. Our unit operation will be a three-branch comparison, i.e., two elements x and y will be compared yielding $x > y$, $x = y$, or $x < y$ as an answer.

---

* Received by the editors December 1, 1973, and in revised form November 15, 1974. This work was supported in part by the National Science Foundation (USA) under Grant GJ-35604 and the National Research Council (Canada) under Grant A-8237.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

‡ Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California, Berkeley, California 94720.

[1] Note: all logarithms are taken to base 2 unless otherwise stated.

3. The results. It is well known that, to lower order terms, a set $S = \{x_1, \cdots, x_n\}$ requires $n \log n$ comparisons to sort and this number is sufficient. We obtain different bounds for multisets which follow from the fact that in general there are no longer $n!$ possible orderings, but some smaller number dependent upon the m_i. Paterson [5] has observed that any sorting algorithm can be applied to a multiset to save comparisons.

THEOREM 3.1 [Paterson]. *Let a sorting algorithm be given which uses $F(n)$ comparisons on a set of size n. Then it can sort a multiset having cardinality n, in*

$$F(n) - \sum_{i=1}^{k} m_i \log m_i + O(n)$$

three-branch comparisons.

Proof. Consider a class of m_i equal elements. It is obvious that only $m_i - 1$ comparisons need be made between these elements in order to establish their equality. Indeed the use of any more such comparisons is necessarily redundant. Now consider the case in which all elements of such a class are adjusted slightly so as to be all distinct, yet still lying in the same positions relative to the rest of the list. Any comparisons made between elements in the class and those outside contribute nothing to our knowledge of the internal order of this class. Therefore $\log (m_i!)$ comparisons must be made between such elements. This implies a savings of $m_i \log m_i - O(m_i)$ comparisons may be made over the case in which all elements of the given class are distinct. Considering all such classes, we see the sort can be performed in

$$F(n) - \sum_{i=1}^{k} m_i \log m_i + O(n)$$

three-branch comparisons. Q.E.D.

COROLLARY 3.2. *A multiset $S = \{x_1, \cdots, x_n\}$ with multiplicities $M = \{m_1, \cdots, m_k\}$ can be sorted in*

$$n \log n - \sum m_i \log m_i + O(n)$$

three-branch comparisons.

It does not follow from this that any $n \log n$ sort algorithm can be adapted easily to sort a multiset in the above upper bound without incurring excessive "administrative costs" in "avoiding unnecessary comparisons". In the cases of treesort and mergesort, however, reasonable modifications can be made, as we now outline.

(An adaptation of mergesort). Suppose $S = \{x_1, \cdots, x_n\}$ has multiplicities $M = \{m_1, \cdots, m_k\}$. Then divide S into sets of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ (S_1 and S_2 of multiplicities $M_1 = \{m_{11}, \cdots, m_{k1}\}$ and $M_2 = \{m_{12}, \cdots, m_{k2}\}$ respectively, where $m_{j1} + m_{j2} = m_j$, $1 \leqq j \leqq k$, noting that some of the m_{ij} can be 0). We sort S_1 and S_2 recursively, and then merge the two sorted lists. Whenever two equal elements are found, one is thrown away, and a record of the number of occurrences of the item is kept with the remaining copy. Again, we observe that $n - k$ "collapsings", or "equal comparisons" will be encountered in sorting the entire list. We now determine an upper bound on the number of "not equal" comparisons.

Suppose S_1 is sortable in

$$\left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor - \sum_{i=1}^{k} m_{i1} \log m_{i1} - \left\lfloor \frac{n}{4} \right\rfloor$$

"not equal" comparisons and that S_2 is sortable in the corresponding time ("0 log 0" means 0). To merge the two sorted lists requires at most k comparisons. Indeed, if l is the number of items present in both lists, at most $k - l$ comparisons resulting in a not equal relation are used, giving a bound of

$$n \log n - n - \sum_{j=1}^{2} \sum_{i=1}^{k} m_{ij} \log m_{ij} - \left\lfloor \frac{n}{2} \right\rfloor + k - l$$

not equal comparisons. Assume, without loss of generality, that the l "shared" items are those whose multiplicities are given by $\{m_{ij} : i = 1, \cdots, l\}$. That is, either m_{i1} or m_{i2} is 0 for $i > l$, hence

$$\sum_{j=1}^{2} \sum_{i=l+1}^{k} m_{ij} \log m_{ij} = \sum_{i=l+1}^{k} m_i \log m_i.$$

If both m_{1i} and m_{2i} are greater than 0, $m_{1i} \log m_{1i} + m_{2i} \log m_{2i}$ is minimized (worst case) subject to $m_{1i} + m_{2i} = m_i$ when $m_{1i} = m_{2i} = m_i/2$. Hence

$$\sum_{j=1}^{2} \sum_{i=1}^{l} m_{ij} \log m_{ij} \geqq \sum_{i=1}^{l} m_i(\log m_i - 1).$$

This yields a bound on the number of not equal comparisons as

$$n \log n - n - \sum_{i=1}^{k} m_i \log m_i + \sum_{i=1}^{l} m_i - \left\lfloor \frac{n}{2} \right\rfloor + k - l$$

$$\leqq n \log n - \sum_{i=1}^{k} m_i \log m_i - \left\lfloor \frac{n}{2} \right\rfloor$$

since

$$n \geqq \sum_{i=1}^{l} m_i + k - l.$$

We now show that a treesort type algorithm also is easily adapted to multisets.

The terms "treesort" and "heapsort" (see Floyd [2] or [3] or Williams [9]) refer to sorting algorithms, which are of the following general form:

1. Arrange the elements of the list into a (presumably implicit) balanced binary tree with an ordering such that if x is the father of y, then $x \geqq y$ (i.e. form a heap). This can be done in $O(n)$ comparisons, and furthermore, the maximal element is at the root of the tree.

2. Repeat the following section until the heap is empty:

(i) Remove the element at the root, it is the "next largest" element. This leaves the root "empty".

(ii) While there is an "empty node" which is not a leaf promote to its position the larger of its sons.

We observe that each iteration through step 2 will use at most $\log n$ (the height of the heap) comparisons and in fact $n \log n + O(n)$ comparisons are used in the entire procedure. Treesort is usually phrased so that it can be performed in place. Since this causes an increase in the number of comparisons required we will confine our attention to the method outlined above which uses an "output buffer".

We adapt this algorithm to sorting multisets with three-way branching by saying that whenever equality is discovered, promote a copy of the duplicated element together with the number of times it is known to occur. That is, if x has been found i times and y, j times, on discovering $x = y$ we promote x (or y) together with its "cardinality", $i + j$. We then initiate § 2 of the algorithm which promotes elements up to the positions x and y had held. An inspection of this procedure and comparison with the usual (previously described) treesort indicates that exactly the same comparisons are performed between unequal pairs of elements in the two cases; however, no "redundant" comparisons are made between equal elements. It follows from the proof of Theorem 3.1 that only $n \log_2 n - \sum m_i \log_2 m_i + O(n)$ comparisons are used.

These algorithms have the interesting property of approaching a bound which is not known at the commencement of the computation. Furthermore, note that sorting a list of length n having k distinct classes takes longest when all multiplicities are approximately the same.

We next give a lower bound on the number of three-branch comparisons to sort a multiset. To lower order terms it is the same as the upper bound.

THEOREM 3.3. *Let S be a multiset whose multiplicities are $m_1, \cdots, m_k$. Then to perform a multisort on S using three way comparisons requires at least*

$$n \log n - \sum_{i=1}^{k} m_i \log m_i - (n - k) \log \log k - O(n)$$

comparisons on the average.

Proof. The sequence of answers can be viewed as a word over the ternary alphabet $\{>, =, <\}$ in which there are exactly $n - k$ occurrences of $=$ in the word (since any additional $=$'s are redundant). There are $\binom{n}{m_1 \cdots m_k}$ ways to place the elements of S into classes. Hence, letting $s = n - k$, we obtain

$$\binom{T}{s} 2^{T-s} \geqq \binom{n}{m_1 \cdots m_k},$$

where T is the minimum average time for the multisort over all sets with such multiplicities. Taking logarithms we see that

$$T \geqq n \log n - \sum_{i=1}^{k} m_i \log m_i - s \log \frac{T}{s} - O(n).$$

Using the fact that $T \leqq \log \binom{n}{m_1 \cdots m_k} + O(n)$, and that the multinomial coefficient is maximal when all the m_i are as nearly equal as possible we get

$$T \geqq n \log n - \sum_{i=1}^{k} m_i \log m_i - s \log \log k - s \log \frac{n}{s} - O(n).$$

and the result follows, since $s \log n/s \leqq n/2$. Q.E.D.

Next we consider a related problem. We are given a multiset S and wish to determine its set of multiplicities M. We know that if we are given M, it does not help much in sorting S; nevertheless we have the following theorem.

THEOREM 3.4. *To find the spectrum M of a multiset using ternary comparisons requires that the total order of S be known.*

Proof. To be able to say that $m_i \in M$ by ternary comparisons we must establish that some subset of S consists of equal elements and has cardinality m_i. Similarly in finding that two classes are distinct using comparisons, one will also find which one contains the larger elements. Thus a sort must be performed. Q.E.D.

Note that this result does not imply that some other method for finding the spectrum would necessarily entail sorting or that it would even find out which particular members of S were in which class.

We now turn our attention to computing the mode of a multiset. We show that on the average any algorithm using only comparisons to do this takes at least $n \log (n/m)$ comparisons to within lower order terms.

THEOREM 3.5. *Let S be a multiset with cardinality n containing k distinct elements and in which the mode occurs m times. Then any algorithm to find the mode of S using only ternary comparisons takes at least*

$$n \log (n/m) - O(n - k) \log \log k - O(n)$$

comparisons on the average.

Proof. The method of proof is to show that, given that the mode has been found, the set can be sorted from this point without too much additional work. Suppose that the mode of S has been found and occurs with multiplicity m. This means that given any $m + 1$ elements of S we already know of an x and y in this subset such that $x < y$. In particular, there are at most m items which have lost no comparisons in the finding of the mode. Place them at the leaves of a balanced binary tree of height $\lceil \log m \rceil$, and run a tournament to find the maximum. Then remove all the maxima from the tree replacing them by items which lost to only those maxima in the process of finding the mode. The tree now contains at most m items, and among them are all the second maxima. So extract them and replace them by the items not on the tree that only they or the maxima beat in finding the mode. Again there are at most m items on the tree among which are all the third maxima. Proceed in this way until the sort is complete. We see that

$$\sum_{i=1}^{k} m_i \log m_i - O(n)$$

("equal") comparisons are "saved" as in the proof of Theorem 3.1. The tree has height $\lceil \log m \rceil$ so an additional

$$n \log m - \sum_{i=1}^{k} m_i \log m_i + O(n)$$

comparisons suffice to sort S. Subtracting this from the lower bound to sort given by Theorem 3.3 we obtain the result. Q.E.D.

This theorem shows that in certain asymptotic situations the mode is essentially as difficult to find as sorting the set. For example, if $m = n^{1/p}$, then the

additional work to sort will be only $(n \log n)(1/p)$. If $m = r$ for some constant r, then the additional work to sort is linear in n, hence to tell if a set has an element of multiplicity r requires $n \log (n/r) - O(n)$ comparisons.

We now attempt to achieve this lower bound on finding the mode.

THEOREM 3.6. *The mode of a multiset S can be found in*

$$3n\lceil \log_2 n/m \rceil$$

three-branch comparisons where S has cardinality n and $m = \max \{m_i\}$.

Proof. The basic idea is to find the mode by finding the median of appropriate subsets of S. We shall say a subset of S is homogeneous if all its elements are known to be equal. Otherwise, it is called heterogeneous. The key step is to take a subset S' of S and determine its median. By doing so we partition S' into two heterogeneous sets, $S^>$ and $S^<$, and one homogeneous set, $S^=$. Let $\mathcal{S}$ be maintained as a set of disjoint heterogeneous subsets of S. Initially, let $\mathcal{S} = \{S\}$ and let m (initially 0) denote the cardinality of the largest "class" (homogeneous subset) found thus far. The algorithm is as follows:

While a member of $\mathcal{S}$ (call it S') is of cardinality $> m$ *do*
begin
Find the median, med, of S' and so partition S' into $S^<$ (those elements less than med), $S^>$ (those greater) and $S^=$ (those equal to med);
Remove S' from $\mathcal{S}$ and add $S^>$ and $S^<$;
if cardinality $(S^=) > m$ *then begin*
$m \leftarrow$ cardinality $(S^=)$
$mode \leftarrow med$
end
end

On termination of this algorithm, $mode$ will contain a value discovered m times in the list. Furthermore, no element will have been discovered more times, and no heterogeneous subsets of size greater than m will remain. From the nature of the partitioning scheme, it follows that $mode$ is indeed the mode of S.

Paterson, Pippenger, and Schönhage [6] have shown that the median of l elements can be found in $3l$ comparisons. Since we repeatedly subdivide the original list until no piece is of size larger than m, it follows that at most $3n\lceil \log (n/m) \rceil$ comparisons are used. Q.E.D.

At the cost of considerable complication, this constant can be reduced from 3 to 6 $[6 \log 6 - 5 \log 5] \simeq 1.54$. Since this algorithm, which is again based on "requested medians", is complicated and not (provably) optimal, we will not present it, but only suggest the key ideas. The Blum et al. [1] algorithm for determining the median is based on dividing the list into a large number of short sorted lists (in their case of constant length). They then find the median of the medians of these lists, which will lie in the middle half of the original list, and so can be considered a reasonable approximation of the true median from which to continue. Using this idea we can take a set of l elements in sorted lists of lengths

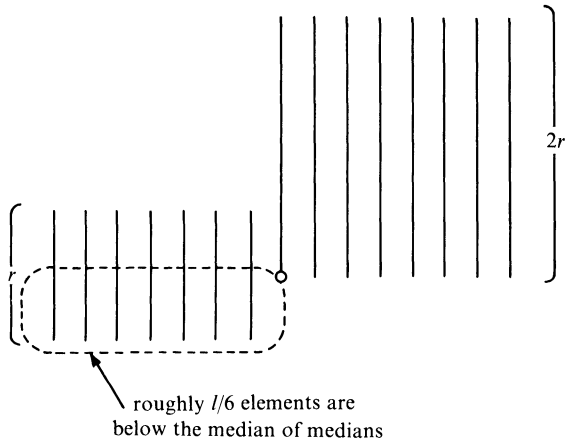


FIG. 1. *In the worst case at least* 1/6 *of the elements lie on either side the median of medians*

between say r and $2r$ and find the "median of medians" in $O(l/r)$ comparisons. Sorted lists lying entirely above, below or equal to this median and lengths between r and $2r$ can be reconstituted in roughly l more comparisons. This median of medians can be seen to lie in the middle 2/3 of the entire set of l elements (see Fig. 1).

If $l/r = o(l)$ then $l + o(l)$ comparisons are used in this basic step. The constant, 1.54, comes from the fact that we are not necessarily splitting the subset at the true median. The choice of r is clearly one of "optimizing", indeed it will "grow" as the process continues and contribute to a lower order term in the "runtime".

Pratt and Yao [7] have shown that at least 1.75 l comparisons are necessary to find the median of l numbers. Our technique is, then, better than any naive application of a median algorithm to compute the mode by finding successive medians.

THEOREM 3.7. *Let A and B be multisets of cardinality n_A and n_B and spectra $M_A = \{m_{A1}, \cdots, m_{Ak}\}$ and $M_B = \{m_{B1}, \cdots, m_{Bj}\}$ where $k \leqq j$. Then to determine whether or not $A \cap B = \varnothing$ requires at least*

$$n_A \log_2 n_A - \sum_{i=1}^{k} m_{Ai} \log_2 m_{Ai} + j \log_2 k - O(n_A \log \log k + n_B)$$

comparisons in the worst case.

Proof. Assume we know that A has k classes and that between any two of them is ordered at least one element of B.

Assume our algorithm eventually determines that $A \cap B = \varnothing$. Then when the algorithm has terminated we shall know the ordering of A and between which classes of A each element of B lies. There are $\binom{n_A}{m_{A1} \cdots m_{Ak}}$ ways to choose elements of A for its classes. If we are given $k - 1$ elements of B which separate the classes of A, there are $(k-1)!$ such separations, and there are then $j - k + 1$ classes of B that are not represented by these $k - 1$ elements. So there are $(k+1)^{n-k+1}$ places to put these elements relative to the classes of A. The bound follows by an argument similar to that in the proof of Theorem 3.3. Q.E.D.

The next corollary generalizes Reingold's result [8].

COROLLARY 3.8. *Let A and B be sets having cardinality n_A and n_B respectively with $n_A \leqq n_B$. Then to tell if $A \cap B = \varnothing$ requires at least*

$$(n_A + n_B) \log_2 n_A - O(n_B)$$

comparisons.

This number is sufficient, for we merely have to sort A and insert elements of B.

In closing, we note that the bound of Theorem 3.7 is probably weak, and we do not have a good algorithm for this problem. In particular, there is no way to know in advance which set has fewer classes.

REFERENCES

[1] M. BLUM, R. W. FLOYD, V. PRATT, R. L. RIVEST AND R. E. TARJAN, *Linear time bounds for median computations*, Proc. 4th Annual ACM Symposium on Theory of Computing, Denver, 1972, pp. 119–124.

[2] R. W. FLOYD, *Treesort*, Algorithm 113, Comm. ACM, 5 (1962), pp. 434.

[3] ———, *Treesort 3*, Algorithm 245, Ibid., 7(1964), p. 701.

[4] D. E. KNUTH, *The Art of Computer Programming, Vol. 3—Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[5] M. PATERSON, Private communication.

[6] M. PATERSON, N. PIPPENGER AND A. SCHÖNHAGE, Private communication.

[7] V. PRATT AND F. F. YAO, *On lower bounds for computing the i-th largest element*, Proc. 14th Annual IEEE Conference on Switching and Automata Theory, Iowa City, 1973, pp. 70–81.

[8] E. M. REINGOLD, *On the optimality of some set algorithms*, J. Assoc. Comput. Mach., 19 (1972), pp. 649–660.

[9] J. W. J. WILLIAMS, *Heapsort*, Algorithm 232, Comm. ACM, 7 (1964), pp. 347–348.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

OPTIMAL ALPHABETIC TREES*

ALON ITAI†

Abstract. An algorithm of Knuth for finding an optimal binary tree is extended in several directions to solve related problems. The first case considered is restricting the depth of the tree by some predetermined integer K, and a Kn^2 algorithm is given. Next, for trees of degree σ, rather than binary trees, $Kn^2 \log \sigma$ and $n^2 \log \sigma$ algorithms are found for the restricted and nonrestricted cases, respectively. For alphabetic trees with letters of unequal cost, a $\sigma^2 n^2$ algorithm is proposed. We conclude with a comparison of alphabetic and nonalphabetic trees and their respective complexities.

Key words. algorithms, alphabetic trees, binary trees, optimal trees, probabilistic search, variable-length codes

1. Introduction. When constructing a code, it is often necessary to minimize the average message length. There is a natural correspondence between a binary prefix code and a binary tree, associating with every leaf a codeword. Assuming that every source word has a fixed probability of occurring, the length of an average message corresponds to the weighted path length of the tree. Let $(w_1, \cdots, w_n)$ denote the weights of the codewords and $l(w_i)$ the length of the path from the root to the weight w_i. The weighted path length (cost) is defined as $WT = \sum_{i=1}^{n} w_i l(w_i)$.

In real time applications, it may be desirable to restrict the length of every codeword. In the tree representation, this restriction is expressed by limiting the length of any path from the root of the tree to a leaf by some integer K.

Similar problems arise when organizing files. Each entry of the file has a certain probability of being requested. There is an order between the entries of the file which must be conserved. No search can take longer than a prespecified maximum. Our aim is to construct a file which fulfills all the requirements and minimizes average search time.

The file is represented as a binary tree with the entries at the nodes. Our aim is to find, among all trees which satisfy the restriction that no path is longer than a given integer K and which preserve the order of the file entries, one with minimum cost.

In the paper, we show that an algorithm of Garey [1], which is an extension of an algorithm of Knuth [4], can be used to solve several problems of this nature.

The running time is of order Kn^2, where n is the number of nodes, K the maximal depth.

In § 7 we show how to extend the results from binary trees to σ-ary trees (i.e., trees in which each node may have up to σ sons). For this problem, we give $n^2 K \log \sigma$ and $n^2 \log \sigma$ algorithms for the restricted and nonrestricted cases, respectively.

* Received by the editors July 30, 1974, and in revised form December 30, 1974.

† Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

Section 8 deals with alphabetic trees with letter of unequal cost. For this problem a $\sigma^2 n^2$ algorithm is proposed. Finally, in the last section, we compare the problem of alphabetic and nonalphabetic trees and their respective complexities.

2. Definitions. A *binary tree* consists of a distinguished node, called *root*, and right and left subtrees of the root. A *binary full tree* consists of a root and left and right subtrees, either both empty or both binary full trees. Nodes occurring in the two subtrees are called *descendants* of the root. All nodes having a given node as a descendant are *ancestors* of that node. If j is the root of a subtree of i, then i is the *father* of j and j is i's *son*. Nodes which have no descendants are called *leaves*. Nodes which are not leaves are called *internal nodes*. From the root there is a unique path to every node i. The length of that path is the number of nodes in the path minus 1 and is called the *depth of node i*, denoted by l_i.

Traversing a tree in postorder (symmetric order) [3] (visit the left subtree, visit the root, visit the right subtree), imposes a full order relation on the nodes. We say that node i is left of node j $(i<j)$ if i precedes j in that order.

An *alphabetic tree* for the sequence of nonnegative weights $(w_1, \cdots, w_n)$ is a binary tree with n leaves, each of which is associated with a weight, such that the leaf of w_i is left of the leaf of w_{i+1} for $i=1, \cdots, n-1$.

The *weighted path length* or *cost* of an alphabetic tree T is given by

$$WT=\sum_{i=1}^{n} w_i l_i,$$

where l_i is the depth of the leaf associated with the weight w_i.

Given a positive integer K a *K restricted alphabetic tree* is an alphabetic tree in which no node has depth greater than K. A K restricted alphabetic tree is optimal if it has minimum cost. In the sequel, we shall use the phrase *optimal tree* in lieu of the cumbersome optimal K restricted alphabetic tree.

3. A dynamic programming solution. We first notice that an optimal tree is a binary full tree (each internal node has exactly two sons), except perhaps when zero weights are involved. However, in this case, there exists an optimal tree which is a full binary tree. Henceforth, we shall assume that all optimal trees are full binary trees.

Let T be an optimal tree, v an internal node of T of depth l_v; then the subtree of T with root v is an optimal $(K-l_v)$ restricted alphabetic tree for its own sequence of weights.

Let $[i, j, k]$, $1 \leqq i \leqq j \leqq n$ and $0 \leqq k \leqq K$, denote the subproblem of finding an optimal tree of depth at most k for $(w_i, w_{i-1}, \cdots, w_j)$. If $k < \lfloor \log_2 (j-i+1) \rfloor$, then no solution exists, and we may set $WT[i, j, k]=\infty$. Otherwise set

$$WT[i, i, k]=0, \qquad i=1, \cdots, n,$$

$$WT[i, i, k]=\sum_{r=i}^{j} w_r+\min_{i \leqq b<j}\{WT[i, b, k-1]+WT[b+1, j, k-1]\}, \tag{1}$$

$$i=1, \cdots, n-1, j=i+1, \cdots, n.$$

The value b_0, for which (1) produces the minimum, is the last node (in postorder) of the left subtree and is called the *breakpoint*. If b_0 is a breakpoint for $[i, j, k]$, we

can construct an optimal tree for $[i, j, k]$ by constructing an optimal tree for $[i, b_0, k-1]$ and $[b_0+1, j, k-1]$ and combining these two trees. Solving $[i, j, k]$ for all k, i, j (looping first on $k = 1, \cdots, K$, then on $p = j-1 = 1, \cdots, n-1$, then on $i = 1, \cdots, n-p$) results in an optimal tree for $[1, n, K]$ the original problem.

After finding the minimum cost, we may reconstruct the tree had we saved the breakpoints b_0 for every $[i, j, k]$.

The resulting algorithm requires execution time proportional to Kn^3. Gilbert and Moore [2] proposed a similar algorithm for trees with no restriction on maximal depth. See that paper for further description and analysis of the algorithm.

4. An improved algorithm. We may improve the previous algorithm by observing the following phenomena.

THEOREM 1. *Let b_0 be the breakpoint of an optimal tree for $[1, n, k]$. Then there exists an optimal tree for $[1, n+1, k]$ whose breakpoint b_0' satisfies $b_0' \geqq b_0$.*

Let $b[i, j, k]$ denote a breakpoint of the subproblem $[i, j, k]$ and $b^l[i, j, k]$ the breakpoint of the optimal tree with the smallest breakpoint *leftmost tree.*

THEOREM 2. *There exist optimal trees such that:*

(a) $b[i, j-1, k] \leqq b[i, j, k] \leqq b[i+1, j, k]$. *Moreover, the leftmost trees satisfy* (a); *in other words,*

(b) $b^l[i, j-1, k] \leqq b^l[i, j, k] \leqq b^l[i+1, j, k]$.

We shall prove the theorems in the next section. Let us first see how we can use these results to obtain an improved algorithm. For fixed k and fixed $p = j-1$, the smallest breakpoint of $[i, j, k]$-$b^l[i, j, k]$ lies between $b^l[i, j-1, k]$ and $b^l[i+1, j, k]$ (Theorem 2(b)). Therefore, we need only consider $b^l[i+1, j, k] - b^l[i, j, k]$ possible breakpoints. Summing over i and substituting $j = i+p$, we find that

$$\sum_{i=1}^{n-p} (1 + b^l[i+1, i+p, k] - b^l[i, i-1+p, k])$$

$$= (n-p) + b^l[n-p+1, n, k] - b^l[1, p, k] \leqq 2n.$$

The improved algorithm determines $b^l[i, j, k]$ and $WT[i, j, k]$ using previously obtained values. The outer loops on k and p introduce a factor of nK. The whole algorithm takes time proportional to n^2K.

Note that the results of Theorem 2(a) are not sufficient since there may be optimal trees which satisfy $b[i, j-1, k] > b[i+1, j, k]$. We overcome this difficulty by insisting on the smallest breakpoint. This refinement does not increase the difficulty of implementing the algorithm and is usually incorporated for the sake of programming convenience [4, p. 22].

Knuth [4] first proposed this algorithm for the unrestricted case. Garey [1] extended it to the restricted case when $w_1 \leqq w_2 \leqq \cdots \leqq w_n$. Here we show that this additional requirement is superfluous.

5. Proof of Theorems 1 and 2. Let T, T' be trees of depth not exceeding K. Let a be a node of T, a' a node of T'. Both T and T' are subgraphs of a binary full tree and can be compared by the "left of" relation; i.e., a left of a' $(a < a')$, $a = a'$ or a right of a' $(a' < a)$. Let x_i denote the position of weight w_i in the tree T, x_i' the

position of w_i in T'. In what follows, we shall assume $k \geqq \log_2 (n+1)$. First we notice that by adding to the right a new node of weight $w_{n+1} = 0$ w_n moves to the left.

LEMMA 3. *Let* $T = T[1, n, k]$ *be an optimal tree for weights* $(w_1, \cdots, w_n)$. *There exists an optimal tree* $T' = T'[1, n+1, k]$ *for weights* $(w_1, \cdots, w_n, 0)$ *such that* $x'_n < x_n$.

Proof. Case 1. x_n is of maximal depth. (i.e., $l_n = k$). In this case, $x'_{n+1} \leqq x_n$ since $x'_n < x'_{n+i}$ by transitivity $x'_n < x_n$.

Case 2. x_n is not of maximal depth (i.e., $l_n < k$). In this case, we may obtain an optimal tree by replacing w_n by a node whose two sons are w_n, w_{n+1}. The resulting tree T' satisfies $WT' = WT + w_n$. To show that T' is optimal, assume the existence of a tree $T^2 = T^2[1, n+1, k]$ which costs less than T', $WT^2 < WT'$. We may construct a tree $T^3 = T^3[1, n, k]$ by deleting w_{n+1} from T^2. This would enable us to decrease the depth of w_n. We then have

$$WT^3 \leqq WT^2 - w_n < (WT + w_n) - w_n = WT,$$

which contradicts the optimality of T. Q.E.D.

Next we note that w_n does not have to move right when w_{n+1} increases.

LEMMA 4. *Let* $T = T[1, n+1, k]$ *be an optimal tree for weights* $(w_1, \cdots, w_n, w_{n+1})$. *There exists an optimal tree* $T' = T'[1, n+1, k]$ *for weights* $(w_1, \cdots, w_n, w'_{n+1})$, $w'_{n+1} > w_{n+1}$, *such that* $x'_n \leqq x_n$.

Proof. If the tree T remains optimal for weights $(w_1, \cdots, w_n, w_{n+1})$, we are done since we can choose $T' = T$. Otherwise, let T' be any optimal tree for $(w_1, \cdots, w'_{n+1})$. Consider the cost of T and T' as functions of the $(n+1)$st weight, w: $WT(w) = w_1 l_1 + \cdots + w_n l_n + w l_{n+1} = C + l_{n+1} w$, $WT'(w) = w_1 l'_1 + \cdots + w_n l'_n + w l_{n+1} = C' + l'_{n+1} w$. The difference in cost $D(w) = WT'(w) - WT(w) = C' - C + (l'_{n+1} - l_{n+1}) w$ is a linear function of w, and we know that $D(w'_{n+1}) < 0 \leqq D(w_{n+1})$.

It follows that $l'_{n+1} < l_{n+1}$. In particular, the father z of x_{n+1} is a node of the right subtree of the father z' of x'_{n+1}. Since x'_n is in the left subtree of z' and x_n is a descendant of z, we have $x'_n < x_n$. In both cases we find that $x'_n \leqq x_n$. Q.E.D.

If we add a new weight w_{n+1} to the right, the weight w_n moves left, because we can do this in two stages: First add a node of weight $w_{n+1} = 0$. This enables us to move w_n to the left. Then increase the weight of the node. This may be done without moving w_n to the right. Thus we have the following corollary.

COROLLARY 5. *If* $T = T[1, n, k]$ *is an optimal tree for* $(w_1, \cdots, w_n)$, *there exists an optimal tree* $T' = T'[1, n+1, k]$ *for* $(w_1, \cdots, w_{n+1})$ *such that* $x'_n < x_n$.

We wish to show that by adding a new node, we form a tree in which none of the previous weights moves right. It has already been shown that we can construct an optimal tree in which w_n moves left. We shall now show that this requirement may be extended, and we can find an optimal tree in which no weight moves to the right.

Some notation follows: For Q a binary tree, let SQ denote the set consisting of all indices of the weights of Q. WQ as before denotes the cost of Q.

LEMMA 6. *Let* $T = T[1, n, k]$ *be an optimal tree for* $(w_1, \cdots, w_n)$. *There exists an optimal tree* $T' = T'[1, n+1, k]$ *for* $(w_1, \cdots, w_{n+1})$ *such that* $x'_i \leqq x_i$, $i = 1, \cdots, n$.

Proof. Assuming to the contrary that no such optimal tree exists, let T' be an optimal tree for weights $(w_1, \cdots, w_{n+1})$ in which some weights move right. Let w_{j-1} be the last weight which moves right. ($x_{j-1} < x'_{j-1}$ but $x'_i \leqq x_i$ for $i = j, \cdots, n$.) By Corollary 5, we may assume $j \leqq n$. Let z be the deepest common ancestor of x_{j-1} and x_j (i.e., a node z which is an ancestor of both x'_{j-1} and x'_j, and no descendant of z has this property).

Comparing x'_{j-1} and x'_j to z, there are three possibilities:

(a) w_{j-1} and w_j do not pass z ($x_{j-1} < x'_{j-1} < z < x'_j \leqq x_j$);

(b) w_{j-1} passes z to the right ($x_{j-1} < z < x'_{j-1} < x'_j \leqq x_j$);

(c) w_j passes z to the left ($x_{j-1} < x'_{j-1} < x'_j < z < x_j$).

(We cannot have x'_{j-1} (or x'_j) equal to z since this would force x'_j to move right of x_j (or x'_{j-1} to move left of x_{j-1}).) z must be a node of T', since otherwise x'_{j-1} would be forced right of z, and so would all its descendants, in particular $x_j < x'_{j-1}$. This contradicts the hypothesis that $x'_{j-1} < x'_j \leqq x_j$.

Let $L(z)$ be the partial tree containing all the nodes left of z in T and the paths connecting them to the root of T. Define $R(z)$ in the same fashion on the nodes right of z in T. $L'(z)$ and $R'(z)$ are the analogous partial trees of T'.

Case (a). w_{j-1} and w_j do not pass z. $SL(z) = \{1, \cdots, j-1\} = SL'(z)$ (the leaves of $L(z)$). Construct T^2 from $L(z)$ and $R'(z)$. The leaves of T^2 are $\{1, \cdots, n+1\}$ the same as of T'.

$$WT^2 = WL(z) + WR'(z) \geqq WT' = WL'(z) + WR'(z).$$

If $WT^2 = WT'$ we have constructed, contrary to the hypothesis, an optimal tree for $(w_1, \cdots, w_n, w_{n+1})$ in which no weights move right.

Therefore $WL(z) > WL'(z)$. Construct T^3 from $L'(z)$ and $R(z)$. $WT^3 = WL'(z) + WR(z) < WL(z) + WR(z) = WT$. T^3 consists of the same weights as T, so the last inequality contradicts the optimality of T.

Case (b). w_{j-1} passes z to the right. In this case,

$$\begin{aligned} SL(z) &= \{1, \cdots, j-1\}, & SL'(z) &= \{1, \cdots, i-1\}, \\ SR(z) &= \{j, \cdots, n\}, & SR'(z) &= \{i, \cdots, n+1\}, \end{aligned} \qquad i < j.$$

Let M' be a tree partial to T', which consists of all the leaves $SL \cap SR'$ and the paths connecting them to the root of T. ($SM' = SL \cap SR' = \{i, \cdots, j-1\}$ is the set of indices of weights which have moved to the right of z.)

When w_{j-1} moves right it must stay left of $x'_j \leqq x_j$. $x'_j \neq x_j$ since we must make space for w_{j-1}. Consequently, w_j moves down left. x_j, the old position of w_j, is an internal node of T'; and as it is an ancestor of x'_{j-i}, it is an internal node of M' with all members of SM' belonging to its left subtree.

Construct a partial tree T^2 consisting of $L(z)$ and $R'(z)$ from which we have deleted the elements of SM'. This enables us to bring w_j up one level. Possibly w_j can be raised by more than that. However, for simplicity we shall raise it by exactly one level.

$$WT^2 = WL(z) + WR'(z) - w_j - WM' \geqq WT' = WL'(z) + WR'(z).$$

The inequality arises from the fact that T^2 and T' are built on the same weights ($ST' = ST^2 = \{1, \cdots, n+1\}$) and T' is optimal.

In case of equality, T^2 is an optimal tree in which no weight has moved right. Otherwise, $WT^2 > WT'$. Construct the tree T^3 from $L'(z)$ and $R(z)$. The elements of SM' are still missing. M' is a partial tree whose root is the root of T'. We may insert M' such that its internal nodes above x_j coincide with the corresponding nodes of $R(z)$. Node x_j of $R(z)$ can be moved down-right one level. This makes space for inserting the rest of M' (which includes all of its leaves).

$$WT^3 = WL'(z) + (WM' + w_j) + WR(z) < WL(z) + WR(z) = WT.$$

As T^3 and T have the same set of weights, this contradicts the optimality of T.

Case (c). w_j moves left of z. This case is analogous to (b), and the proof goes along the same lines. Q.E.D.

Theorem 1 is now an easy consequence of Lemma 6, since the optimal tree found in Lemma 6 fulfills all the requirements of Theorem 1.

Now instead of adding a weight let us consider deleting one. The following lemma will enable us to prove Theorem 2.

LEMMA 7. *Let $T[1, n, k]$ be an optimal tree. There exists an optimal tree $T'[2, n, k]$ such that $x'_i \leqq x_i$ for $i = 2, \cdots, n$.*

The proof is similar and, in some respects, simpler than that of Lemma 6 and, therefore, will not be pursued here.

We are now ready to prove Theorem 2. The first inequality of part (a), $b[i, j-1, k] \leqq b[i, j, k]$, is just Theorem 1. The second inequality, $b[i, j, k] \leqq b[i+1, j, k]$, follows from Lemma 7 just as Theorem 1 from Lemma 6.

Part (b), $b'[i, j-1, k] \leqq b'[i, j, k]$, is true since if $b'[i, j, k] < b'[i, j-1, k]$, then by the second inequality of part (a) and symmetry there exists an optimal tree $T'[i, j-1, k]$ such that

$$b'[i, j-1, k] \leqq b'[i, j, k] < b'[i, j-1, k].$$

This contradicts the definition of $T'[i, j-1, k]$ as an optimal tree with the smallest breakpoint. The second inequality follows similarly and concludes the proof of Theorem 2.

6. Applications and related problems. The results of the previous sections lend themselves to several applications. First consider the K restricted Huffman problem [1]. Given n source words, each with a predetermined frequency of occurrence, construct a code with minimum average message length satisfying the restriction that no codeword has length greater than K. Just as in Huffman's original problem, we consider the order between the source words immaterial. The following lemma enables us to use the previous results to solve this problem.

LEMMA 8. *If $K \geqq \lceil \log_2 n \rceil$ and $w_1 \geqq \cdots \geqq w_n$, there exists an optimal K restricted tree for $(w_1, \cdots, w_n)$ such that the leaves associated with the weights preserve the order* (i.e., *x_i is left of x_{i+1}*, $i = 1, \cdots, n-1$). The proof is well known [3].

As a consequence of this lemma, we obtain an algorithm for the K-restricted Huffman problem:

(a) Sort the weights such that $w_1 \geqq w_2 \geqq \cdots \geqq w_n$.

(b) Apply the algorithm of § 4 to the new sequence. This is the algorithm which was proposed by Garey [1].

As previously mentioned, a nonrestricted version of the algorithm was first suggested by Knuth [4]. He used the algorithm to find an optimal alphabetic tree in which information may be stored in the internal nodes as well as in the leaves.

The extended algorithm may be applied to the K restricted version of Knuth's problem; i.e., find a tree of minimum cost in which weights are associated with all nodes, the order of the weights is preserved and no path has depth greater than K. The proof is essentially similar to that of § 5 and will not be presented.

Note that Knuth's original problem may be viewed as a special case of the K-restricted problem when K is sufficiently large ($K \geqq n$).

7. Alphabetic trees of degree $\sigma > 2$. Up to now, all algorithms heavily depended on the fact that we dealt with binary trees. Even though binary codes are of greatest importance, it is worthwhile to consider codes over an alphabet of more than two letters. Similarly, there are applications of σ-ary search trees for $\sigma > 2$.

Define a *tree of degree σ* (*σ-ary tree*) as consisting of a distinguished node, *root* and at most σ *subtrees* (the subtrees are ordered). Alphabetic σ-ary trees are defined similarly to alphabetic binary trees. Our object is to find optimal alphabetic σ-ary trees, i.e., such trees of minimum cost. We shall outline how to find optimal trees of nonrestricted depth. The extension to trees of restricted depth is straightforward.

Let an s-forest be a sequence of s trees. The cost of an s-forest is the sum of the costs of all its trees. Denote an optimal s-forest on weights $(w_i, \cdots, w_j)$ by $F_s[i, j]$. Note that $F_1[i, j] = T[i, j]$. The cost of the s-forest, $s > 1$, is

$$WF_s[i, j] = \min_{i \leqq b < j} \{WF_{s'}[i, b] + WF_{s-s'}[b+1, j]\}$$

for any s' $1 \leqq s' < s$. The cost of a σ-ary trees is

$$WT[i, j] = W_{ij} + WF_\sigma[i, j], \quad \text{where } W_{ij} = \sum_{r=i}^{j} w_r.$$

Using these equations we may extend the dynamic programming solutions of the previous sections to find optimal trees and optimal forests. When $\delta = j - i \geqq \sigma$, there is always an optimal σ-ary tree for $(w_i, \cdots, w_j)$ with exactly σ subtrees.

For each $\delta = 1, \cdots, n-1$, we shall find optimal σ-trees and s-forests for weights $(w_i, \cdots, w_j)$, $j = i + \delta$. The values of s for which we shall find optimal s-forests will depend on σ as follows: First, include the numbers $2, 4, 8, \cdots, 2^{\lfloor \log_2 \sigma \rfloor}$; next include the sequence $(s_0, \cdots, s_m)$, where $s_0 = \sigma$, $s_{i+1} = s_i - 2^{\lfloor \log_2 s_i \rfloor}$ and s_m is the last number which is not a power of 2. From the construction, it is clear that a forest of s_i trees can be constructed by combining a forest of s_{i+1} trees and a forest of $2^{\lfloor \log_2 s_i \rfloor}$ trees. As $m \leqq \lfloor \log_2 \sigma \rfloor$, we conclude that we need to consider at most $2\lfloor \log_2 \sigma \rfloor$ values of s.

Applying this extension to Gilbert and Moore's [2] algorithm yields an $n^3 \log \sigma$ algorithm.

Given a forest $F_s[i, j]$, let s be constructed as $s = s' + (s - s')$ in the above sequence; define the *breakpoint* as the index of the rightmost weight of the s'th tree.

Using this definition, we notice that Theorems 1, 2 hold also for s-forests and σ-ary trees. This enables us to take advantage of Knuth's observation and cut the running time down to $n^2 \log_2 \sigma$.

8. Alphabetic trees with letters of unequal cost. An interesting problem arises when we consider constructing an optimal alphabetic tree with letters of unequal cost. Specifically, for an alphabet of σ letters, let $(c_1, \cdots, c_\sigma)$ be the *cost of the letters*. The *cost of an edge* (a, b) in a σ-ary tree is c_i where b is the ith son of a. (In general, the σ-ary tree need not be full; i.e., not every node has σ sons. For $\sigma = 5$, a node may have a second and fourth son while lacking the first, third and fifth sons.) The *cost of a node* is the sum of the costs of all the edges of the unique path from the root to that node. For weights $(w_1, \cdots, w_n)$, an *alphabetic tree* is a σ-ary tree with weights on the nodes such that if $i < j$, then w_i is left of w_j. The *cost of an alphabetic tree* is $WT[1, n] = \sum_{i=1}^{n} p_i w_i$, where p_i is the cost of the node associated with w_i.

Our aim is to find, for given weights, an alphabetic tree of minimum cost (optimal tree). Note that, as the tree is alphabetic, we may not assume that the costs of the letters satisfy $c_1 \leqq c_2 \leqq \cdots \leqq c_\sigma$. (We may well have $c_1 < c_2$, $c_2 > c_3$.) We shall consider the case in which the weights are only at the leaves.

Let $T_{\alpha,\beta}[i, j]$ be an optimal tree for weights $(w_i, \cdots, w_j)$ in which the root has no son smaller than α and none greater than β. For every α,β,i,j $(1 \leqq \alpha \leqq \beta \leqq \sigma;\ 1 \leqq i \leqq j \leqq n)$, we shall find $WT_{\alpha,\beta}[i, j]$, the cost of an optimal tree $T_{\alpha,\beta}[i, j]$. (This implies that $T_{\alpha,\beta}$ has at least one edge).

The following equations will provide means to calculate $WT_{\alpha,\beta}[i, j]$:

$$WT_{\alpha,\alpha}[i, i] = c_\alpha w_i;$$

the tree $T_{\alpha,\alpha}[i, i]$ uses at least one edge, in this case that edge must be α.

$$WT_{\alpha,\beta}[i, i] = \min_{\alpha \leqq \gamma \leqq \beta} \{WT_{\gamma,\gamma}[i, i]\}, \qquad \alpha < \beta;$$

the root must have exactly one son (edge γ).

$$(2) \qquad WT_{\alpha,\alpha}[i, j] = c_\alpha W_{ij} + \min_{\substack{i \leqq b < j \\ 1 \leqq \gamma < \sigma}} \{WT_{\gamma,\gamma}[i, b] + WT_{\gamma+1,\sigma}[b+1, j]\}, \ i < j.$$

The root has exactly one son (edge α). The son must have at least two sons; let the first be γ. We choose γ and b, the index of the rightmost weight of $T_{\gamma,\gamma}[i, b]$, so as to minimize the sum of the cost of the two trees—$WT_{\gamma,\gamma}[i, b] + WT_{\gamma+1,\sigma}[b+1, j]$.

$$WT_{\alpha,\beta}[i, j] = \min_{i \leqq b < j} \{WT_{\alpha+1,\beta}[i, j],\ WT_{\alpha,\alpha}[i, b] + WT_{\alpha+1,\beta}[b+1, j],\ WT_{\alpha,\alpha}[i, j]\}, \qquad \alpha < \beta,\ i < j.$$

We should consider three cases: In the first, it is not worthwhile to use edge α at all, hence the optimal tree is $WT_{\alpha+1,\beta}[i, j]$; in the second, α is truly the first edge, but there are additional edges. The cost of the optimal tree is $WT_{\alpha,\alpha}[i, b]$

$+ WT_{\alpha+1,\beta}[b+1, j]$. The last case is when only edge α is used (this might make some smaller edges available). The cost then is $WT_{\alpha,\alpha}[i, j]$. (Now we shall use (2) above, where $\gamma < \sigma$ and $b < j$, and hence we are not led into an infinite loop).

We compute $WT_{\alpha,\beta}[i, j]$ in order of increasing $j - i = p$, and for a fixed value of p in order of increasing $\beta - \alpha$. This insures that we never reference values which have not yet been computed.

This algorithm can be improved by use of the fact that adding an additional rightmost node does not cause any weight to move to the right. The statement and proof are similar to that of Theorems 1, 2. This enables us to reduce the computation time by a factor of n to $\sigma^2 n^2$.

Thus, if we consider σ fixed (e.g., $\sigma = 2$), the requirement for letters of different cost does not increase the known time complexity of the problem.

9. Alphabetic trees and nonalphabetic trees. The problem of alphabetic trees is closely related to that of nonalphabetic trees (i.e., trees in which the order of the weights is immaterial). However, the relation between the complexity of the two problems is unclear. For given weights, there are $n!$ times more nonalphabetic trees than alphabetic trees. This implies a larger domain for the nonalphabetic case. However, the lack of the order constraint may help us in our search.

We shall now give a closer look at some of the problems and show the relationship between the alphabetic and the nonalphabetic cases.

Case (a). Binary trees, weights at all nodes, letters of equal cost. Knuth [4] considered the alphabetic problem and found an n^2 algorithm. For the nonalphabetic case, there exists a linear solution as follows:

First notice that for weights $(w_1, \cdots, w_n)$ all optimal trees must satisfy

(i) if $w_i < w_j$, then $l_i \geqq l_j$;

(ii) all nodes, except perhaps those of the last two levels, have exactly two sons.

Without loss of generality, $n = 2^l - 1$ (otherwise add weights $(w_{n+1}, \cdots, w_{2^l-1} = 0)$, $2^l < 2n$, when the tree is found delete those weights). For such n, the optimal tree is the complete binary tree of depth $l - 1$. (The shape of the tree does not depend on the weights, only on their number.)

We now have to distribute the weights in the tree. This can be done in linear time by means of the median algorithm [5]. The deepest level will contain all nodes less than or equal to the median. The second deepest level will contain all nodes less than or equal the median of the remaining nodes.

$$\text{Time } (n) = CN + Cn/2 + Cn/4 + \cdots + C1$$

$$\leqq Cn(1 + \tfrac{1}{2} + \tfrac{1}{4} + \cdots) \leqq 2Cn.$$

C is the constant of the median algorithm. The internal order of the nodes at each level is arbitrary, since it does not change the cost of the tree.

Case (b). σ-ary trees, weights only at the leaves, letters of equal cost. The nonalphabetic case is Huffman's problem, for which his algorithm yields an $n \log n$ solution. In the alphabetic case with $\sigma = 2$, the T-C algorithm [6], [7] also yields an $n \log n$ algorithm. For $\sigma > 2$, we may apply the algorithm of § 7, the time bound of which is $n^2 \log \sigma$.

Case (c). Binary trees, weights at all nodes, letters of unequal cost. For the nonalphabetic case, as in (a) above, the shape of the tree is independent of the actual weights. Therefore, we can find a tree of n minimum cost vertices in time $n \log n$; number the nodes so that cost (node$_i$) $\leqq$ cost (node$_{i+1}$), $i = 1, \cdots, n-1$.

Order the weights so that $w_1 \geqq w_2 \geqq \cdots \geqq w_n$; then associate with the ith node the ith weight. The optimality of the tree resulting from this algorithm is proven similarly to that of (a) above. The time bound on the algorithm is $n \log n$.

For alphabetic trees an algorithm similar to that of § 7 yields an n^2 solution.

Case (d). σ-ary trees, weights only at the leaves, letters of unequal cost. No polynomially bound algorithm is known—see [8] for an integer programming solution.

In § 8, we have given a $\sigma^2 n^2$ solution for the alphabetic case. It is not known whether it is possible to reduce the nonalphabetic case to the alphabetic case since no result similar to Lemma 8 of § 6 is known to hold.

Case (e). Restricting the maximum depth of the tree to K. For the alphabetic case, this causes an additional factor of K. For the nonalphabetic case, (a) and (c) retain their previous bounds, whereas the appropriate variant of (b) requires a reduction to the alphabetic case (see Lemma 8, § 6).

In conclusion, we may say that in most cases there are slightly better algorithms for the nonalphabetic case. However, there is a case (d) for which no polynomially bound algorithm is known for the nonalphabetic case, whereas the alphabetic case has a polynomial algorithm.

REFERENCES

[1] M. R. GAREY, *Optimal binary search trees with restricted maximal depth*, this Journal, 2 (1974), pp. 101–110.

[2] E. N. GILBERT AND E. F. MOORE, *Variable-length binary encodings*, Bell System Tech. J. 38 (1959), pp. 933–968.

[3] DONALD E. KNUTH, *The Art of Computer Programming, 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[4] ———, *Optimum binary search trees*, Acta Informat., 1 (1971), pp. 14–25.

[5] ———, *The Art of Computer Programming, 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[6] T. C. HU AND A. C. TUCKER, *Optimum computer search trees*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[7] T. C. HU, *A new proof of the T-C algorithm*, Ibid., 25 (1973), pp. 83–94.

[8] R. M. KARP, *Minimum-redundancy coding for the discrete noiseless channel*, IEEE Trans. Information Theory, IT-7 (1961), pp. 27–39.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

PARTIAL-MATCH RETRIEVAL ALGORITHMS*

RONALD L. RIVEST†

Abstract. We examine the efficiency of hash-coding and tree-search algorithms for retrieving from a file of k-letter words all words which match a partially-specified input query word (for example, retrieving all six-letter English words of the form S**R*H where "*" is a "don't care" character).

We precisely characterize those balanced hash-coding algorithms with minimum average number of lists examined. Use of the first few letters of each word as a list index is shown to be one such optimal algorithm. A new class of combinatorial designs (called associative block designs) provides better hash functions with a greatly reduced worst-case number of lists examined, yet with optimal average behavior maintained. Another efficient variant involves storing each word in several lists.

Tree-search algorithms are shown to be approximately as efficient as hash-coding algorithms, on the average.

In general, these algorithms require time about $O(n^{(k-s)/k})$ to respond to a query word with s letters specified, given a file of n k-letter words. Previous algorithms either required time $O(s \cdot n/k)$ or else used exorbitant amounts of storage.

Key words. searching, associative retrieval, partial-match retrieval, hash-coding, tree-search, analysis of algorithms

QUOTATIONS

Oh where, oh where, has my little dog gone?
Oh where, oh where can he be?
With his tail cut short, and his ears cut long,
Oh where, Oh where can he be?

[Nursery rhyme]

You must look where it is not, as well as where it is.
[*Gnomologia—Adages and Proverbs*, by T. Fuller (1732)]

1. Introduction.

We examine algorithms for performing partial-match searches of a direct-access file to determine their optimal forms and achievable efficiency. First we present a model for file and query specifications, within which we will analyze various search algorithms. Section 2 discusses the historical development of the "associative search" problem and reviews previously published search algorithms. Section 3 examines generalized hash-coding search algorithms, and § 4 studies a tree-search algorithm.

We begin with a general outline of an information retrieval system, and then proceed to define our problem more precisely.

An information retrieval system consists of the following parts:

(i) a collection of information, called a *file*. An individual unit of a file is called a *record*.

(ii) a storage or recording procedure by which to represent a file (in the abstract) on a physical medium for future reference. This operation we call *encoding* a file. The encoded version of a file must, of course, be distinguishable

* Received by the editors February 21, 1974, and in revised form February 19, 1975. This work was supported by the National Science Foundation under Grants GJ-331-70X and GJ-43634X.

† Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

from the encoded versions of other files. The medium used is entirely arbitrary; for example, punched or printed cards, ferromagnetic cores, magnetic tape or disk, holograms or knotted ropes. There are clearly many possible encoding functions, even for a given storage medium. To choose the best one for an application is the *encoding* or *data structure* problem.

(iii) a method by which to access and read (or decode) an encoded file. The access method depends only on the storage medium used, while the encoding function determines what interpretation should be given to the accessed data. The encoded version of the file will, in general, consist of the encodings of its constituent records, together with the encoding of some auxiliary information. If (the encoded version of) any particular record can be independently accessed in (approximately) unit time, we say the file is stored on a direct-access storage device. (Magnetic disks are direct-access, whereas magnetic tapes are not.) The access time usually consists of two independent quantities: the *physical access* time needed to move a reading head or some other mechanical unit into position, and the *transmission* time required to actually read the desired data. The transmission time is proportional to the amount of information read, while the physical access time may depend on the relative location of the last item of information read.

(iv) a user of the system, who has one or more queries (information requests) to pose to the system. *The response to a query is assumed to be a subset of the file*—that is, the user expects some of the file's records to be retrieved and presented to him. If the user presents his queries one at a time in an interactive fashion, we say that the retrieval system is being used *on-line*, otherwise we say that it is being used in *batch* mode. We shall only consider on-line systems.

(v) a search algorithm. This is a procedure for accessing and reading part of the encoded file in order to produce the response to a user's query. It is, of course, dependent, but not entirely, on the choice of storage medium and encoding function. This algorithm may be performed either by a computer or some individual who can access the file (such as a librarian).

The above broad outline of an information retrieval system needs to be fleshed out with more detail in order to make precise the problem to be studied. We now present some formal definitions required for the rest of this paper. These details restrict the model's generality somewhat, although it remains a good approximation to a large class of practical situations.

1.1. Attributes, records and files. A *record* r is defined to be an ordered k-tuple $(r_1, r_2, \cdots, r_k)$ of values chosen from some finite set Σ. That is, each record is an ordered list of k *keys*, or *attributes*. For convenience, we assume that $\Sigma = \{0, 1, \cdots, v-1\}$, so that the set Σ^k is the set of all k-letter words over the alphabet Σ, and has size v^k. We shall generally restrict our attention to the case $v = 2$, so that the set Σ^k is just the set of k-bit words. Since any record type can be encoded as a binary string, this entails no great loss in generality. Furthermore, it seems to be the case that binary records are the hardest type for which to design partial-match retrieval algorithms, since the user has the greatest flexibility in specifying queries for a given number of valid records.

A *file* F is defined to be any nonempty subset of Σ^k.

These definitions are not the most general possible; however, they do model a significant fraction of practical applications. A more general scheme, such as that proposed by Hsiao and Harary [25], would define a record to be an unordered collection of (attribute, value) pairs, rather than an ordered list of values for a predetermined set of attributes, as we have chosen here. Making realistic assumptions about the frequency of occurrence of each attribute, and related problems, seems to be difficult, and we shall not pursue these questions in this paper.

1.2. Queries. Let Q denote the set of queries which the information retrieval system is designed to handle. For a given file F, the proper response to a query $q \in Q$ is denoted by $q(F)$ and is assumed to be a (perhaps null) subset of F.

The following sections categorize various query types and describe the particular query types to be considered in this paper.

1.2.1. Intersection queries. The most common query type is certainly the *intersection query*, named after a characteristic of its response definition: a record in F is to be retrieved if and only if it is in a specified subset $q(\Sigma^k)$ of Σ^k, so that

$$q(F) \stackrel{\text{def}}{=} F \cap q(\Sigma^k).$$

(This notation is consistent since $F = \Sigma^k$ implies $q(F) = q(\Sigma^k)$.) The sets $q(\Sigma^k)$ completely characterize the functions $q(F)$ for any file F by the above intersection formula. Intersection queries enjoy the property that whether some record $r \in F$ is in $q(F)$ does not depend upon the rest of the file (that is, upon $F - \{r\}$) so that no "global" dependencies are involved. The class of intersection queries contains many important subclasses which we present in a hierarchy of increasing generality:

1. *Exact-match queries.* Each $q(\Sigma^k)$ contains just a single record of Σ^k. An exact-match query thus asks whether a specific record is present in F.

2. *Single-key queries.* $q(\Sigma^k)$ contains all records having a particular value for a specified attribute. For example, consider the query defined by $q(\Sigma^k) = \{r \in \Sigma^k | r_3 = 1\}$.

3. *Partial-match queries.* A "partial-match query q with s keys specified" (for some $s \leqq k$) is represented by a record $\hat{r} \in R$ with $k - s$ keys replaced by the special symbol "*" (meaning "unspecified"). If $\hat{r} = (\hat{r}_1, \cdots \hat{r}_k)$, then for $k - s$ values of j, we have $\hat{r}_j =$ "*". The set $q(\sum^k)$ is the set of all records agreeing with $\hat{r}$ in the specified positions. Thus

$$q(\Sigma^k) = \{r \in \Sigma^k | (\forall j, 1 \leqq j \leqq k)[(\hat{r}_j = *) \vee (\hat{r}_j = r_j)]\}.$$

A sample application might be a crossword puzzle dictionary, where a typical query could require finding all words of the form "B*T**R" (that is: BATHER, BATTER, BETTER, BETTOR, BITTER, BOTHER, BUTLER, BUTTER). We shall use Q_s throughout to denote the set of all partial-match queries with s keys specified.

4. *Range queries.* These are the same as partial-match queries except that a range of desired values rather than just a single value may be specified for each attribute. For example, consider the query defined by

$$q(\Sigma^k) = \{r \in \Sigma^k | (1 \leqq r_1 \leqq 3) \wedge (1 \leqq r_2 \leqq 4)\}.$$

5. *Best-match queries with restricted distance.* These require that a distance function d be defined on Σ^k. Query q will specify a record $\hat{r}$ and a distance λ, and have

$$q(\Sigma^k) = \{r \in \Sigma^k | d(r, \hat{r}) \leqq \lambda\}.$$

Query q requests all records within distance λ of the record $\hat{r}$ to be retrieved. the distance function $d(r, \hat{r})$ is usually defined to be the number of attribute positions for which r and $\hat{r}$ have different values; this is the Hamming distance metric.

6. *Boolean queries.* These are defined by Boolean functions of the attributes. For example, consider the query q defined by

$$q(\Sigma^k) = \{r \in \Sigma^k | ((r_1 = 0) \vee (r_2 = 1)) \wedge (r_3 \neq 3)\}.$$

The class of Boolean queries is identical to the class of intersection queries, since one can construct a Boolean function which is true only for records in some given subset $q(\Sigma^k)$ of Σ^k (the *characteristic function* of $q(\Sigma^k)$).

Note that each intersection query requires *total recall;* that is, *every* record in F meeting the specification must be retrieved. Many practical applications have limitations on the number of records to be retrieved, so as not to burden the user with too much information if he has specified a query too loosely.

1.2.2. Best-match queries. A different query type is the *pure best-match query.* A pure best-match query q requests the retrieval of all the nearest neighbors in F of a record $\hat{r} \in \Sigma^k$ using the Hamming distance metric d over Σ^k. Performing a pure best-match search is equivalent to decoding the input word $\hat{r}$ into one or more of the "code words" in F, using a maximum likelihood decoding rule (see Peterson [35]). Thus we have

$$q(F) = \{r \in F | (\forall r' \in F)(d(r', \hat{r}) \geqq d(r, \hat{r}))\}.$$

1.2.3. Query types to be considered. In this paper we shall only consider partial-match queries. The justification for this choice is that this query is quite common yet has not been "solved" in the sense of having known optimal search algorithms to answer it. In addition, partial- and best-match query types are usually considered as the paradigms of "associative" queries. The simpler intersection query types already have adequate algorithms for handling them. The more general situation where it is desired to handle any intersection query can be easily shown to require searching the entire file in almost all cases, if the file is encoded in a reasonably efficient manner. (Besides, it takes an average of $|\Sigma^k|$ bits to specify which intersection query one is interested in, so that it would generally take longer to specify the query than to read the entire file!) A practical retrieval system must therefore be based on a restricted set of query types or detailed knowledge of the query statistics.

1.3. Complexity measures. The difficulty of performing a particular task on a computer is usually measured in terms of the amount of time required. We shall measure the difficulty of performing an associative search by the amount of time it takes to perform that search.

Our measure is the "on-line" measure, that is, how much time it takes to answer a single query. This is the appropriate measure for interactive retrieval systems, where it is desired to minimize the user's waiting time. Many information retrieval systems can, of course, handle queries more efficiently in a batch manner—that is, they can accumulate a number of queries until it becomes efficient to make a pass through the entire file answering all the queries at once, perhaps after having sorted the queries. The practicability of designing a retrieval system to operate "on-line" thus depends on the relative efficiency with which a single query can be answered. That is the study of this paper.

When a file is stored on a secondary storage device such as a magnetic disk unit, the time taken to search for a particular set of items can be measured in terms of (i) the number of distinct access or read commands issued, and (ii) the amount of data transmitted from secondary storage to main storage. For most of our modeling, we shall consider only the number of accesses. Thus, for the generalizations of hash-coding schemes discussed in § 3, we count only the number of buckets accessed to answer the query.

Several measures are explicitly *not* considered here. The amount of storage space used to represent the file is not considered, except in § 3.3 to show that using extra storage space may reduce the time taken to answer the query. The time required to update a particular file structure is also not considered—this can always be kept quite small for the data structures examined.

1.4. Results to be presented. We give a brief exposition of the historical development of "associative" search algorithms in § 2.

In § 3 we study generalized hash functions as a means for answering partial-match queries. We derive a lower bound on their achievable performance, and precisely characterize the class of optimal hash functions. We then introduce a new class of combinatorial designs, called associative block designs. Interpreted as hash functions, associative block designs have excellent worst-case behavior while maintaining optimum average retrieval times. We also examine a method for utilizing storage redundancy (that is, we examine the efficiency gains obtainable by storing each record more than once).

In § 4 we study "tries" as a means for responding to partial match queries. "Tries" (plural of "trie") are a particular kind of tree in which the ith branching decision is made only according to the ith bit of the specific record being inserted or searched for, and not according to the results of comparisons between that record and another record in the tree. Their average performance turns out to be nearly the same as the optimal hash functions of § 3.

The results of § 3 and § 4 seem to support the following.

Conjecture. There is a positive constant c such that for all positive integers n, k and s, the average time required by any algorithm to answer a single partial match query $q \in Q_s$ must be at least

$$cn^{(k-s)/k},$$

where the average is taken over all queries $q \in Q_s$ and all files F of n k-bit records which are represented efficiently on a direct-access storage device. (That is, no more than $O(n \cdot k)$ bits of storage are used.)

2. Historical background.

2.1. Origins in hardware design. Associative search problems were first discussed by people interested in building associative memory *devices*. According to Slade [44], the first associative memory design was proposed by Dudley Buck in 1955. The hoped-for technological breakthrough allowing large associative memories to be built cheaply has not (yet) occurred, however. Small associative memories (on the order of 10 words) have found applications—most notably in "paging boxes" for virtual memory systems (see [12]). Minker [32] has written an excellent survey of the development of associative processors.

2.2. Exact-match algorithms. New search algorithms for use on a conventional computer with random-access memory were also being rapidly discovered in the 1950's. The first problem studied (since it is an extremely important practical problem) was the problem of searching for an exact match in a file of single-key records. Binary searching of an ordered file was first proposed by Mauchly [30]. The use of binary trees for searching was invented in the early 1950's according to [28], with published algorithms appearing around 1960 (see, for example, Windley [46]—there were also many others).

Tries were first described about the same time by Rene de la Briandais [11]. Tries are roughly as efficient as binary trees for exact-match searches. We shall examine trie algorithms for performing partial-match searches in § 4.

Hash-coding (invented by Luhn around 1953, according to Knuth [28]) seems the best solution for many applications. Given b storage locations (with $b \geqq |F|$) in which to store the records of the file, a *hash function* $h : \Sigma^k \to \{1, 2, \cdots, b\}$ is used to compute the address $h(r)$ of the storage location at which to store each record r. The function h is chosen to be a suitably "random" function of the input—the goal is to have each record of the file assigned to a distinct storage location. Unfortunately, this is nearly impossible to achieve (consider generalizations of the "birthday phenomenon" as in Knuth [28, § 6.4]), so a method must be used to handle "collisions" (two records hashing to the same address). Perhaps the simplest solution (*separate chaining*) maintains b distinct *lists*, or *buckets*. A record r is stored in list L_j (where $1 \leqq j \leqq b$) iff $h(r) = j$. Each bucket can now store an arbitrary number of records, so collisions are no longer a problem. To determine if an arbitrary record $r \in \Sigma^k$ is in the file, one need merely examine the contents of list $L_{h(r)}$ to see if it is present there. Since the expected size of each list is small, very little work need be done. Chaining can be implemented easily with simple linear linked list techniques (see [28, § 6.4]).

2.3. Single-key search algorithms. The next problem to be considered was that of single-key retrieval for records having more than one key (that is, $k > 1$). This is often called the problem of "retrieval on secondary keys". L. R. Johnson [26] proposed the use of k distinct hash functions h_i and k sets of lists L_j^i—for $1 \leqq i \leqq k$ and $1 \leqq j \leqq b$. Record r is stored in k lists—list $L^i_{h_i(r_i)}$ for $1 \leqq i \leqq k$. This is an efficient solution, although storage and updating time will grow with k. Prywes

and Gray suggested a similar solution—called Multilist—in which each attribute-value is associated with a unique list through the use of indices (search trees) instead of hash functions (see [20], [36]). Davis and Lin [10] describe another variant in which list techniques are replaced by compact storage of the record addresses relevant to each bucket. The above methods are often called *inverted list* techniques since a separate list is maintained of all the records having a particular attribute value, thus mapping attribute to records rather than the reverse as in an ordinary file.

2.4. Partial-match search algorithms. Inverted list techniques, while adequate for single-key retrieval of multiple-key records, do not work well for partial-match queries unless the number s of keys specified in the query is small. This is because the response to a query is the intersection of s lists of the inverted list system. The amount of work required to perform this intersection *grows* with the number of keys given, while the expected number of records satisfying the query decreases! One would expect a "reasonable" algorithm to do an amount of work that decreases if the expected size $E(|q(F)|)$ of the response decreases. One might even hope to do work proportional to the number of records in the answer. Unfortunately, no such "linear" algorithms have been discovered that do not use exorbitant amounts of storage. The algorithms presented in §§ 3 and 4, while nonlinear, easily outperform inverted list techniques. These algorithms do an amount of work that decreases approximately exponentially with s. When $s = 0$, *the whole file must, of course, be searched, and when* $s = k$, unit work must be done. In between, log(work) decreases approximately linearly with s.

J. A. Feldman's and P. D. Rovner's system LEAP [13] allows complete generality in specifying a partial match query. LEAP handles only 3-key records, however, so that there are at most eight query types. This is not as restrictive as might seem at first, since any kind of data can in fact be expressed as a collection of "triples": (attributename, objectname, value). While arbitrary Boolean queries are easily programmed, the theoretical retrieval efficiency is equivalent to an inverted list system.

Several authors have published algorithms for the partial match problem different from the inverted list technique. One approach is to use a very large number of short lists so that each query's response will be the union of some of the lists, instead of an intersection. Wong and Chiang [47] discuss this approach in detail. Note, however, that the requisite number of lists is at least $|\Sigma^k|$ if the system must handle all partial-match queries (since exact-match queries are a subset of the partial-match queries). Having such a large number of lists (most of them empty if $|F| < |\Sigma^k|$, as is usual) is not practical. A large number of authors (C. T. Abraham, S. P. Ghosh, D. K. Ray-Chaudhuri, G. G. Koch, David K. Chow and R. C. Bose—see references for titles and dates) have therefore considered the case where s is not allowed to exceed some fixed small value s' (for example, $s' = 3$). Then the number of lists required is $\binom{k}{s'} v^{s'}$. This is achieved by reserving a bucket for the response to each query with the maximal number s' of keys given; the response to other queries is then the union of existing buckets. Note that each record is now stored in $\binom{k}{s'}$ buckets, however! One can reduce the number of

buckets used and record redundancy somewhat, by the clever use of combinatorial designs, but another approach is really needed to escape combinatorial explosion.

The first efficient solution to an associative retrieval problem is described by Richard A. Gustafson in his Ph.D. thesis [21], [22]. Gustafson assumes that each record r is an *unordered* list $\{r_1, r_2, \cdots\}$ of at most k' attribute values (these might be key words, where the records represent documents) chosen from a very large universe of possible attribute values. Let w be chosen so that $\binom{w}{k'}$ is a reasonable number of lists to have in the system, and let a hash function h map attribute values into the range $\{1, 2, \cdots, w\}$. Each list is associated with a unique w-bit word containing exactly k' ones, and each record r is stored on the list associated with the word with ones only in positions $h(r_1), h(r_2), \cdots, h(r_{k'})$. (If these are not all distinct positions, extra ones are added randomly until there are exactly k' ones.) A query specifying attributes $a_1, a_2, \cdots, a_t$ (with $t \leqq k'$) need only examine the $\binom{w-t}{k'-t}$ lists associated with words with ones in positions $h(a_1), h(a_2), \cdots, h(a_t)$. The amount of work thus decreases rapidly with t. Note that the query response is not merely the union of the relevant lists, since undesired records may also be stored on these lists. We are guaranteed, however, that all the desired records are on the examined lists. In essence, Gustafson reduces the number of record types by creating w attribute classes, a record being filed according to which attribute classes describe it. His method has the following desirable properties:

(a) each record is stored on only one list (so updating is simple), and

(b) the expected amount of work required to answer a query decreases approximately exponentially with the number of attributes specified. His definition of a record differs from ours, however, so that the queries allowable in his system are a proper subset of our partial match queries—those with no zeros specified. (Convert each of his records into a long bitstring with ones in exactly k' places—each bit position corresponding to a permissible key word in the system.)

Terry Welch, in his Ph.D. thesis [45], studies the achievable performance of file structures which include directories. His main result is that the size of the directory is the critical component of such systems. He briefly considers directoryless files, and derives a lower bound on the average time required to perform a partial match search with hash-coding methods that is much lower than the precise answer given in § 3. He also presents Elias's algorithm for handling best-match queries, which the author shows to be minimize the average number of lists examined in [40] and [41].

3. Hashing algorithms for partial-match queries. Given a set Σ^k of possible records, and a number b of lists (buckets) desired in a filing scheme, we wish to construct a hash function $h : \Sigma^k \to \{1, \cdots, b\}$ so that partial match queries can be answered efficiently (either on the average or in the worst case). A record r in our file F is stored in list L_j if and only if $h(r) = j$, with collisions handled by separate

chaining. We define block B_j of the partition of Σ^k induced by h to be

$$B_j = \{r \in \Sigma^k | h(r) = j\},$$

so that

$$L_j = B_j \cap F.$$

If $|B_j| = |\Sigma^k|/b$ for all j, then we say that h is a *balanced* hash function.

To answer a partial-match query $q \in Q$, we must examine the contents of the lists with indices in

$$h(q) = \{j | (B_j \cap q(\Sigma^k)) \neq \text{null}\},$$

or equivalently,

$$h(q) = \bigcup_{r \in q(\Sigma^k)} \{h(r)\}.$$

Here we make the natural extension of h onto the domain Q.

The basic retrieval algorithm can thus be expressed

$$q(F) = \bigcup_{i \in h(q)} (L_i \cap q(\Sigma^k)),$$

or equivalently in pseudo-ALGOL:

Procedure SEARCH $(q, h, \{L_1, \cdots, L_b\})$;
comment SEARCH prints the response to a partial-match query $q \in Q$, given that the file $F \subseteq \Sigma^k$ is stored on lists $\{L_1, \cdots, L_b\}$ according to hash function h. ;
begin integer i; **record** r;
for each $i \in h(q)$ **do**
for each $r \in L_i$ **do**
if $r \in q(\Sigma^k)$ **then** print (r)
end SEARCH;

The difficulty of computing the set $h(q)$ depends very heavily on the nature of the hash function h. It is conceivable that for some pseudo-random hash functions h, it is more time-consuming to determine whether $i \in h(q)$ than it is to read the entire list L_i from the secondary storage device. Such hash functions are, of course, useless, since one would always skip the computation of $h(q)$ and read the entire file to answer a query. We will restrict our attention to hash functions h for which the time required to compute the set $h(q)$ of indices of lists that need to be searched is negligible in comparison with the time required to read those lists.

We denote the average and worst-case number of lists examined by SEARCH to answer a partial-match query with s keys specified, given that the file

was stored using hash function h, by

$$A_s(h) = |Q_s|^{-1} \sum_{q \in Q_s} |h(q)| \quad \text{and} \quad W_s(h) = \max_{q \in Q_s} |h(q)|,$$

respectively.

We shall denote the average number of lists examined, taken over all partial-match queries in Q, by $A(h)$. We denote the minimum possible average number of lists examined for a query $q \in Q_s$ by $A_{\min}(k, w, s, v)$ where h is assumed to be a balanced hash function mapping $\Sigma^k = \{0, 1, \cdots, v-1\}^k$ onto $\{1, 2, \cdots, b\}$, where $b = 2^w$, w being a natural number. If v is omitted, $v = 2$ is to be assumed.

Note that the number of lists examined does not depend on the particular file F being searched, but only on the hash function h being used.

3.1. Hash functions minimizing average search time. We first concentrate on the average number of buckets examined by SEARCH to calculate the response to a partial-match query q. That is, we consider the functions $A(h)$ and $A_s(h)$ and determine for which hash functions h are $A(h)$ and $A_s(h)$ minimized.

In § 3.1.1 we consider nonbinary records, (that is, where $|\Sigma|$ is relatively large), and show that a simple string-homomorphism hash function is optimal.

In § 3.1.2 we concentrate on binary records, for which a straightforward (ε-free) homomorphism technique would require using an exorbitant number of buckets (essentially one list per record in Σ^k). A relatively complicated combinatorial argument shows that selecting a subset of the record bits for use as address bits is optimal.

3.1.1. Optimal hash functions for nonbinary records. When k (the number of letters in each record) is less than or equal to w (the number of bits needed to describe a list index), then a very simple string-homomorphism scheme will provide efficient retrieval. This case arises frequently for nonbinary records.

We illustrate the principle by means of an example. Suppose we wish to construct a "crossword puzzle" dictionary for six-letter English words. Let $b = 2^w$ be the number of lists used. Given a word (for example, "SEARCH"), we construct a w-bit list index (bucket address) by forming the concatenation:

$$h(\text{"SEARCH"}) = g(\text{"S"})\, g(\text{"E"})\, g(\text{"A"})\, g(\text{"R"})\, g(\text{"C"})\, g(\text{"H"})$$

of six $(w/6)$-bit values; here g is an auxiliary hash function mapping the alphabet Σ into $(w/6)$-bit values.

For a partial-match query with s letters specified, we have, in this case,

$$A_s(h) = W_s(h) = 2^{w - s(w/6)}.$$

This approach is clearly feasible as long as $b \geqq 2^k$, since one or more bits of the list index can be associated with each attribute.

A similar technique has been proposed by M. Arisawa [2], in which the ith key determines, via an auxiliary hash function, the residue class of the list index modulo the ith prime (see also [34]).

For nonbinary records, the analysis fortunately turns out to be simpler than for binary records. In this section we prove that schemes that use the above string-homomorphism idea are, in fact, optimal.

Let the universe of records be Σ^k, where $\Sigma = \{0, 1, \cdots, v-1\}$ and $v > 2$. Let $Q_{\min}(x, k)$ denote the minimum possible number of partial-match queries in Q which require examination of a list L whose corresponding block B has size x. The following inequality gives a lower bound for $Q_{\min}$.

$$Q_{\min}(x, k) \geqq \min \left(Q_{\min}(\max_i f_i, k-1) + \sum_{0 \leqq i < v} Q_{\min}(f_i, k-1)\right),$$

where the minimum is taken over all sets of nonnegative integers $f_0, \cdots, f_{v-1}$ such that $\sum_{0 \leqq i < v} f_i = x$. (Here f_i denotes the number of records in B that begin with an i.) The term $Q_{\min}(\max f_i, k-1)$ is a lower bound on the number of partial-match queries beginning with a "*", and $Q_{\min}(f_i, k-1)$ is a lower bound on the number of partial-match queries beginning with "i", that require examination of L.

We can perform the analysis by passing to the continuous case. Define the function $Q'_{\min}$ as follows.

$$Q'_{\min}(x, k) = \inf \left(Q'_{\min}(\sup_{0 \leqq z \leqq v} f(z), k-1) + \int_0^v Q'_{\min}(f(z), k-1)\, dz\right),$$

where the infimum is taken over all nonnegative functions $f(z)$ such that $\int_0^v f(z)\, dz = x$. The appropriate initial conditions in this case are $Q'_{\min}(x, 0) = 1$ for $0 \leqq x \leqq 1$ (counting the totally unspecified query), $Q'_{\min}(x, 0) = \infty$ for $x > 1$ (this forces the above definition of $Q'_{\min}$ to pick values for $f(z)$ so that all records are distinguished after all the positions are examined), and $Q'_{\min}(x, 0) = 0$ otherwise. The function $Q'_{\min}(x, k)$ is obviously a lower bound for $Q_{\min}(x, k)$, since the definition for $Q'_{\min}(x, k)$ reduces to the above lower bound for $Q_{\min}$ (with equality holding) if $f(z)$ is defined $f(z) = f_i$ for $i \leqq z < i+1$.

The solution of the above recurrence for $Q'_{\min}$ is

$$Q'_{\min}(x, k) = \begin{cases} 0 & \text{if } x \leqq 0, \\ (x^{1/k} + 1)^k & \text{if } 0 < x \leqq v^k, \\ \infty & \text{otherwise.} \end{cases}$$

The infimum is reached in the definition of $Q'_{\min}(x, k)$ when $f(z)$ is the following function:

$$f(z) = \begin{cases} x^{(k-1)/k} & \text{for } 0 \leqq z \leqq x^{1/k}, \\ 0 & \text{otherwise.} \end{cases}$$

When x is the kth power of some integer y, $1 \leqq y \leqq v$, the lower bound provided by $Q'_{\min}(x, k)$ is, in fact, achievable.

THEOREM 1. *If B is a subset of Σ^k containing $x = y^k$ records, for some integer y, $1 \leqq y \leqq v = |\Sigma|$, then the number $Q(B)$ of queries which require examination of the list L associated with B is minimized when*

$$B = \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_k,$$

where each $\Sigma_i \subseteq \Sigma$ is of size y.

Proof. $Q(B) = Q'_{\min}(x, k)$ in this case. □

Theorem 1 says that our crossword puzzle hashing scheme is optimal, as long as the auxiliary function g divides the alphabet Σ into four equal pieces. This is not possible for the English 26-letter alphabet (Greek puzzlists are in luck with their 24-letter alphabet, however); we conjecture that splitting Σ into four nearly-equal pieces would be optimal in this case.

3.1.2. The optimal hash functions for binary records. In this section we prove a conjecture due to Welch [45]: when $k > \log_2(b)$, an optimal hash function is one which merely extracts the first $\log_2(b)$ bits of each record for use as a list index. As we shall see later, this hash function is only one of many which minimize the average number of lists examined, some of which also do better at minimizing the worst-case behavior as well. Sacerdoti [43] has also suggested that this scheme may be practical for partial-match retrieval.

We shall only consider balanced hash functions in this section. This eliminates considering obviously "degenerate" hash functions mapping all the records into a single list (costing one list examination per search). Furthermore, if each record in Σ^k is equally likely to appear in the file (independent of other records), then the expected length of each list will be the same. A formal justification for the restriction to balanced hash functions will be given later on, after we examine more closely the average search time of optimal balanced hash functions.

The following theorem gives a precise characterization of which balanced hash functions h minimize $A(h)$, the average number of lists examined.

THEOREM 2. *Let $h : \{0, 1\}^k \to \{1, \cdots, b\}$ be a balanced hash function with $b = 2^w$ buckets for some integer w, $1 \leqq w \leqq k$. Then $A(h)$ is minimal if and only if each block B_i is a $q(\{0, 1\}^k)$ for some query $q \in Q_w$.*

The geometry of the sets $q(\{0, 1\}^k)$ thus is reflected in an appealing fashion in the optimal shape for the blocks B_i. The set of records in each optimal B_i can be described by a string in $\{0, 1, *\}^k$ containing exactly w bits and $k - w$ *'s.

The following corollary is immediate.

COROLLARY 1. *The hash function which extracts the first w bits of each record $r \in \{0, 1\}^k$ to use as a list index minimizes $A(h)$.*

The proof of Theorem 2 is unfortunately somewhat lengthy, although interesting in that we prove a little more than is claimed.

Let $B_i \subset \{0, 1\}^k$ be any block. Then by $Q(B_i)$, we denote $|\{q \in Q | (q(\{0, 1\}^k \cap B_i) \neq \varnothing\}|$, the number of queries $q \in Q$ which examine list L_i. Denote by $Q_{\min}(x, k)$ the minimal value of $Q(B_i)$ for any B_i of size x, $B_i \subset \{0, 1\}^k$. We now note that

$$\begin{aligned} A(h) &= |Q|^{-1} \cdot \sum_{q \in Q} |h(q)| \\ &= |\{(B_i, q) | (q(\{0, 1\}^k) \cap B_i) \neq \varnothing\}| \cdot |Q|^{-1} \\ &= \sum_{1 \leqq i \leqq b} Q(B_i) \cdot |Q|^{-1} \geqq b \cdot Q_{\min}(2^{k-w}, k) \cdot |Q|^{-1}. \end{aligned}$$

To finish the proof of the theorem, we need only show that $Q(B_i) = Q_{\min}(2^{k-w}, k)$ if and only if B_i is of the form $q(\{0, 1\}^k)$ for some $q \in Q_w$. The rest of the proof involves four parts: calculation of $Q(B_i)$ for B_i of the proper form, calculation of

$Q_{\min}(2^{k-w}, k)$, the demonstration of equality, and then the demonstration of the "only if" portion.

Calculation of $Q(B_i)$. For simplicity of notation, we use the symbols x, y, z to denote either positive integers or their k-bit binary representation. Occasionally we may wish to explicitly indicate that some number t of bits is to be used; we denote this string by $x : t$ (so that $9 : 5 = 01001$). The length of a string x in $\{0, 1\}^*$ we denote by $|x|$. Concatenation of strings is represented by concatenation of their symbols; $0x$ denotes a zero followed by the string x. A string of t ones we denote by 1^t.

If x is a string, we denote by $\underline{x}$ the set of those $x+1$ strings of length $|x|$ which denote integers not greater than x. For example, $\underline{011} = \{000, 001, 010, 011\}$. If $x = 2^{k-w} - 1$, then $\underline{x : k}$ describes the set $q(\{0, 1\}^k)$ for $q = 0^w *^{k-w}$, which is in Q_w. Furthermore, $Q(q(\{0, 1\}^k))$ does not depend on which $q \in Q_w$ is chosen; this is always $2^w 3^{k-w}$ (since each * in q can be replaced by 0, 1 or *, and each specified position optionally replaced by a *, to obtain a query counted in $Q(q(\{0, 1\}^k))$).

Thus $Q(B_i) = 2^w 3^{k-w}$ for $B_i = \underline{x : k}$ with $x = 2^{k-w} - 1$. To show this is optimal, it is necessary to calculate $Q(\underline{x})$ for arbitrary strings x.

LEMMA 1. (a) $Q(\text{nullstring}) = 1$; (b) $Q(\underline{0x}) = 2Q(\underline{x})$; (c) $Q(\underline{1x}) = 2Q(\underline{1^{|x|}}) + Q(\underline{x})$.

Proof. Take (a) to be true by definition. For (b), any query examining $\underline{x}$ can be preceded by either a 0 or a * to obtain a query examining $\underline{0x}$. Part (c) follows from the fact that $\underline{1x} = \underline{0(1^{|x|})} \cup \underline{1x}$; there are $2Q(\underline{1^{|x|}})$ queries starting with 0 or *, and $Q(\underline{x})$ starting with 1. □

The preceding lemma permits $Q(\underline{x})$ to be easily computed for arbitrary strings x; we list some particular values:

$x =$	null	0	1	00	01	10	11		
$Q(\underline{x}) =$	1	2	3	4	6	8	9		
$x =$	000	001	010	011	100	101	110	111	
$Q(\underline{x}) =$	8	12	16	18	22	24	26	27	

If x is the string $x_1 x_2 \cdots x_k$ and z_i denotes the number of zeros in $x_1 \cdots x_i$, then Lemma 1 implies that

$$Q(\underline{x}) = 2^{z_k} + \sum_{1 \le i \le k} x_i 3^{k-i} 2^{z_i + 1}.$$

LEMMA 2. $Q(\underline{x1}) = 3Q(\underline{x})$.

Proof. Directly from the above formula for $Q(\underline{x})$ or by noting that if q is counted in $Q(\underline{x})$, then $q0$, $q1$ and $q*$ will be counted in $Q(\underline{x1})$. □

Using $p(x)$ to denote 2^{z_k} (where z_k is the number of zeros in x), we have also the following.

LEMMA 3. $Q(\underline{x : k}) = Q(\underline{x - 1 : k}) + p(x : k)$.

Proof. The only queries counted in $Q(\underline{x})$ but not $Q(\underline{x-1})$ will be those obtained by replacing an arbitrary subset of the zeros in x by *'s □

Now that we know quite a bit about $Q(q(\{0,1\}^k))$ for $q \in Q_w$, we turn our attention to $Q_{\min}$.

A lower bound for $Q_{\min}(x, k)$. The following inequality holds for $Q_{\min}(x, k)$, the minimal number of queries $Q(B_i)$ for any $B_i \subset \{0,1\}^k$, $|B_i| = x$:

$$Q_{\min}(x, k) \geqq \min\,(2Q_{\min}(x_0, k-1) + Q_{\min}(x_1, k-1)),$$

where the minimum is taken over all pairs of nonnegative integers x_0, x_1 such that $x_0 + x_1 = x$, $x_0 \geqq x_1$, and $x_0 \leqq 2^{k-1}$. To show this, let B_i contain x_0 records starting with a 0 and x_1 with a 1. Then $Q(B_i)$ must count at least $2Q_{\min}(x_0, k-1)$ queries beginning with a zero or a *, and at least $Q_{\min}(x_1, k-1)$ which start with a 1. (Nothing is lost by assuming $x_0 \geqq x_1$.) For $k = 1$, we have $Q_{\min}(x, 1) = Q(\underline{x-1 : 1})$, by inspection.

Showing $Q(B_i) = Q_{\min}(2^{k-w}, k)$ *if* $B_i = \underline{2^{k-w}-1 : k}$. We will, in fact, prove the stronger statement that $Q(B_i) = Q_{\min}(x+1, k)$ if $B_i = \underline{x : k}$, by induction on k. Since $Q(\underline{x : k}) \geqq Q_{\min}(x+1, k)$ necessarily, with equality holding for $k = 1$, as we have seen, equality can be proved in general using our lower bound for $Q_{\min}(x, k)$ if we can show the following:

$$Q(\underline{x : k}) \leqq \min\,(2Q(\underline{y : k-1}) + Q(\underline{z : k-1})), \tag{1}$$

where the minimum is taken over all pairs of nonnegative integers y, z such that $y + z + 1 = x$, $y \geqq z$ and $y \leqq 2^{k-1}$. By induction, the right-hand side of (1) is a lower bound for $Q_{\min}(x+1, k)$. The case in (1) of $x = y$ corresponding to $x_1 = 0$ in our lower bound for $Q_{\min}(x, k)$ holds by Lemma 1 (b). To prove (1), we consider four cases, according to the last bits of y and z.

Case 1. $y : k-1 = y'1$, $z : k-1 = z'1$, *and* $x : k = x'1$. In this case, (1) is true by Lemma 2, since we know by induction that $Q(\underline{x'}) \leqq 2Q(\underline{y'}) + Q(\underline{z'})$.

Case 2. $y : k-1 = y'0$, $z : k-1 = z'1$, *and* $x : k = x'0$. If $p(x') \leqq 2p(y')$, then (1) is true since it is equivalent by Lemmas 2 and 3 to

$$3Q(\underline{x'-1}) + 2p(x') \leqq 6Q(\underline{y'-1}) + 4p(y') + 3Q(\underline{z'}),$$

but we know by induction that $Q(\underline{x'-1}) \leqq 2Q(\underline{y'-1}) + Q(\underline{z'})$. Otherwise if $p(x') > 2p(y')$, we use the fact that (1) says

$$3Q(\underline{x'}) - p(x') \leqq 6Q(\underline{y'}) - 2p(y') + 3Q(\underline{z'})$$

and we know that $Q(\underline{x'}) \leqq 2Q(\underline{y'}) + Q(\underline{z'})$ by induction.

Case 3. $y : k-1 = y'1$, $z : k-1 = z'0$, *and* $x : k = x'0$. Depending on the truth or falsity of $p(x') \leqq p(z')$, we use induction and the fact that (1) is implied by either

$$3Q(\underline{x'-1}) + 2p(x') \leqq 6Q(\underline{y'}) + 3Q(\underline{z'-1}) + 2p(z')$$

or

$$3Q(\underline{x'}) - p(x') \leqq 6Q(\underline{y'}) + 3Q(\underline{z'}) - p(z').$$

Case 4. $y : k-1 = y'0$, $z : k-1 = z'0$, *and* $x : k = x'1$. If $p(y') \leqq p(z')$, we use induction and the fact that (1) is implied by

$$3Q(\underline{x'}) \leqq 6Q(\underline{y'}) - 2p(y') + 3Q(\underline{z'-1}) + 2p(z').$$

Otherwise we use the fact that (1) is implied by

$$3Q(\underline{x'}) \leqq 6Q(\underline{y'-1}) + 4p(y') + 3Q(\underline{z'}) - p(z').$$

This completes the proof that $Q(\underline{x : k}) = Q_{\min}(x+1, k)$.

Showing $Q(B_i) = Q_{\min}(2^{k-w}, k)$ *only if* $B_i = q(\{0, 1\}^k)$ *for some* $q \in Q_w$. We need only that (1) holds with equality for $x = 2^{k-w} - 1$ only if $y = z$, since this implies that $Q(B_i) > Q_{\min}(2^{k-w}, k)$ if B_i is not of the form $q(\{0, 1\}^k)$ for $q \in Q_w$. To see this, let $B_i = 0C \cup 1D$ for $C, D \subset \{0, 1\}^{k-1}$. Then note that if $B_i \neq q(\{0, 1\}^k)$ for some $q \in Q_w$, then either $|C| > 0$, $|D| > 0$, and $|C| \neq |D|$; or $|C| = |D|$ but at least one of C, D is not of the form $q(\{0, 1\}^{k-1})$ for any $q \in Q_{w-1}$.

Now $x : k = 0^w 1^{k-w}$. If $y = z$, then (1) holds with equality by Lemma 1. To show (1) holds with equality only if $y = z$, suppose $y = 2^{k-w-1} + t - 1$ and $z = 2^{k-w-1} - t - 1$ for any t, $0 < t < 2^{k-w-1}$. Then (1) holding with strict inequality says:

$$Q(\underline{0^w 1^{k-w}}) < 2Q(\underline{y : k-1}) + Q(\underline{z : k-1})$$

or

$$Q(\underline{01^{k-w}}) < 2Q(\underline{y : k-w}) + Q(\underline{z : k-w})$$

or

$$Q(\underline{01^{k-w}}) < Q(\underline{y : k-w+1}) + Q(\underline{z : k-w}).$$

Subtracting 2^{k-w-1} from both x and y, we get that (1) means

$$Q(\underline{01^{k-w-1}}) < Q(\underline{t-1 : k-w}) + Q(\underline{2^{k-w-1} - t - 1 : k-w}).$$

It is simpler to note that the general statement

$$Q(\underline{x : k}) < Q(\underline{t-1 : k}) + Q(\underline{x-t : k})$$

is always true; in fact, it is implied directly by (1), Lemma 1 and the fact that $Q(\underline{x-t : k})$ is always positive. This completes our proof that $Q(B_i) = Q_{\min}(2^{k-w}, k)$ only if $B_i = q(\{0, 1\}^k)$ for some $q \in Q_w$, and also finishes our proof of Theorem 2. □

While Theorem 2 only counted queries in Q, the same result holds if we count queries in Q_s.

THEOREM 3. *Let* h *and* b *be as in Theorem* 1. *Then* $A_s(h)$ *is minimal for* $0 < s < k$ *if and only if each block* B_i *is a* $q(\{0, 1\}^k)$ *for some* $q \in Q_w$.

This can't be asserted in an "iff" manner for $s = 0$ or $s = k$, since $A_0(h) = b$ and $A_k(h) = 1$ independent of h.

Theorem 3 can be proved in the same manner as Theorem 2. We note here the differences, using $Q_s(B_i)$ and $Q_{\min,s}(x, k)$ to count queries in Q_s rather than Q.

LEMMA 4. (a) $Q_0(\underline{x})=1$, *for all* x. (b) Q_s (nullstring) $=0$ *for* $s\geqq 1$. (c) $Q_s(\underline{0x})=Q_s(\underline{x})+Q_{s-1}(\underline{x})$, *for* $s\geqq 1$ *and all* x. (d) $Q_s(\underline{1x})=Q_s(\underline{01^{|x|}})+Q_{s-1}(\underline{x})$, *for* $s\geqq 1$ *and all* x.

LEMMA 5. $Q_s(\underline{x})-Q_s(\underline{x-1})=\binom{z_k}{k-s}$, *where* z_k *denotes the number of zeros in* x *and* $|x|=k$.

LEMMA 6. $Q(\underline{x1})=2Q_{s-1}(\underline{x})+Q_s(\underline{x})$.

LEMMA 7. $Q_{\min,s}(x,k)\geqq\min(Q_{\min,s}(x_0,k-1)+Q_{\min,s}(x_0,k-1)+Q_{\min,s-1}(x_1,k-1))$ *where the minimum is taken over all pairs of nonnegative integers* x_0, x_1, *such that* $x_0+x_1=x$, $x_0\geqq x_1$ *and* $x_0\leqq 2^{k-1}$.

The proofs of these lemmas are omitted here (see [40]). The proof of Theorem 2 then proceeds along the same lines as that of Theorem 1; with (1) being replaced by

$$Q_s(\underline{x:k})\leqq\min(Q_s(\underline{y:k-1})+Q_{s-1}(\underline{y:k-1})+Q_{s-1}(\underline{z:k-1})).$$

We omit details here of the proof, as it varies little from the proof of (1). The "only if" portion of the proof is also similar, except that the inductive hypothesis has to be applied more than once (for varying s values). □

Calculation of $A_s(h)$. Now that Theorems 2 and 3 tell us what the optimal balanced hash functions $h:\{0,1\}^k\to\{1,\cdots,b=2^w\}$ are like, we can calculate $A_s(h)$ easily, using the optimal h from Corollary 1 to Theorem 2 (using the first w bits of $x\in\{0,1\}^k$ as the hash value). We get

$$A_{\min}(k,w,s)=A_s(h)=\binom{k}{s}^{-1}\sum_{0\leqq i\leqq s}\binom{w}{i}\binom{k-w}{s-i}2^{w-i}\geqq b^{1-s/k},$$

where the first sum considers the ways in which i of the s specified bits fall in the first w positions; when this happens 2^{w-i} buckets must be searched. The inequality is a special case of a mean value theorem [24, Thm. 59]. In fact, $b^{1-s/k}$ is a reasonable approximation to $A_s(h)$, as long as w is not too small.

A better approximation may be obtained by replacing the hypergeometric probability

$$\binom{k}{s}^{-1}\binom{w}{i}\binom{k-w}{s-i}$$

by its approximation

$$\binom{w}{i}(s/k)^i(1-s/k)^{w-i}.$$

Then we have

$$A_{\min}(k,w,s)\approx\sum_i\binom{w}{i}(s/k)^i(1-s/k)^{w-i}2^{w-i}$$

$$=\sum_i\binom{w}{i}(s/k)^i(2-2\cdot s/k)^{w-i}=(2-s/k)^w.$$

For example, when half of the bits are specified in a query ($s=k/2$), we should expect to look at $(1.5)^w$ lists.

3.1.3. The optimal number of lists. In this section we use the results of the last section to derive the optimal number of lists (buckets) in a hash-coding partial-match retrieval scheme. We also give justification for the use of balanced hash functions.

The basic result of the preceding section was that the optimal shape for a block B is a "subcube" of $\{0, 1\}^k$; that is, $B = q(\{0, 1\}^k)$ for some partial-match query q, under the assumption that each partial-match query is equally likely. Let us now assume that each record $r \in \Sigma^k$ is equally likely to appear in F (equivalently, every file F of a given size is equally likely). In this case, the expected length of a list L_i is just $|F| \cdot |B_i| \cdot 2^{-k}$.

We use the following simple model for the true retrieval cost (rather than just counting the number of lists searched): α seconds are required to access each list L_i, and ρ seconds are required to read each record in L_i. The total time to search L_i is then $\alpha + \rho|L_i|$. Suppose $B_i = q(\{0, 1\}^k)$ for some $q \in Q_w$. The average expected time per partial-match query spent searching list L_i is then just

$$E((\alpha + \rho|L_i|) \cdot (2^w 3^{k-w}) \cdot 3^{-k}) = (\alpha + \rho|F| \cdot |B_i| \cdot 2^{-k}) \cdot (2/3)^w,$$

since there are $2^w 3^{k-w}$ queries which must examine L_i out of 3^k possible queries, and $(E|L_i|) = |F| \cdot |B_i| \cdot 2^{-k}$. To minimize the total average cost (the above summed over lists L_i), it is sufficient to minimize the cost per element of $\{0, 1\}^k$. Dividing the above by $|B_i| = 2^{k-w}$, we want to minimize

$$(2/3)^w 2^{-k}[\alpha \cdot 2^w + \rho|f|].$$

Taking the derivative with respect to w, setting the result to zero and solving for w, we get that the total expected cost per query is minimized when

$$w = \log_2 \left(\frac{-\rho|F| \ln (2/3)}{\alpha \ln (4/3)} \right) \cong \log_2 (\alpha^{-1} \rho|F| \cdot 1.408).$$

(The second derivative is positive, so this is a minimum.)

For example, if $\rho = .001$, $\alpha = .05$, $|F| = 10^6$ and $k = 40$, then the optimal hashing scheme would have $w \approx 15$, so that $2^{15} = 32{,}768$ buckets would be used. The average time per query turns out to be about 6.5 seconds.

Note that the optimal choice of w does not depend on k, due to the fact that the probability $(2/3)^w$ of examining L_i when $|B_i| = 2^{k-w}$ does not depend on k. Since our above analysis only considered a single arbitrary bucket, all buckets should have the same optimal size determined above. That is, the hash function should be balanced.

3.2. Minimizing the worst-case number of lists searched. The worst-case behavior of the hash function of Corollary 1 is obviously poor; if none of the specified bits occur in the first w positions, then every list must be searched. In this section we find that other optimal average-time hash functions exist which have much improved worst-case behavior, often approximately equal to the average behavior. We also consider a simpler strategy involving storing each record in several lists.

To obtain good worst-case behavior $W_s(h)$ for $h : \{0, 1\}^k \to \{1, \cdots, b = 2^w\}$, the hash function $h(x)$ must depend on all of the bits of x, so that each specified bit contributes approximately equally and independently towards decreasing the

number of lists searched. We shall also restrict our attention to balanced hash functions satisfying the conditions of Theorems 2 and 3 so that optimal average time behavior is ensured. While we have no proof that these block shapes are necessary for optimal worst-case behavior, the fact that $W_s(h)$ is bounded below by $A_s(h)$ makes it desirable to keep $A_s(h)$ minimal.

3.2.1. An example. Let us consider, by way of introduction, an example with $k=4$, $w=3$. The following table describes an interesting hash function h; row i describes the query $q \in Q_3$ such that $B_i = q(\{0,1\}^4)$. Thus $h(0110)=6$ and $h(1110)=4$, each $x \in \{0,1\}^4$ is stored in a unique bucket. (This function was first pointed out to the author by Donald E. Knuth.)

TABLE 1
A hash function $h : \{0,1\}^4 \to \{1, \cdots, 8\}$

		Bit position			
		1	2	3	4
	1	0	0	*	0
	2	1	0	0	*
Bucket	3	*	1	0	0
address	4	1	*	1	0
	5	1	1	*	1
	6	0	1	1	*
	7	*	0	1	1
	8	0	*	0	1

This can be interpreted as a perfect matching on the Boolean 4-cube, as indicated in Fig. 1, since each block contains two adjacent points and distinct blocks are disjoint. In general, we have the problem of packing $\{0,1\}^k$ with disjoint $(k-w)$-dimensional subspaces.

Whereas the hash function of Corollary 1 would have all its *'s in the fourth column, here the *'s are divided equally among the four columns. As a result, we have $W_1(h)=5$ instead of 8. For example, we need only examine buckets 1, 4, 5, 6 and 7 for the query **1*. Table 2 lists $W_s(h)$ and $\lceil A_s(h) \rceil$ for $0 \leqq s \leqq 4$; we see that we have reduced $W_s(h)$ so that $W_s(h) = \lceil A_s(h) \rceil$; no further reduction is possible.

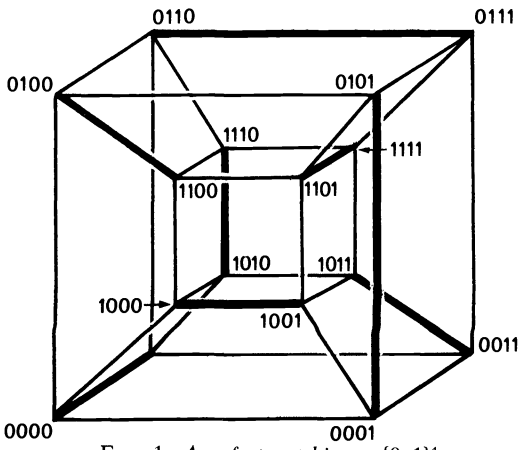


FIG. 1. *A perfect matching on* $\{0,1\}^4$

TABLE 2

$W_s(h)$ and $A_s(h)$

s	0	1	2	3	4
$W_s(h)$	8	5	3	2	1
$\lceil A_s(h) \rceil$	8	5	3	2	1

3.2.2. Definition of ABD's. Let us call a table such as Table 1 an "associative block design of type (k, w)", or an ABD(k, w) for short. To be precise, *an* ABD(k, w) *is a table with* $b = 2^w$ *rows and* k *columns with entries from* $\{0, 1, *\}$ *such that* $k > w$ *and*

(i) each row has w digits and $k - w$ *'s,

(ii) the rows represent disjoint subsets of $\{0, 1\}^k$. That is, given any two rows, there exists a column in which they contain differing digits.

(iii) each column contains the same number $b \cdot (k-w)/k$ of *'s.

Conditions (i) and (ii) ensure that $A_s(h)$ is minimal, by Theorem 3, since each row of the table represents a partial-match query in Q_w by condition (i), and by (ii) the corresponding sets $q(\{0, 1\}^k)$ are disjoint.

Condition (iii) attempts to restrict the class of ABD's to those hash functions with good worst-case behavior $W_s(h)$ by requiring a certain amount of uniformity in the utilization of each record bit by h. In fact, (iii) implies that $W_1(h)$ is minimal by (i) of Theorem 4, to follow. More stringent uniformity conditions are conceivable, perhaps involving the distribution of t-tuples within each t-subset of columns, but (iii) alone is enough to make the construction of ABD's a difficult combinatorial problem, comparable to the construction of balanced block designs (see [9]). Furthermore, with the exception of a construction due to Preparata based on BCH codes for the case $w = k - 1$, the existence of ABD's of arbitrary size does not seem to be answered by any previous results of combinatorial design theory. (See [40], [41], however, for an interpretation of ABD's as a special case of group-divisible incomplete block designs, defined in [5].)

3.2.3. Characteristics of ABD's. The following lemma gives some additional characteristics of ABD's that are implied by the definition.

THEOREM 4. *An* ABD(k, w)

(i) *has exactly* $b \cdot w/2k$ 0's (*or* 1's) *in each column,*

(ii) *has* $\binom{w}{u}$ *rows which agree in exactly* u *positions with any given record* $r \in \{0, 1\}^k$, *for any* u, $0 \leqq u \leqq w$,

(iii) *is such that*

$$k \cdot \left(\frac{b \cdot w}{2k}\right)^2 \geqq \binom{b}{2},$$

or equivalently,

$$\frac{k}{w} \geqq \frac{w}{2} \cdot \left(\frac{b}{b-1}\right).$$

Proof. (i) There are as many records in $\{0, 1\}^k$ having a 0 in a given column as there are having a 1. A row of the ABD containing a * in that column represents a block B containing an equal number of each type. Rows containing a digit in that column represent blocks containing 2^{k-w} records of that type.

(ii) Let a_u denote the number of rows in the ABD which agree in exactly u positions with the given record r. We have that a_u is nonzero only if $0 \leqq u \leqq w$, since any row of the ABD contains w digits (*'s don't "agree" with digits). Also $a_0 = 1$, since the complement of r is stored in a unique list. Furthermore,

$$a_u = \binom{k}{u} - \sum_{0 \leqq v \leqq u} a_v \binom{k-w}{u-v},$$

since of the $\binom{k}{u}$ records that agree with r in u positions,

$$\sum_{0 \leqq v < u} a_v \binom{k-w}{u-v}$$

are accounted for by rows which agree with r in v positions, for some $v < u$ (by replacing $u-v$ of the $k-w$ stars in that row by digits which agree with r, and the other $k-w-u+v$ stars by digits complementary to r). Each record remaining must be in a separate list, since a row agreeing with r in u places can represent at most one record agreeing with r in u places, and any row agreeing with r in more than u places can represent none. The formula $a_u = \binom{w}{u}$ is the solution to the above recurrence.

(iii) Between each pair of rows in the ABD there must be at least one column in which they contain differing digits. There must be at least $\binom{b}{2}$ such row-row-column differences. On the other hand, each column can contribute at most $(bw/2k)^2$ such differences by (i) of this theorem. □

Parts (i) and (iii) of the above theorem can be used to restrict the search for ABD's. Note that (i) implies that $bw/2k$ must be integral, so that no ABD(5, 4)'s exist, for example. Part (iii) implies that to achieve large (record length)/(list index length) ratios k/w, we must let w grow to at least $2k/w$, approximately. For $k \leqq 20$ the above restrictions imply that the ABD(k, w)'s could only exist for the following (k, w) pairs:

(4, 3),	(8, w) for $4 \leqq w \leqq 7$,
(10, 5),	(12, 6), (12, 9)
(14, 7)	(16, w) for $6 \leqq w \leqq 15$,
(18, $3t$) for $2 \leqq t \leqq 5$,	(20, 10), (20, 15).

By an extension of the reasoning used to show (iii), an ABD(8, 4) has been shown not to exist, so that an ABD(8, 5) would be the next possible size after our ABD(8, 4) of § 3.2.1 for which existence is possible. To date, the existence of an ABD(8, 5) has not been settled; ABD(8, 6)'s and ABD(8, 7)'s are shown to exist in the following section.

3.2.4. ABD construction techniques. We present here several direct construction techniques which provide infinite classes of ABD's. The general question of the existence of an ABD of arbitrary type (k, w) seems to be extremely difficult; the positive nature of the very partial results obtained here suggests, however, that ABD's are not scarce.

We first present a simple infinite class of ABD's. The construction here is due to Ronald Graham. Franco Preparata has discovered another construction for a class of the same parameters, based on cyclic BCH erro-correcting codes [38].

THEOREM 5. *An* ABD$(2^t, 2^t - 1)$ *exists for* $t \geqq 2$.

Proof. We extend our notation for an ABD; a row containing r "–" 's will represent 2^r rows of the actual ABD obtained by independently replacing each – with a 0 or 1. The construction has two parts:

(i) Rows 1 to $t+1$ have –'s in positions $t+2$ to k. Row i for $1 \leqq i \leqq t+1$ has its star in column i, the remaining columns contain digits. (For example, columns 1 to $t+1$ of these rows might contain cyclic shifts of $*10^{t-1}$.)

(ii) Row i for $t+2 \leqq i \leqq 2k-t-1$ contains digits in columns 1 to $t+1$, a * in column $t+1+\lfloor (i-t)/2 \rfloor$, and –'s elsewhere. The digits used are arbitrary except they must satisfy part (ii) of the ABD definition. It is easy to check that this is an ABD. □

We present in Table 3 an ABD(8, 7)$(t=3)$ constructed this way.

TABLE 3
An ABD(8, 7)

	1	2	3	4	5	6	7	8
1	*	1	0	0	–	–	–	–
2	0	*	1	0	–	–	–	–
3	0	0	*	1	–	–	–	–
4	1	0	0	*	–	–	–	–
5	0	0	0	0	*	–	–	–
6	0	1	0	1	*	–	–	–
7	0	1	1	1	–	*	–	–
8	1	0	1	0	–	*	–	–
9	1	0	1	1	–	–	*	–
10	1	1	0	1	–	–	*	–
11	1	1	1	0	–	–	–	*
12	1	1	1	1	–	–	–	*

Graham has also constructed an ABD(16, 13) with a similar two-part method. This design is given in Table 4.

TABLE 4
An ABD(16, 13)

```
*0001**---------
*0101--**-------
*0111----**-----
1*000------**---
1*010--------**-
1*011*---------*
01*00-**--------
01*01---**------
11*01-----**----
001*0-------**--
101*0---------**
111*0**---------
0001*--**-------
0101*----**-----
0111*------**---
00000--------***
11111--------***
```

The ABD's of Theorem 5, while interesting as a solution to a combinatorial problem, are essentially useless as hash functions since the number of buckets is unacceptably large. We wish to have ABD's such that the ratio k/w does not tend to 1. The following theorem does just that.

THEOREM 6. *Given an* ABD(k, w) *and an* ABD(k', w') *such that* $k/w = k'/w'$, *one can construct an* ABD$(k+k', w+w')$.

Proof. For each possible pair of rows (R_1, R_2) with $R_1 \in$ ABD(k, w), $R_2 \in$ ABD(k', w'), let the concatenation R_1R_2 be a row of the ABD$(k+k', w+w')$. This is easily shown to be a legal ABD. □

We can now form an ABD(8, 6) or an ABD(12, 9) from the design of Table 1. Table 5 gives the ABD(8, 6) so constructed.

TABLE 5
An ABD(8, 6)

	12345678		12345678		12345678		12345678
1	00*000*0	17	*10000*0	33	11*100*0	49	*01100*0
2	00*0100*	18	*100100*	34	11*1100*	50	*011100*
3	00*0*100	19	*100*100	35	11*1*100	51	*011*100
4	00*01*10	20	*1001*10	36	11*11*10	52	*0111*10
5	00*011*1	21	*10011*1	37	11*111*1	53	*01111*1
6	00*0011*	22	*100011*	38	11*1011*	54	*011011*
7	00*0*011	23	*100*011	39	11*1*011	55	*011*011
8	00*00*01	24	*1000*01	40	11*10*01	56	*0110*01
9	100*00*0	25	1*1000*0	41	011*00*0	57	0*0100*0
10	100*100*	26	1*10100*	42	011*100*	58	0*01100*
11	100**100	27	1*10*100	43	011**100	59	0*01*100
12	100*1*10	28	1*101*10	44	011*1*10	60	0*011*10
13	100*11*1	29	1*1011*1	45	011*11*1	61	0*0111*1
14	100*011*	30	1*10011*	46	011*011*	62	0*01011*
15	100**011	31	1*10*011	47	011**011	63	0*01*011
16	100*0*01	32	1*100*01	48	011*0*01	64	0*010*01

Theorem 5 allows us to construct an infinite family of ABD's of type $(4t, 3t)$, for $t \geqq 1$, using the ABD of Table 1. This is approaching utility (an ABD(16, 12) is conceivably useful, say) but we need "starting" designs with large k/w to obtain a family with large k/w. On the other hand, we know by Theorem 4 (iii) that designs with large k/w must have k at least $2(k/w)^2$ approximately. Unfortunately, these tables get rapidly unmanageable by hand. Computer searches for an ABD(8, 5) or an ABD(10, 5) also ran out of time before finding any. The question as to whether ABD(k, w)'s existed with $k/w > 4/3$ thus remained open until the following theorem, showing that k/w can be arbitrarily large, was discovered.

THEOREM 7. *Given an* ABD(k, w) *and an* ABD(k', w') *one can construct an* ABD(kk', ww').

Proof. Each row R of the first ABD generates $2^{w(w'-1)}$ rows of the resultant ABD, as follows. The set of rows of the ABD(k', w') is arbitrarily divided into equal-sized subsets, A_0 and A_1. Each character x of R is replaced by a string of k' characters: if $x = *$, x is replaced by $*^{k'}$, otherwise x is replaced by some row in A_x. The w digits of R are replaced independently in all possible ways by rows from the corresponding sets A_0 and A_1.

This generates a table with $2^{ww'}$ rows of length kk', each row having $(k-w)k' + w(k'-w') = kk' - ww'$ *'s. Any two rows of the resultant ABD must contain differing digits in at least one column, since rows replacing differing digits must differ, or if the two rows were generated from the same row of the first design, then one of the digits replaced will have been replaced by two (differing) rows from the second ABD. The number of *'s in each column turns out to be $2^{ww'}(kk' - ww')/kk'$, as required, so that we have created an ABD(kk', ww'). □

The theorem allows us to form ABD's with arbitrarily large ratios k/w. For example, we can now construct an ABD(16, 9) or an ABD(64, 27) (in general, an ABD(4^t, 3^t) for $t \geqq 1$) from the ABD(4, 3) of Table 1. The following table illustrates the rows generated for an ABD(16, 9).

TABLE 6
Rows of an ABD(16, 9)

00*0 →	00*000*0****00*0
	00*000*0****100*
	00*000*0*****100
	00*000*0****1*10
	00*0100*****00*0
	...
100* →	11*100*000*0****
	11*100*0*100****
	11*100*01*10****
	...
⋮	
0*01 →	00*0****00*011*1
	00*0****00*0011*
	00*0****00*0*011
	...

ABD(4, 3) rows → ABD(16, 9) rows

The preceding theorem allows us to construct an ABD(4^t, 3^t) for $t \geq 1$, for example, from our ABD(4, 3). While this does provide large k/w designs, they are quite likely not the smallest designs for the given ratio, since by Theorem 4 (iii) we might hope to have w grow linearly with (k/w), whereas here it grows like $(k/w)^{\log_{4/3}(3)} > (k/w)^4$. At present, however, it is our only way of constructing large k/w designs.

3.2.5. Analysis of ABD search times. Let us examine the worst-case number of lists examined $W_s(h)$ for hash functions associated with the ABD's of the last section.

Consider first the hash function h associated with an ABD($k+k'$, $w+w'$) which was created by the concatenation (Theorem 6) of an ABD(k, w) and an ABD(k', w') (with associated hash functions g and g', respectively). Then

$$W_s(h) = \max_{u+v=s} W_u(g) \cdot W_v(g')$$

for $0 \leq s \leq k+k'$, $0 \leq u \leq k$, $0 \leq v \leq k'$. For example, the ABD(8, 6) created from two of our ABD(4, 3)'s has $W_s(h)$ as in Table 7.

TABLE 7
Performance of an ABD(8, 6)

s	0	1	2	3	4	5	6	7	8
$W_s(h)$	64	40	25	16	10	6	4	2	1
$\lceil A_s(h) \rceil$	64	40	25	15	9	6	4	2	1

Also shown are the values of $\lceil A_s(h) \rceil$, which is a lower bound for $W_s(h)$. The ABD(8, 6) is seen to do nearly as well as possible. The exact asymptotics for the worst-case behavior of the repeated concatenation of an ABD with itself are quite simple to figure out for given values of k and w. Suppose we concatenate an ABD(k, w) with worst-case behavior $W_s(g)$ with itself m times, yielding an ABD(mk, mw). Consider a partial match query $q \in Q_s$. Let y_i be the number of k-column blocks which have exactly i specified bits, for $0 \leq i \leq k$, so that

$$\sum_{0 \leq i \leq k} y_i = m \quad \text{and} \quad \sum_{0 \leq i \leq k} i y_i = s.$$

We also have, of course, the condition that

$$y_i \geq 0 \quad \text{for} \quad 0 \leq i \leq k.$$

The worst-case behavior $W_s(h)$ of the resultant ABD(mk, mw) is defined by

$$W_s(h) = \max \prod_{0 \leq i \leq k} W_i(g)^{y_i},$$

where the maximum is taken over all sets of integers $y_o, \cdots, y_k$ satisfying the three conditions on the y_i's given above.

Let $W'_s(h) = \log W_s(h)$ for any h and for all s, so that

$$W'_s(h) = \max \sum_{0 \leq i \leq k} W'_i(g) \cdot y_i,$$

yielding an integer programming problem in $(k+1)$-dimensional space. Since we desire the asymptotic behavior as $m \to \infty$, the solution to the corresponding linear programming problem, in which each y_i is replaced by the corresponding fraction $x_i = y_i/m$, will give us the asymptotic behavior. The problem to be solved is thus:

$$\text{maximize } W'_s(h) = \sum_{0 \leqq i \leqq k} W'_i(g) x_i,$$

subject to

$$\sum_{0 \leqq i \leqq k} x_i = 1,$$

$$\sum_{0 \leqq i \leqq k} i x_i = s/m,$$

$$x_i \geqq 0 \quad \text{for} \quad 0 \leqq i \leqq k.$$

We must have at least $k-1$ of the x_i's equal to zero in the optimal solution, since there are only $k+3$ constraints for this problem in $k+1$ dimensions. Let x_i and x_j be the two nonzero values, with $i < j$. If $W'_s(g)$ is a concave function, we have

$$i = \lfloor s/m \rfloor = j-1$$

and

$$W'_s(h) = W'_i(g)(j - s/m)/(j-i) + W'_j(g)(s/m - i)/(j-i).$$

This is the general solution. When s/m is a multiple of $1/k$, then only $x_{sk/m}$ is nonzero, equal to one. This solution does not apply when $W'_s(g)$ is not concave. (For example, $W'_s(g)$ for our ABD(4, 3) is not quite concave, since $W_3(g) = 2$ is a little too large. This convexity is the cause of the discrepancies of Table 4.) One can show, by a combinatorial argument, that if $W_s(g) = A(k, w, s)$ for $0 \leqq s \leqq k$, then $W_s(h) = A(mk, mw, s)$ for $0 \leqq s \leqq mk$ as well. Thus concatenation of ABD's can be expected to preserve near-optimal worst-case behavior.

The behavior of an ABD constructed by insertion is more difficult to work out. It seems the worst case here occurs when the specified bits occur together in blocks corresponding to the digits of the first ABD (of type (k, w)) used in the construction.

Under this assumption (for which I have no proof) the worst-case behavior of an ABD(16, 9) created from two of our ABD(4, 3)'s can be calculated to be as given in Table 8.

TABLE 8
Performance of an ABD(16, 9)

s	0	1	2	3	4	5	6	7	8
$W_s(h)$	512	368	272	224	176	116	76	56	36
$\lceil A_s(h) \rceil$	512	368	263	186	131	91	63	43	30

s	9	10	11	12	13	14	15	16
$W_s(h)$	33	24	16	8	5	3	2	1
$\lceil A_s(h) \rceil$	20	14	9	6	4	3	2	1

We see that while $W_s(h)$ approximates $A_s(h)$ reasonably well, performance has been degraded somewhat. We conjecture that better ABD(16, 9)'s exist. (It is a situation reminiscent of the fact that recursive or systematic constructions of error-correcting codes tend not to work as well as a random code.) An exhaustive search by computer for better designs appears to be infeasible, so that a better construction method is needed. (A sophisticated backtracking procedure was unable to determine whether an ABD(8, 5) exists or not, using one hour of computer time.)

While it is not too difficult to calculate $W_s(h)$ for small ABD's constructed by insertion (assuming that our conjecture about the nature of the worst case is correct), the asymptotic analysis seems difficult. In any case, the ABD's so constructed yield large k/w designs with significantly improved $W_s(h)$.

3.2.6. Irregular ABD's. The difficulty of constructing ABD's leads one to attempt simpler, less tightly constrained hash functions. Such ad hoc hash functions are easy to construct for small values of k and w. For example, consider the case $k = 3$, $w = 2$ (which does not satisfy the divisibility constraint of Theorem 4, so that an ABD(3, 2) can not exist). The "design" in Table 9 yields reasonably good worst-case performance.

TABLE 9
An "irregular" (3, 2) design

	1	2	3
1	0	0	*
2	1	*	0
3	*	1	1
4	0	1	0
	1	0	1

Here bucket 4 contains both records 010 and 101. This hash function has worst-case behavior as in Table 10.

TABLE 10
Behavior of the previous design

s	0	1	2	3
$W_s(h)$	4	3	2	1

Concatenating this function with itself will yield larger "designs" having a k/w ratio of 3/2 and having good worst-case retrieval times. Another "design" yields a k/w ratio of 2 (Table 11).

TABLE 11
An "irregular" (4, 2) design

	1	2	3	4
1	0	0	*	*
2	*	1	*	0
3	*	1	1	1
	1	0	1	*
4	*	1	0	1
	1	0	0	*

The above hash function has worst-case behavior shown in Table 12.

TABLE 12

Behavior of the irregular (4, 2) *design*

s	0	1	2	3	4
$W_s(h)$	4	4	3	2	1

3.2.7. Conclusions on ABD's. Associative designs minimize the average retrieval time for partial-match queries, since the blocks have the optimal shape specified in Theorems 2 and 3. In addition, they reduce greatly the worst-case retrieval time, often nearly to the average time. While the recursive or iterative nature of our ABD construction techniques lends itself to simple implementation, the complexity of the hash function computation may be such that using ABD's is justified only when the file is stored on slow secondary storage devices.

In summary, ABD's can be used to minimize the worst-case performance of hashing algorithms with no increase in either the average retrieval time of the amount of storage used.

3.3. Benefits of storage redundancy. The perhaps difficult problems involved in constructing an ABD for a particular application can be circumvented if the user can afford a moderate amount of storage redundancy to achieve good worst-case behavior. By moderate I really mean moderate—the redundancy factor is not subject to combinatorial explosion as in the designs of Ghosh et al. Furthermore, both the worst-case and average behavior is even slightly improved over the designs of § 3.1 and the ABD's of § 3.2.

The technique is actually quite simple, and will be illustrated by an example. Suppose we have a file of 2^{30} 100-bit records (that is, each record consists of 100 one-bit keys). The method of the previous section would have required the construction of an ABD(100, w), for w near 20—a difficult task. Let us instead simply create five ($= 100/20$) independent filing systems, and let each record be filed once in each system. Each bucket system will have 2^{20} lists. The first system will use the first 20 bits of each record as its list index, the second system will use the second 20 bits of the record, and so on.

Now suppose we have a query $q \in Q_s$. At least one of the five systems will have at least $s/5$ bits specified for its list index—so we can use this system to retrieve the desired records. The number of buckets searched is not more than $2^{20-\lceil s/5 \rceil}$.

In general, if $b = 2^w$ is the number of buckets per system, and we have k-bit records to store (records with nonbinary keys can, of course, always be encoded into binary), we will establish $m = k/w$ distinct bucket systems, divide the record into m w-bit fields, and use each field as a bucket address in one of the systems.

The worst-case behavior of this scheme follows a strict geometric inequality:

$$W_s(h) \leqq 2^{w-\lceil ws/k \rceil}.$$

This surpasses even the best achievable *average* behavior of hash functions with no storage redundancy, although not by very much. If half of the bits are given in a query (i.e., $s = k/2$), then at most $\text{sqrt}(b) = 2^{w/2}$ buckets need be searched. The

average behavior of this scheme is difficult to compute, but it seems likely that it will approach the worst-case behavior, especially if w is large.

The above idea can be generalized further. Instead of taking each of the m subfields of the record and using it directly as an address, one can treat each subfield as a record and use an ABD(k/m, w) or some other method (such as the trie algorithm of § 4) to calculate an address from each subfield. The efficiency of this composite method will, of course, depend on the efficiency of the methods chosen for each subfield.

4. Trie algorithms for partial-match queries. The results of § 3, that an optimal block shape for a hashing algorithm for partial-match retrieval is a "subcube" of Σ^k, suggests using tries as an alternative data structure. Since the set of records stored in each subtrie of a trie is a subcube of Σ^k, recursively split into smaller subcubes at each level, tries might perform as efficiently as the optimal hash functions of the preceding section.

4.1. Definition of tries. Tries were introduced by Rene de la Briandais [11], and were further elaborated on by E. Fredkin [15], D. E. Knuth [28, § 6.3], and G. Gwehenberger [23].

A trie stores records at its leaves (external nodes). Each internal node at level i (the root is at level 1) specifies an $|\Sigma|$-way branch based on the ith character of the word being stored. As an example, consider the file

$$F = \{000, 001, 100, 101, 111\}$$

of three-bit binary words (that is, $\Sigma = \{0, 1\}$). They would be stored in a trie shown in Fig. 2. Here a left branch is taken when the corresponding bit is zero; a right branch is taken otherwise. A record is stored as a leaf as soon as it is the only record in its subtree; this reduces the average path length from the the root to a leaf in a random binary trie from k to about $\log_2|F|$ (see [28, § 6.3]).

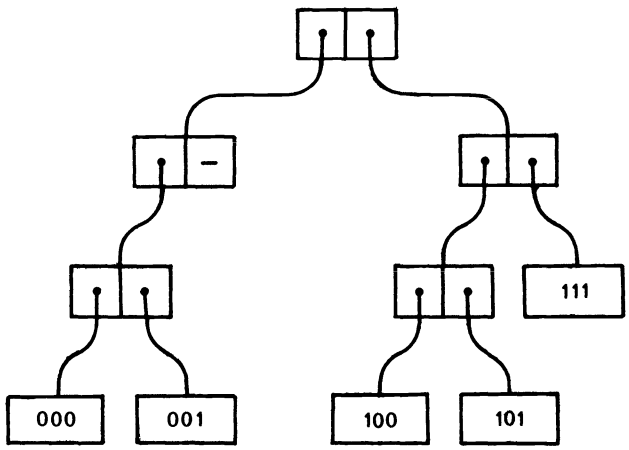


FIG. 2. *A trie*

A link to an empty subtree, such as that for records of type 01^* in the figure, is indicated by the null link "–". Useless internal nodes having but one nonnull link can be eliminated by storing at each internal node the character position on which to branch. This refinement, introduced by D. R. Morrison [33], shall not be pursued further here. Knuth has shown that the number of internal nodes will be roughly $|F|/(\ln 2) \approx 1.44|F|$, if $k \gg \log_2|F|$, for random binary tries; here the number of such useless nodes will not be excessive.

4.2. Searching tries. Given a partial match query $q = (q_1, \cdots, q_k)$, where $q_i \in (\Sigma \cup \{\text{“*”}\})$ for $1 \leqq i \leqq k$, and a trie T with root node N, the following procedure prints all records in T satisfying q. The initial call has the argument *level* equal to one. If M is an internal node of T, the M_c denotes the root of the subtree corresponding to the character $c \in \Sigma$, or *null* if no such subtree exists.

Procedure Triesearch (N, q, level);
begin
if N is a leaf containing record r **then**
    begin if r satisfies q **then** print (r) **end**
else if $q_{\text{level}} \neq$ “*” **then**
    begin if $N_{q_{\text{level}}} \neq$ **null** then Triesearch ($N_{q_{\text{level}}}$, q, level + 1) **end**
else for $c \in \Sigma$ **do**
    if $N_c \neq$ null **then** Triesearch (N_c, q, level + 1)
end Triesearch

4.3. Analysis of Triesearch. We will restrict our attention to binary tries in this section; the analysis for general $|\Sigma|$-ary tries would be similar. Since for nonbinary alphabets, record r could be encoded in binary by concatenating binary representations of each character r_i (for $1 \leqq i \leqq k$), this entails no real loss in generality; binary tries can be used to store arbitrary data. In fact, Theorem 1 suggests that this might be even a good idea, since we could then branch on the first bit of the representation of each character of r before branching on the second bit of each, and thus obtain record-spaces for each subtree more closely in agreement with Theorem 1. Bentley [3] has examined a similar approach in more detail; we shall not pursue it further here.

Our cost measure $c(n, s, k)$ shall be the average number of internal and external nodes examined by Triesearch to respond to a partial match query $q \in Q_s$, given that the trie contains n k-bit records.

Consider an arbitrary node M at level $w+1$ in a trie. There are at most 2^w nodes at this level. Let $m_1 \cdots m_w$ denote the common prefix of the records in the subtree with root M; the bits $m_1 \cdots m_w$ specify which w branches to take to get from the root of the trie to M. Finally, let $p(n, w, k)$ denote the probability that in a random trie there is a node M at level $w+1$ with prefix $m_1 \cdots m_w$ (this is independent of the actual values of $m_1 \cdots m_w$ if each n-record file F is equally likely). Note that there will be such a node if and only if the number of records with prefix $m_1 \cdots m_w$ is not zero and the number with prefix $m_1 \cdots m_{w-1}$ is not one. Thus

$$p(n, w, k) = \binom{2^k}{n}^{-1}\left[\binom{2^k}{n} - \binom{2^k - 2^{k-w}}{n} - \binom{2^k - 2^{k-w+1}}{n-1} \cdot 2^{k-w}\right].$$

Assuming that $n \ll 2^k$, we may approximate the latter two terms by the probability of having zero (resp., one) success in 2^{k-w} (resp., 2^{k-w+1}) trials, where the probability of success is $n2^{-k}$. Using the Poisson approximation to the binomial distribution, we have

$$p(n, w, k) \cong 1 - \exp(-n2^{-w}) - n2^{-w} \exp(-n2^{-w+1}).$$

This expression is independent of k, as we might expect, since once enough bits of a record are known to distinguish it, it is stored as a leaf in the trie, independent of the total record length. The function $p(n, w, k)$ is very nearly a step function; it is approximately 1 for $w < \log_2(n)$, going very quickly to 0 for $w > \log_2(n)$, passing the value $\frac{1}{2}$ at $(\log_2(n) - .0093)$, approximately.

Thus probability that a node M at level $w+1$ will be examined for a partial match query $q \in Q_s$ is just $A_{\min}(k, w, s)$, where $A_{\min}(k, w, s)$ is the function defined in § 3.12. Since there are 2^{-w} nodes at level $w+1$ in a complete binary tree, we get that the total expected cost of Triesearch is

$$1 + \sum_{1 \leq w \leq k} p(n, w, k) \cdot A_{\min}(k, w, s).$$

$$\cong \sum_{1 \leq w \leq k} (1 - \exp(-n2^{-w}) - n2^{-w} \exp(-n2^{-w+1}))(2 - s/k)^w.$$

Substituting $w = \log_2(n) + z$, we get that the above is equivalent to

$$\sum_{-\log_2(n) \leq z \leq k - \log_2(n)} (1 - \exp(-2^{-z}) - 2^{-z} \exp(-2^{1-z}))(2 - s/k)^{\log_2(n)+z}.$$

For $z \leq -1$, $p(n \log_2(n) + z, k) \geq .82$, so the sum for negative z is $ck(k-s)^{-1}(2 - s/k)^{\log_2(n)}$ for some c, $.82 \leq c \leq 1.00$. For $z \geq 0$, the sum is maximized when $s/k = 0$. However, even in this case the sum is not more than 1.54 $(2 - s/k)^{\log_2(n)}$, by numerical calculation, so that the total cost of the algorithm is approximately

$$ck(k-s)^{-1}(2 - s/k)^{\log_2(n)},$$

where the constant c of proportionality is less than 2.54.

4.4. Conclusions on tries. Tries are roughly as efficient as the optimal hashing algorithms for random data for which the number of lists used is $|F|$. For the usual situation involving highly nonrandom data, tries are probably the best practical choice, since the tries will store any data efficiently, whereas a hashing algorithm which selects record bits to use as a list index might result in a large number of empty lists and a few very long lists. For nonrandom data, an interesting problem arises if the branching decision may be made on any of the untested bits; we wish to choose the bit on which to split the subfile that will yield the best behavior. (Note that it is the most unbalanced tries which perform best.) For this modification, it may also be possible to take into consideration the probability that any given bit may be queried.

5. Conclusions. The hashing and trie-search algorithms presented perform more efficiently than any previously published partial-match search algorithms.

Retrieving from a file of n k-bit words all words that match a query with s bits specified takes approximately $n^{\log_2(2-s/k)}$ time, a little more than $n^{1-s/k}$, our conjectured lower bound on the time required by any algorithm. The main open problems are the proof or disproof of this conjecture, the existence questions for ABD's of general parameters, and the generalization of the results of this paper to handle nonrandom data with nonuniform query distribution probabilities.

Acknowledgments. I would like to thank the many people who provided constructive criticism of this research while in progress, with particular thanks to N. G. de Bruijn, Robert W. Floyd, Ronald Graham, David Klarner, Donald Knuth, Malcolm Newey and Franco Preparata. I would also like to thank Terry Bowes and Allison Platt for typing the manuscript and drawing the figures.

REFERENCES

[1] C. T. ABRAHAM, S. P. GHOSH AND D. K. RAY-CHAUDURI, *File organization schemes based on finite geometries*, Information and Control, 12 (1968), pp. 143–163.

[2] M. ARISAWA, *Residue hash method*, J. Inf. Proc. Soc. Japan, 12 (1971), pp. 163–167.

[3] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching*, unpublished manuscript, Dept. of Computer Sci., Univ. of North Carolina, Chapel Hill.

[4] J. L. BENTLEY AND R. A. FINKEL, *Quad trees: A data structure for retrieval on composite keys*, Acta Informat., 4 (1974), pp. 1–10.

[5] R. C. BOSE AND W. S. CONNOR, *Combinatorial properties of group divisible incomplete block designs*, Ann. Math. Statist., 23 (1952), pp. 367–383.

[6] R. C. BOSE, C. T. ABRAHAM AND S. P. GHOSH, *File organization of records with multi-valued attributes for multi-attribute queries*, Combinatorial Mathematics and its Applications, UNC Press, 1969, pp. 277–297.

[7] R. C. BOSE AND GARY G. KOCH, *The design of combinatorial information retrieval systems for files with multiple-valued attributes*, SIAM J. Appl. Math., 17 (1969), pp. 1203–1214.

[8] D. K. CHOW, *New balanced-file organization schemes*, Information and Control, 15 (1969), pp. 377–396.

[9] W. S. CONNOR, JR., *On the structure of balanced incomplete block designs*, Ann. Math. Statist., 23 (1952), pp. 57–71.

[10] D. R. DAVIS AND A. D. LIN, *Secondary key retrieval using an IBM 7090-1301 system*, Comm. ACM, 8 (1965), pp. 243–246.

[11] R. DE LA BRIANDAIS, Proc. Western Joint Computer Conference, 15 (1959), pp. 295–298.

[12] P. J. DENNING, *Virtual memory*, Comput. Surveys, 2 (1970), pp. 153–189.

[13] J. A. FELDMAN AND P. D. ROVNER, *An ALGOL-based associative language*, Comm. ACM, 12 (1969), pp. 439–449.

[14] W. FELLER, *An Introduction to Probability Theory and its Applications*, vol. 1, John Wiley, New York, 1950.

[15] E. FREDKIN, *Trie memory*, Comm. ACM 3 (1960), pp. 490–500.

[16] S. P. GHOSH AND C. T. ABRAHAM, *Application of finite geometry in file organization for records with multiple-valued attributes*, IBM J. of Res. Develop., 12 (1968), pp. 180–187.

[17] S. P. GHOSH, *Organization of records with unequal multiple valued attributes and combinatorial queries of order 2*, Information Sci., 1 (1969), pp. 363–380.

[18] ———, *File organization: Consecutive storage of relevant records on a drum type storage*, IBM Res. Rep. RJ 895, 1971.

[19] ———, *File organization: The consecutive retrieval property*, Comm. ACM 15, (1972), pp. 802–808.

[20] H. J. GRAY AND N. S. PRYWES, *Outline for a multi-list organized system*, Paper 41, Ann. Meeting of the ACM, Cambridge, Mass., 1959.

[21] R. A. GUSTAFSON, *A randomized combinatorial file structure for storage and retrieval systems*, Ph.D. thesis, Univ. of South Carolina, Columbia, 1969.

[22] ———, *Elements of the randomized combinatorial file structure*, Proc. of the Symposium on Inf. Storage and Retrieval, ACM SIGIR, Univ. of Maryland, April 1971, pp. 163–174.

[23] G. GWEHENBERGER, *Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen*, Elektron. Rechenanl., 10 (1968), pp. 223–226.

[24] G. H. HARDY, J. E. LITTLEWOOD AND G. PÓLYA, *Inequalities*, Cambridge University Press, London, 1959.

[25] D. HSIAO AND F. HARARY, *A formal system for information retrieval from files*, Comm. ACM, 13 (1970), pp. 67–73.

[26] L. R. JOHNSON, *An indirect chaining method for addressing on secondary keys*, Ibid., 4 (1961), pp. 218–222.

[27] J. R. KISEDA, H. E. PETERSON, W. E. SEEDBACK AND M. TEIG, *A magnetic associative memory*. IBM J. Res. Develop., 5 (1961), pp. 106–121.

[28] D. E. KNUTH, *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1972.

[29] G. G. KOCH, *A class of covers for finite projective geometries which are related to the design of combinatorial filing systems*, J. Combinatorial Theory, 7 (1969), pp. 215–220.

[30] J. MAUCHLY, *Theory and Techniques for the Design of Electronic Digital Computers*, G. W. Patterson, ed.; vol. 1, 1946, §§ 9.7–9.8; vol. 3, 1946, §§ 22.8–22.9.

[31] J. MINKER, *An overview of associative or content addressable memory systems and a KWIC index to the literature*, Tech. Rep. TR-157, Univ. of Maryland Computer Center, College Park, 1971.

[32] ———, *Associative memories and processors: A description and appraisal*, University of Maryland and Auerbach Corp. Tech. Rep. Tr-195, 1972.

[33] D. R. MORRISON, *PATRICIA—Practical algorithm to retrieve information coded in alphanumeric*, J. Assoc. Comput. Mach., 15 (1968), pp. 514–534.

[34] S. NISHIHARA, S. ISHIGAKA AND H. HAGIWARA, *A variant of residue hashing method of associative multiple-attribute data*, Rep. A-9, Data Processing Center, Kyoto Univ., Kyoto, Japan, 1972.

[35] W. W. PETERSON AND E. J. WELDON, *Error-Correcting Codes*, MIT Press, Cambridge, Mass., 1972.

[36] N. S. PRYWES AND H. J. GRAY, *The organization of a Multilist-type associative memory*, IEEE Trans. on Communication and Electronics, 68 (1963), pp. 488–492.

[37] N. S. PRYWES, *Man-computer problem solving with Multilist*, Proc. IEEE, 54 (1966), pp. 1788–1801.

[38] F. PREPARATA, Personal communication.

[39] D. K. RAY-CHAUDURI, *Combinatorial information retrieval systems for files*, SIAM J. Appl. Math., 16 (1968), pp. 973–992.

[40] R. L. RIVEST, *Analysis of associative retrieval algorithms*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, Calif., 1974.

[41] ———, *On the optimality of Elias's algorithm for performing best-match searches*, IFIP Proc., Stockholm, Aug. 1974, pp. 678–681.

[42] J. A. RUDOLPH, *A production implementation of an associative array processor—STARAN*, Proc. Fall Joint Comp. Conf., 1972, pp. 229–241.

[43] E. D. SACERDOTI, *Combinatorial mincing*, Unpublished manuscript, 1973.

[44] A. E. SLADE, *A discussion of associative memories from a device point of view*, American Documentation Institute 27th Ann. Meeting, 1964.

[45] T. A. WELCH, *Bounds on the information retrieval efficiency of static file structures*, Project MAC Rep. MAC-TR-88, Mass. Inst. of Tech., Cambridge, Mass., 1971; Ph.D. thesis.

[46] P. F. WINDLEY, *Trees, forests, and rearranging*, Comput. J., 3 (1960), pp. 84–88.

[47] E. WONG AND T. C. CHIANG, *Canonical structure in attribute based file organization*, Comm. ACM, 14 (1971), pp. 593–597.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

MACHINE SELECTION OF ELEMENTS IN CROSSWORD PUZZLES: AN APPLICATION OF COMPUTATIONAL LINGUISTICS*

LAWRENCE J. MAZLACK†

Abstract. This paper reports on the construction of a crossword puzzle generator. After an unsuccessful attempt to construct puzzles by whole word insertion, puzzles were constructed letter by letter. Heuristically determined decision structure was required. The constructor resolved questions of letter selection, ordering and reordering of the solution sequence, dictionary structure and access, and decision path selection. The decision basis for letter selection was based on a pseudo-probabilistic approach.

Key words. crossword puzzles, computational linguistics, heuristic symbolic processing, artificial intelligence, nonhuman solution method

1. Introduction. The widespread use of computers is a recent phenomenon. Most machines are essentially used as glorified adding machines. However, computers are also capable of being symbol manipulators, and this is emerging as one of the most interesting areas of computer study. Within symbol manipulation, the area of text processing is one of the greatest interest. An application of text processing that has not been examined in any detail is the special problem of crossword puzzle construction. Crossword puzzles are presently in nearly every newspaper in the country, with widely varying levels of complexity. On the level of symbol manipulation alone, computer construction of crossword puzzles is an interesting problem.

However, it would also seem to be fair to assert that the construction and solution of crossword puzzles is an activity requiring direction and some degree of intelligence. On this basis, an investigation into if and how computers go about constructing crossword puzzles is a worthwhile endeavor as it can provide us with clearly defined problem results and hints with regard to the question of the nature of machine intelligence and its potential.

Additionally, the problems and insights that arise in the construction of crossword puzzles and double-crostics has a relationship to mechanical language translation [1] and to a knowledge of the patterns that are examined in linguistic analysis, both on a micro and on a macro basis.

There are really two different problem areas related to crossword puzzles. The problem that confronts most people is the completion of a puzzle from a list of clues. However, before this problem can occur, the puzzle must be laid out by a puzzle builder.

This paper addresses itself to the development of computer oriented techniques for the initial building of crossword puzzles. The most directly applicable methods of rapid puzzle construction were investigated.

The techniques described in this paper center around a mechanism, called a puzzle constructor, which seeks to build a crossword puzzle within a construction called a puzzle matrix. This mechanism uses a combination of heuristic rules and applied probability. The constructor will only produce a filled in crossword

* Received by the editors August 5, 1974, and in revised form February 27, 1975.

† Department of Computing and Information Science, University of Guelph, Guelph, Ontario, Canada.

puzzle. It will not generate clues or definitions. Clue generation is an art form in itself, and the criteria involved are of a literary nature.

The initial input to the puzzle constructor is a crossword puzzle matrix of blanks and null spaces with one or more initial words loaded into the puzzle to initiate the constructor, as in Fig. 1.

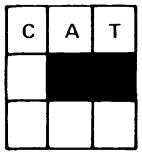


FIG. 1. *Initial puzzle entry form*

The purpose of the initial word is to provide a starting point for the constructor much like the initiating number in a random number routine. Any word of the appropriate length may be inserted. The initial word may or may not completely fill a given word space. More than one initial word may be used.

The puzzle shape to be solved can be of any form. For example, it could be Christmas tree shaped for use during the Christmas season.

A dictionary of words to be potentially used was developed. Words that were not in the dictionary being used were considered to be illegal. A dictionary is defined as the listing of all the words eligible to be used as entries into the puzzle. In a published dictionary these would be the lexes, or the words that the dictionary entries define.

A dictionary is a large thing to handle. In the dictionary's machine readable form, it must not consume too much space. Dictionary format, storage and search proved to be a major area of evaluation and development. The constructor approaches explored were not algorithmic approaches. An algorithmic approach guarantees a solution, if there is one. The puzzle constructor follows a set of heuristics to attain a solution. Heuristics may be thought of as shortcuts, or as rule of thumb approaches. They are used to provide a speedy solution. Complete enumeration, i.e., successively inserting all possible word combinations, would be an algorithmic approach to puzzle building. However, such an approach would require an excessive amount of machine time.

The consideration of the constructor's solution strategy is different from puzzle constructor heuristics. The solution strategy of the constructor must produce a basic approach to the problem. The heuristics take that approach and apply it in a way that will produce a problem solution.

The constructor was to run on an IBM 360/50 in 212K or less. (212K was the maximum storage available.) An increase in storage would allow a larger core resident dictionary. Likewise, a decrease in storage would decrease the maximum dictionary size that could be core resident. In addition, cards and magnetic tape were the only storage media available other than the CPU core itself. An extremely severe restriction was the nonavailability of disk storage.

2. Previous and related work. The subject matter of this paper is largely unexplored. Mention of one previous constructor has been found [2]. This program had few points of intersection between words and was solved by complete

enumeration. Additionally, one simple double-crostic solver [1] and one common puzzle solver [3] were found.

With these exceptions, no other algorithms were discovered. However, it does seem plausible that someone had previously developed a trivial exhaustive iterative solution.

This topic does not immediately drop into any single category. Many of the techniques used come out of text processing, while the strategies and heuristics relate to some of the topics in artificial intelligence. Quantitative linguistics generally deals with the meaningful relationship between words and word groups [4]. Of course this is not uniformly true; people like Herdan [5] and Dewey [6] have dealt extensively with phonology as well as with word relationships. Nonetheless, most work in the linguistics area that appeared to be potentially beneficial tended to be either an analysis of existing texts [7], or an abstracting into transformation analysis [8], [9]. The use of the latter type of analysis would appear to be similar to dealing with a theorem prover like that of Newell, Shaw and Simon [10]. There are àreas of common concern. Techniques used in language translation [11] were used in the final puzzle constructor.

The strategy followed by the puzzle constructor is abstract in the same sense as Jacobs' PERCY [12] was, in that decisions are made without exploring the course of actions involved in executing these decisions. Instead of fully examining the implications of any decisions, a set of goals intermediate to the principal task are evaluated. Another point of similarity is that a strategy component evaluates and selects goals while other components take responsibility for decision execution.

Many of the techniques eventually used in the problem resolution have a direct relationship to game programs. For example, Shannon's [13] concept of node evaluation, also later applied by Samuel [14], was used in the precedence ordering of the puzzle constructor. However, their tree search strategies were not applied. Instead, the process was more like Slagel's [15] calculus problem solver.

The techniques used in this work take maximum advantage of previous approaches to other problems. However, there was little direct application as the construction of crossword puzzles is an endeavor largely divorced from previous work.

3. Significant problem areas.

3.1. General considerations. There are several major interconnected problems. They are dictionary form and compaction, dictionary search techniques, word construction techniques, solution strategy and overall puzzle constructor heuristics. This paper will concentrate on word construction techniques and on solution strategy.

By dictionary compaction, it is meant the compression of a dictionary of words to be used in the construction of the crossword puzzles into a readily machine-storable form.

Dictionary search techniques refer to the methodology by which a given word dictionary is examined to return an answer to a query. The query may be of several forms. It might ask if a word is in the dictionary. It might ask for the return

of a word of a specified nature, for example, a word of six characters with the last letter being an "A".

3.2. Constructor strategy—Whole word vs. letter by letter. Initially, the constructor was designed to fit whole words into a given word space, which is apparently the method used by human puzzle constructors. This technique (not considered herein) was eventually set aside as it amounted to a solution by complete enumeration. In addition, this method was highly time and space consumptive.

3.3. Dictionary form, search and compaction techniques. It is not really possible to solve any of the crossword puzzle constructor problem areas by itself. Dictionary search and compaction are really two separate questions with a high degree of interrelationship. The way the dictionary is laid out strongly constrains the search techniques that may be used.

There are four dictionary problem areas that must be resolved. They are: 1. dictionary contents, 2. dictionary organization, 3. what to search for and 4. search techniques.

It was necessary to compact the dictionary so that it could fit into core, as system limitations forced the complete dictionary to be core resident. However, the dictionary must be rapidly accessed after it is compacted.

An extremely fast accessing scheme for one set of conditions might be very slow for another set of necessary conditions. Or, the search scheme could cause the dictionary to blow up to unmanageable size. After much investigation (not considered herein), a dictionary and attendent cross reference table was constructed. The dictionary was in a tree form specified in tableau form and allowed backward and forward travel through the tree.

3.4. Letter by letter approach and heuristics. The letter by letter method fills in each letter by itself without an immediate reference to a given whole word. The idea behind this approach is that if each local area of the puzzle is correct, the puzzle in its entirety can eventually be made to be valid. This approach is much like solving boundary value problems in heat transfer [16] or electric fields by relaxation [17].

There are two basic approaches to heuristic search. They are labyrinthic methods and node evaluation methods [18]. A labyrinthic method proceeds down a search tree and has an explicit mechanism for deciding direction in the tree. GPS [19] uses a labyrinthic approach. The whole word approach is largely a labyrinthic approach.

Node evaluation methods use an evaluation function which assigns a value to each direction, based upon conditions when the node is evaluated. Samuel's checker playing programs [14] are examples of this approach.

The letter by letter method falls largely into a node evaluation scheme. The whole word method is more like a labyrinthic method. The whole word method tries to fit words into the puzzle spaces and backs up when it cannot. The whole word constructor was to start from the initial words in the puzzle and move forward in a determined path. Words fitting into the combination of spaces and letters existing in the puzzle are sequentially inserted. The letter by letter constructor

fills empty spaces whose solution order is determined by a precedence stack. The initial order of precedence may be changed depending on the relative success of the constructor at filling given spaces. The initial precedence ordering is determined by a desirability weighting of the empty puzzle spaces. The letters selected to go into each blank space are determined by a statistical selection scheme.

4. Letter by letter method. When constructing a crossword puzzle letter space by letter space, a dictionary search is made only when a word is completed to determine if the word space is filled with a valid word. This approach eliminates the problem of illegal letter combinations in a word space that can occur when two whole words are selected and are placed in parallel to each other.

When filling in crossword puzzles letter space by letter space, the constructor repeatedly faces the following situation: each letter space to be filled in is simultaneously located in two words (a horizontal one and a vertical one) in which one or more letters have already been filled in. Letter spaces (2, 1), (3, 1), (3, 3), (vertical, horizontal) of Fig. 2 illustrate this condition.

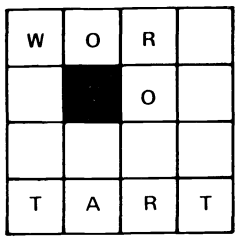


FIG. 2. *Partially completed puzzle*

In degenerate cases, one of the two words vanishes as illustrated by letter spaces (1, 4), (2, 4), (3, 2) of Fig. 2.

Two basic problems must be solved:

1. In what order should the empty letter spaces in the puzzle be filled in?
2. What letter should be inserted in each empty space?

The method used to solve these problems is discussed in more detail in the next two sections.

4.1. Letter space precedence establishment. In constructing a puzzle, letter by letter, there is a question of the precedence in which the blank letter spaces are to be filled. It is apparent that construction must begin in relation to the initial or key words (for example, "CAT" is the key word in Fig. 3(a) of the puzzle. Otherwise, the words constructed may turn out to be incompatible with the key word(s). If word space 3 of Fig. 3(b) contains the first letter space filled, then the resulting construction may conflict with the key word and no solution may be quickly realizable.

Another decision that must be made is whether or not to complete each word space as soon as possible after any letter space within it is filled, or to fill each

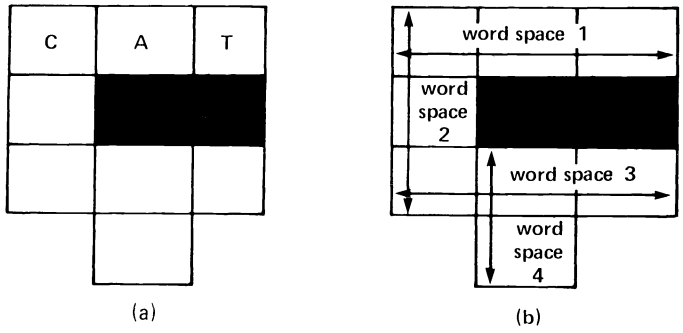


FIG. 3. *Puzzle with key word of "CAT"*

letter space according to a tasking technique. For example,

C	A	T
	R	

FIG. 4. *Partially filled* 3 × 3 *puzzle with "CAT" being the key word*

in Fig. 4, if letter spaces (1, 1), (1, 2), (1, 3) (vertical, horizontal) contain the key word, and if letter space (2, 2) is the first letter space filled, should the constructor fill letter space (3, 2) next and complete the word, or should it fill some other letter space?

The advantage in filling (3, 2) first is that we will have completed a word. However, completion of a word space, per se, does not lead us to Valhalla, as the word formed may make the valid completion of other word spaces impossible. In addition, this approach separates from the approach of solution by relaxation. The end product of the constructor is to fill all the letter spaces as well as all the word spaces. It was then decided to order the filling of the letter spaces by a tasking algorithm which is dependent on each letter space's "weight". Letter space precedence ordering will be discussed in greater detail later.

When an illegal word is generated in a word space, then the constructor backs up and deletes the last letter inserted in the word space of the illegally formed word. If it is possible to insert another letter there, the constructor will do so; if not the constructor will back up further.

The letters selected to fill each letter space will be done using occurrence ratios. Occurrence ratios are essentially a set of letter within word positional, serial, and displacement statistics which were collected for this purpose. They describe the fraction of all words of a given type (for example, six letter words beginning with a letter "B") in which the kth letter has a specific value, such as "S".

4.1.1. Letter space precedence ordering. The letter spaces are ordered for solution by placing them in a letter space precedence stack, henceforth known as the precedence stack. Not all the letter spaces in the puzzle matrix are initially

placed in the precedence stack. Initially, only the letter spaces horizontally or vertically "visible" to the key word(s) are placed onto the precedence stack. For example, in Fig. 5(a), the letter spaces (2, 1), (2, 3), (2, 4), (3, 1), (3, 3) and (4, 3) are visible to the key word "BOAT". The letter spaces (3, 2), (4, 2) and (4, 4) are not. Note, that letter spaces may be added to the precedence stack by more than one key word, for example (3, 2) in Fig. 5(b).

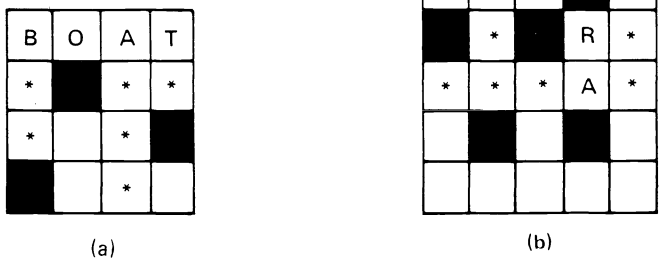


FIG. 5. *Initial precedence stack loading (marked by *)*

A heuristic weight is assigned to each letter space. The purpose of the weighting is to attempt to place a numeric value on each letter space to indicate its relative importance to the solution. The higher the weight, the greater the value. (The weighting calculations are described in the following section.) The visible letter spaces are ordered by weight, greatest to the top. The ordered letter spaces are then placed at the top of the precedence stack. Letter spaces not in the initial stack ordering are added to the precedence stack as the solution progresses. As additional letters are selected, any cells not already in the solution stack that are adjacent to them are placed on the bottom of the solution stack, sorted within the subgroup, highest first.

The solution stack contains the following information:

1. *Vertical position* of the matrix cell that it represents.
2. *Horizontal position* of the matrix cell that it represents.
3. *Positional weight* of the matrix cell that it represents.

The vertical and horizontal position information provide an indirect reference to a more complex cell structure which contains information used by the puzzle constructor in resolving the puzzle.

4.1.2. Weighting. The letter space heuristic weighting was developed by considering the degrees of freedom and the interconnections of each letter space. The reasons behind considering these factors were:

1. It is more important to complete long words than short words as the ratio of valid words to all possible character string permutations decreases with word length.

2. Letter spaces with letter spaces contiguous to them in four directions, as letter space 1 in Fig. 6, are more important than more restricted letter spaces, as letter spaces 2 and 3 of Fig. 6. This is simply because the letter space having more open directions must occur in more letter strings.

3. Denser locations should be resolved first. In this context, a denser location means a letter space that is directly connected through a word space to relatively more letter spaces. For example, in Fig. 6, letter space Cb is the densest location as it is directly connected to 5 other letter spaces (Ab, Bb, Ca, Cc, Db) while letter space Dd is the least dense location, being connected to no other letter space.

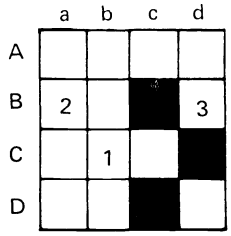


FIG. 6. *Letter space weighting relative importance of particular letter spaces*

This method of precedence establishment was found to be relatively successful. Generally the reordering of the relative letter space precedence was not found to be of great magnitude. When the magnitude of the reordering was large, a pattern in the reordering was not apparent. One type of systematic variation would be the forced serial ordering of the letter spaces into either a left to right or a right to left completion of a given word space. Such an ordering does occasionally occur, but deserialization occurs as often. Figure 7 shows the reordering required in the solution of Puzzle 20. Changes occur in the precedence relationship between letter spaces due to the constructor's reordering the precedences to resolve construction difficulties.

4.2. Letter selection. The following information is available to the puzzle constructor faced with the problem of filling in a vacant letter space which exists in one or two partially completed word spaces:

1. The lengths of the horizontal and vertical words.
2. The position of the intersection-space in the horizontal and vertical words.
3. The positions of previously-filled-in letters (relative to the vacant letter space) in the horizontal and vertical words.

The dictionary of allowable words can be used to compile statistics from the dictionary being used for the puzzle (prior to the construction of the crossword puzzle) on the relative suitability of different letters placed at the intersection. There are several different statistics that could be used. These statistics can then be developed into occurrence ratios which indicate the relative suitability of inserting different letters at a vacant letter space. The following sections discuss in more detail the statistics used by this puzzle constructor.

Depending on the puzzle geometry and the previously inserted letter spaces, various statistics may be applied. The applied occurrence ratios determine which letters are eligible for insertion and the attempted order of insertion. The mechanism for controlling the selection process is something defined as a "letter selection vector". This is discussed in more detail in the following sections.

					115/115		112/112	114/114		108/110	102/108	107/93
4/4	1/1	6/6			113/113		110/111			104/94	98/106	103/95
3/3	2/2	5/5	9/9		111/101	109/102	105/103	101/104			95/100	99/96
7/7	8/8		10/10	12/12		106/109	100/107	97/105		91/91	93/98	96/97
		14/15	11/11	13/13	16/14			94/99	92/92	89/89		
24/24	20/20	17/17		15/16	18/18	22/22	27/27		90/90	88/88	86/87	82/86
29/29	23/23			19/19	21/21	25/25	30/31					76/79
33/34	28/28	32/33	36/36		26/26	31/32	34/30	38/38		57/58	64/64	70/70
		35/35	40/40	43/44			37/37	41/41	44/45	50/50		
47/42	42/43	39/39		48/48	53/54	62/62		45/46	49/49	55/56	63/63	
52/53	46/47			54/55	61/61	67/67	72/72		56/57		69/69	75/78
59/59	51/52	60/60			68/68	71/71	78/81				74/77	81/85
65/66	58/51	66/65		79/83	73/73	77/80	83/82		87/74	85/75	80/76	84/84

FIG. 7. *Final and initial ordering of puzzle* 20 (*initial ordering/final ordering*)

4.2.1. Letter selection vector. A letter selection vector is developed for each letter space at the particular time that the attempt is being made to fill each letter space. This vector is to be calculated anew at every fresh attempt to fill a given letter space because of the potential changes in the letter spaces around it. The vector has 26 components, one for the calculated occurrence ratio of each of the 26 possible letters.

An occurrence ratio vector's elements are formed by taking the minimum occurrence ratio for each letter from the group of different occurrence ratios that can be ascribed to the vacant letter space under consideration.

After the occurrence ratio vector has been formed, the letter selection vector is formed by sorting the higher occurrence rates to the top. The heuristic concept is that the letter with the highest occurrence rate will advance the constructor more toward the final solution than the one with the next highest occurrence ratio. A zero letter selection vector element indicates a respective entry letter which is a letter that cannot validly occur and thus would hinder the final puzzle solution.

4.2.2. Letter selection vector formation.

1. *General.* The letter selection vector is constructed by examining the local area surrounding the letter space in question. In this case, the local area is defined as being all letter spaces extending in a horizontal or vertical direction from the letter space in question until a null space or an edge is met.

A set of three minimal occurrence vectors is formed using three types of occurrence ratios (to be described in part 2). Then the letter selection vector is formed by taking the minimum at each vector element.

2. *Selection of the occurrence ratios used.* In selecting the ratios to be used, several criteria were used. Compactness was considered to be important. It was decided to have all the occurrence ratios core resident to save lookup time. Hence, the occurrence ratio tables had to be relatively compact. The ratios chosen also had to be relatively disjoint. That is, each must supply different information in order to justify its existence. For example, it would be of no value to separately accumulate two letter series frequencies from both a right to left scan and a left to right scan of the same letter group as the statistics are symmetric.

It is implicit that the statistics must be useful. Obviously, when forced to choose between equally compact and disjoint ratios, the more useful one is to be chosen.

However, usefulness alone is not enough. For example, the knowledge of three-letter series occurrence ratios by beginning letter position within words of a given length would be very useful. But, it would use an excessive amount of storage.

Four types of occurrence ratios were selected. They were letter frequency by letter position within the word, two-letter series frequency, three-letter series frequency, and generalized two-letter series frequency. The nature of these statistics is discussed in more detail in the following three sections. These consumed the space available. The use of additional occurrence ratios would require either more core, the elimination of one of the previously selected ratios, or the use of on-line storage.

3. *Positional minimal vector.* The minimal vector coming out of the position statistics is of the form of (1), assuming n_v is the position in the word vertically, n_h is the position in the word horizontally, l_v is the length of the horizontal word passing through this space, and l_h is the length of the vertical word passing through this space.

$$\text{(1)}\quad \overline{\text{PMV}} = \min \left\{ \left[\begin{array}{l} \text{occurrence ratio of the} \\ \text{letter A in position } n_h \\ \text{in a word of length } l_h \\ \hline \text{occurrence ratio of the} \\ \text{letter B in position } n_h \\ \text{in a word of length } l_h \\ \hline \vdots \\ \hline \text{occurrence ratio of the} \\ \text{letter Z in position } n_h \\ \text{in a word of length } l_h \end{array} \right], \left[\begin{array}{l} \text{occurrence ratio of the} \\ \text{letter A in position } n_v \\ \text{in a word of length } l_v \\ \hline \text{occurrence ratio of the} \\ \text{letter B in position } n_v \\ \text{in a word of length } l_v \\ \hline \vdots \\ \hline \text{occurrence ratio of the} \\ \text{letter Z in position } n_v \\ \text{in a word of length } l_v \end{array} \right] \right\}.$$

(Note for formulas (1), (2), (3) and (4): If $\overline{U} = (u_1, \cdots, u_{26})$ and $\overline{V} = (v_1, \cdots, v_{26})$ are two vectors, then $\min(\overline{U}, \overline{V})$ means $(\min(u_1, v_1), \cdots, \min(u_{26}, v_{26}))$.)

4. *Series minimal vector.* The vector coming out of the two-letter and three-letter series statistics is similar, although it is more complex in derivation. $\overline{\text{SMV}}$

is the minimum vector formed from the series occurrence ratio tables. $\overline{\text{SMV}}$ is constructed by assuming the placement of letters A through Z for the location of the blank we are examining for any letter series which are already in the puzzle which extend vertically and horizontally contiguously away from the space we are trying to fill. For example, if the puzzle was of the form _D_AM_, and it was desired to fill the spaces between the "D" and the "A", $\overline{\text{SMV}}$ would be formed as follows in (2).

$$(2)\quad \overline{\text{SMV}} = \min\left\{\begin{bmatrix}\text{occurrence ratio of the letter A following the letter D}\\ \hline \text{occurrence ratio of the letter A following the letter D}\\ \hline \vdots\\ \hline \text{occurrence ratio of the letter Z following the letter D}\end{bmatrix}, \begin{bmatrix}\text{occurrence ratio of the letter A occurring before the letter A}\\ \hline \text{occurrence ratio of the letter B occurring before the letter A}\\ \hline \vdots\\ \hline \text{occurrence ratio of the letter Z occurring before the letter A}\end{bmatrix}, \begin{bmatrix}\text{occurrence ratio of the letter A occurring before the letters AM}\\ \hline \text{occurrence ratio of the letter B occurring before the letters AM}\\ \hline \vdots\\ \hline \text{occurrence ratio of the letter Z occurring before the letters AM}\end{bmatrix}\right\}.$$

Of course, if two letters are contiguously known on the left side as well, the three-letter series tables would be used. Also, normally, vertical series components would come into play.

5. *Generalized series minimal vector.* The vector coming out of the generalized series statistics is similar to the previous vectors. $\overline{\text{DMV}}$ is formed by taking the minimum of the occurrence ratios for distance letter combinations not immediately contiguous to the blank to be filled, but vertically and horizontally exposed to the blank that is to be filled. For example, assume that we are working on F_D__A_, and we are attempting to fill the space immediately preceding the A. Equation (3) shows how the vector is formed.

$$(3)\quad \overline{\text{DMV}} = \min\left\{\begin{bmatrix}\text{occurrence ratio of the letter A following the letter F at a distance 4}\\ \hline \text{occurrence ratio of the letter B following the letter F at a distance 4}\\ \hline \vdots\\ \hline \text{occurrence ratio of the letter Z following the letter F at a distance 4}\end{bmatrix}, \begin{bmatrix}\text{occurrence ratio of the letter A following the letter D at a distance 2}\\ \hline \text{occurrence ratio of the letter B following the letter D at a distance 2}\\ \hline \vdots\\ \hline \text{occurrence ratio of the letter Z following the letter D at a distance 2}\end{bmatrix}\right\}.$$

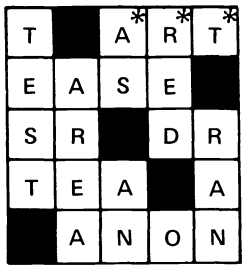
It is not necessary to consider distance 1 ratios. These ratios are redundant with the two-letter series ratios. In the previous example, these would be the occurrence ratios of a given letter preceding the letter A at a distance 1.

6. *Letter selection vector construction.* The letter selection vector is then formed in two steps. First, the vector minimum is taken of the three ratio vectors. Equation (4) expresses this in terms of (1), (2) and (3).

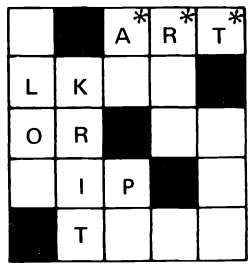
$$\overline{\mathrm{MV}} = \min(\overline{\mathrm{PMV}}, \overline{\mathrm{SMV}}, \overline{\mathrm{DMV}}). \tag{4}$$

$\overline{\mathrm{MV}}$ is the minimum vector. The minimum vector is then placed into a structure containing both the vector values and the associated letter number (i.e., A = 1, B = 2, $\cdots$, Z = 26). This structure is then sorted, forcing the structure elements with the highest minimum occurrence ratios to the top. The resulting structure is then called the letter selection vector.

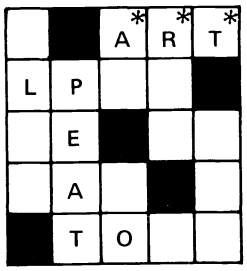
7. *Minimal set of occurrence ratios.* It was decided to see if the occurrence ratios selected constituted a minimal set, i.e., if solutions could be readily attained without all of the occurrence ratios. In order to perform this test, the puzzle shown in Fig. 8 was chosen. The constructor was run four times. A different solution



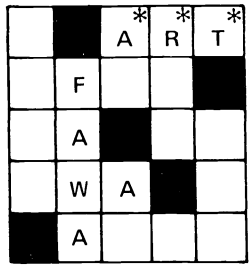
(a) Solution with all ratios in force, l_{78}



(b) Solution without two and three letter series ratios (last matrix before fail on l_{900})



(c) Solution without generalized series ratios (last matrix before fail on l_{478})



(d) Solution without positional ratios (last matrix before fail on l_{480})

FIG. 8. *Puzzle* 8 *solutions excluding various occurrence ratios* (* *indicates initiating letters*)

progression was developed each time. The first time was with all of the ratios being used to form the letter selections vector. Then the constructor was run three more times, eliminating the use of the generalized series ratios, the positional ratios and the series ratios. A dictionary consisting of 2,000 words with a maximum word length of four was used. The results are summarized in Table 1. No test runs were performed with the elimination of two ratios as the elimination of a single ratio caused failure in all cases.

TABLE 1
Excluded ratio test results for Fig. 8

Ratios excluded	Success/Failure	Iterations	Valid words generated
NONE	success	78	16
generalized series	failure	478	52
positional	failure	480	35
series	failure	900	16

4.2.3. Letter selection. When the precedence stack pointer points at the precedence stack information associated with a given letter space, the letter to fill the letter space is selected using the letter selection vector. The letter selection vector is created each time an attempt is made to fill any letter space.

If the present attempt at a given precedence stack level is the first attempt to fill the letter space, the letter at the top of the letter selection vector will be inserted into the letter space. The highest occurrence ratio is considered to be at the top of the letter selection vector. If the letter selected causes an illegal word to be formed, the letter just inserted will be deleted.

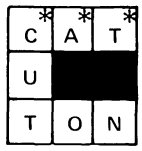
When a letter is deleted, the constructor will lower the pointer in the letter selection vector element. A cutoff upper bound of retries has been established. When the pointer in the letter selection vector indicates a zero element, or the upper bound on retries is exceeded, the constructor will back up and the precedence stack will be reordered.

4.2.4. Constructor back up. When a word space has been filled, a dictionary search is made to determine if the developed word is in the dictionary. If the word is in the dictionary, the constructor continues on to fill the next letter space in the precedence stack. If the word developed is not in the dictionary, it is not considered to be a valid word. When a nonvalid word is developed, the last letter inserted into that word is deleted. This initiates the back up process.

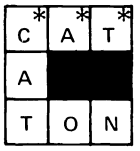
The back up process constrains the selection of a new letter and reorders the precedence stack. Because of space limits, this paper will not examine the back up process.

4.2.5. Precedence heuristics. A set of heuristics were developed to reorder the initial search sequence when construction difficulties were encountered. These heuristics will not be discussed here.

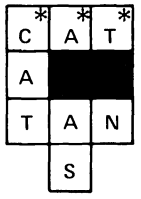
4.2.6. Solutions. Several puzzles were completed by the constructor. One set of the puzzles were those containing word spaces of up to four letters in length. Various size dictionaries were used. Machine constraints restricted the maximum dictionary size. The largest dictionary of words of length four or less that could be accommodated was one of 2,000 words.



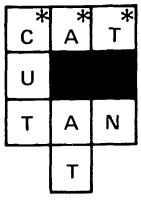
puzzle 1 (1)



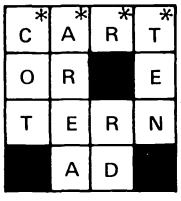
puzzle 1 (2)



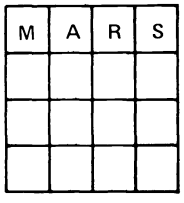
puzzle 2 (1)



puzzle 2 (2)

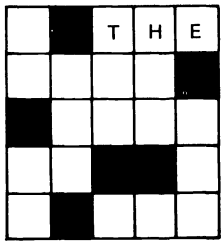


puzzle 3 (1) failed
puzzle 3 (2)

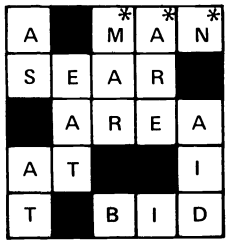


puzzle 4 (1,2) failed

puzzle 5 (2)



puzzle 6 (2)
failed

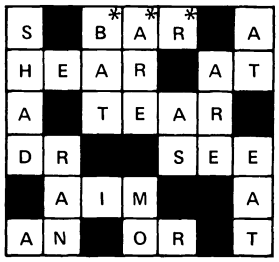


puzzle 7 (2)

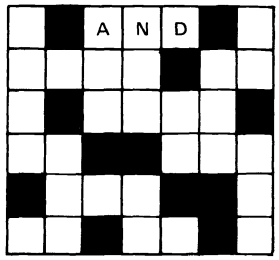
FIG. 9. *All puzzle constructor attempts* (* = *original keyword letter*, 1 = 1,000 *word dictionary result*, 2 = 2,000 *word dictionary result*)

Figure 9 (continued)

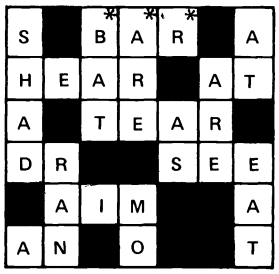
puzzle 8 (2)



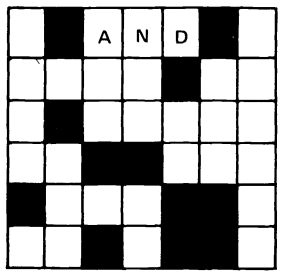
puzzle 9 (2)



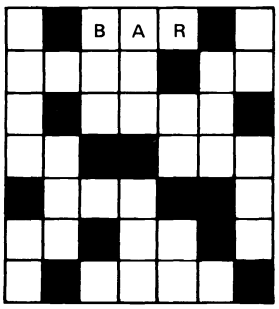
puzzle 10 (2) failed



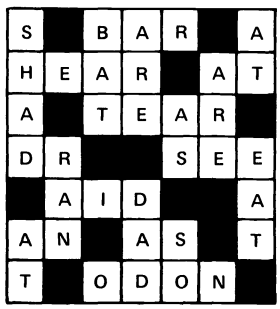
puzzle 11 (2)



puzzle 12 (2) failed

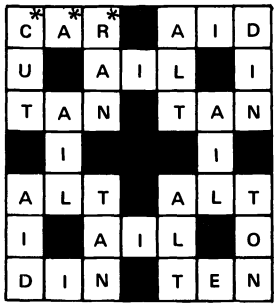


puzzle 13 (2)
failed

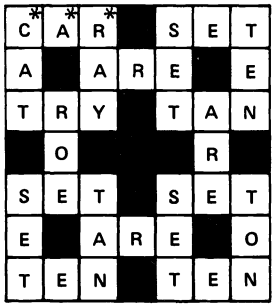


puzzle 13 (2)
observed solution, if ODON in dictionary (constructor failed to recognize the solution and claimed a failure).

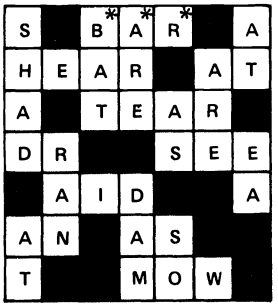
Figure 9 (continued)



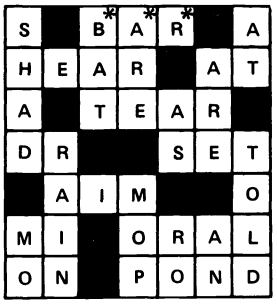
puzzle 14 (1)



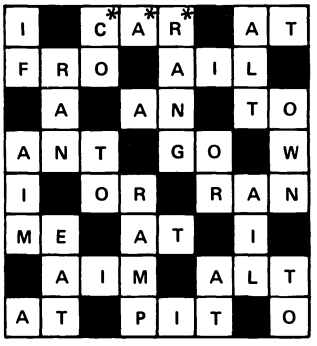
puzzle 14 (2)



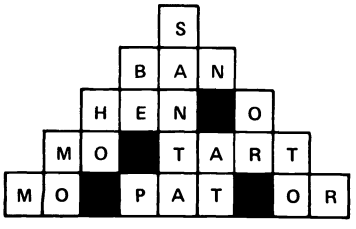
puzzle 15 (2)



puzzle 16 (2)

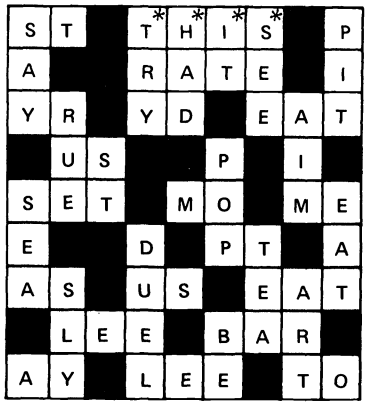


puzzle 17 (1) failed
puzzle 17 (2)

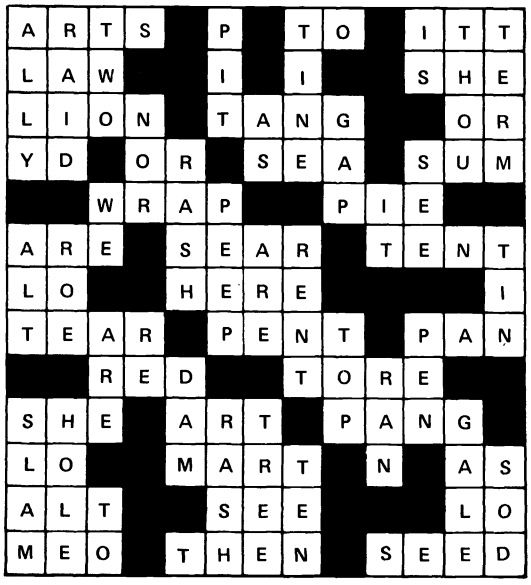


puzzle 18 (2)

Figure 9 (continued)



puzzle 19 (2)



puzzle 20 (2)

There appears to be a minimal dictionary size which is a function of maximum word size and the number of words in the dictionary. The dictionary richness cutoff for words of length four or less appears to lie somewhere between 1,000 and 2,000 words when the words are selected from all the words of from two to four letters of the Webster's Children's Dictionary [20]. A dictionary of 1,000 words of character four or less is inadequate, and generally fails to produce a valid solution.

Table 2 shows a comparison between the number of iterations and valid words generated between the 1,000 word dictionary and the 2,000 word dictionary runs. As can be seen, the use of the larger dictionary always produced more valid words per iteration. Also, if a solution was obtained, the solution was always faster utilizing a larger dictionary.

TABLE 2
Comparison between 1,000 *word and* 2,000 *word dictionary construction attempts.* * *Indicates that the constructor failed while attempting to fill the puzzle*

	Half dictionary		Full dictionary	
Puzzle	Iterations	Valid words	Iterations	Valid words
1	6	2	5	2
2	8	3	7	3
3	39*	4	13	7
4	84*	0	101*	1
14	58	19	36	19
17	35*	1	96	35

In generating words, the constructor's average ratio of (words constructed)/(valid words generated) for all cases was 0.8438. This means that the basic algorithm is extremely successful in terms of generating words that did not have to be eventually deleted. Only puzzle 5 had a low ratio (0.1852), largely due to a great deal of difficulty filling one word space due to a lack of dictionary richness.

Table 3 summarizes the puzzles which were successfully generated using the 2,000 word dictionary. Figure 9 illustrates the results for the puzzle constructor.

4.2.7. Failures, misses and difficulties. It is felt that most of the failures were due to a lack of dictionary richness. This can easily be seen in the marked difference in success and efficiency between the four-letter dictionary cases of 1,000 and 2,000 words demonstrated in Table 2. One case that failed against both dictionaries is shown in Fig. 10.

M	A	R	S

FIG. 10. *Puzzle* 4

TABLE 3

Summary of word length $\leqq$ 4, 2,000 *word dictionary successes* (*where Bks = Blanks, TS = Total Squares, WS = Word Spaces, DS = Dictionary Search, Its = Iterations*)

Puzzle	Bks	Nulls	Key word length	TS	Letters / Total Sp.	Its	WS	Valid words gen.	DS	Blanks / Its	Valid words / DS	Word spaces / Valid words gen.	Its / Blanks
1	5	2	3	10	0.8000	5	2	2	2	1	1	1	1
2	6	4	3	13	0.6923	7	3	3	4	0.8571	0.7500	1	1.1667
3	13	3	4	20	0.8500	13	7	7	10	1	0.7000	1	1
5	16	6	3	25	0.7600	296	10	54	212	0.0540	0.2547	0.1852	18.5000
7	16	6	3	25	0.7600	49	10	11	37	0.3265	0.2973	0.9091	3.0620
8	17	5	3	25	0.8000	78	11	16	65	0.2178	0.2462	0.6875	4.5914
9	28	12	3	43	0.7209	36	17	17	25	0.7778	0.6800	1	1.2850
11	27	13	3	43	0.6977	34	16	16	24	0.7941	0.6667	1	1.2592
14	33	13	3	49	0.7347	36	19	19	21	0.9166	0.9047	1	1.0909
15	31	15	3	49	0.6939	46	19	20	34	0.6739	0.5882	0.9500	1.4839
17	43	18	3	64	0.7188	96	29	35	79	0.4479	0.4430	0.8286	2.2326
18	16	4	5	25	0.8400	23	13	14	19	0.6956	0.7368	0.9286	1.4380
19	52	25	4	81	0.6914	231	35	70	209	0.2251	0.3349	0.5000	4.4423
20	149	50	4	169	0.7041	1045	70	85	728	0.1426	0.1168	0.8235	7.0126
										Ave. = 0.5806	Ave. = 0.5514	Ave. = 0.8438	Ave. = 1.7224

The author does not believe that this puzzle is readily solvable without using a relatively rich dictionary. Indeed, it is difficult to find any solution to the puzzle that does not involve both symmetries and proper names. (The constructor never came close to solving puzzle 4.)

However, the constructor did generate solutions to other puzzles that were not recognized because of a lack of dictionary richness. In these cases, the constructor continued on to either find another solution or to constructor defined failure. In doing this, the constructor's work effort was often markedly extended.

It is believed that an interactive facility could markedly speed up puzzle solution. To simulate such a capability, a short "add" list was attached to the general dictionary for two of the most difficult puzzles, 5 and 20. (The final solutions for puzzles 5 and 20 are illustrated in Fig. 9.) Without this capability, the constructor initially failed on puzzle 5 after missing the words SEN, SAN and NYET. However, after placing NYET on the short list, a solution was found on iteration 296. (Puzzle 5 was particularly difficult because of the difficulty imposed by the requirement for finding two parallel words of length three beginning with the letter S.)

In constructing puzzle 20, the constructor initially failed to recognize as valid CRAP, DRAP, GAST, HOAR, ITT, MEO, NON, PLOP, GRAM, PRAT, PRE and TEE, and was terminated after 1,000 iterations with the majority of the puzzle unresolved. The words MEO and ITT were placed on the add list and the constructor produced the solution illustrated on iteration 1045. (This was one of the least efficient constructions attempted. Its inefficiencies were largely due to repeatedly constructing and rejecting as valid actually valid words.)

4.2.8. Extensions. Utilizing the constructor in its present form, there are several things that can be easily done (once computer dollars are obtained) to improve the constructor's performance: increase dictionary size, eliminate duplicate words and increase word size.

1. *Increase dictionary size.* The key to larger and more interesting puzzles is to increase the dictionary size. This requires either running on a VOS system and/or writing a paging routine. As the author is presently running on an IBM 370/155, this means writing a paging routine to hold the dictionary. The author plans to do this.

2. *Duplicate words.* The present constructor has the capability to eliminate duplicate words. This capability has been disabled because of the extremely small dictionary used (normally a 300,000 word dictionary could be considered to be available).

Once a richer dictionary is available, duplicate words will probably not be generated as often. In any case, duplicate words will not then be allowed.

3. *Increase word size.* Increased word size will allow the creation of more interesting puzzles as well as the use of a more symmetric puzzle layout. Until a larger dictionary may be used, a sufficiently rich dictionary is not attainable to enable the constructor to construct large word puzzles.

4. *Improved performance.* Once a larger dictionary may be used, it is expected that the constructor will become both significantly more efficient and be able to

solve previously unresolvable problems. A richer dictionary will be more efficient because:

(i) The constructor will not bypass valid words as often.
(ii) The occurrence ratios will be more accurate, and thus more directive to a fast solution.
(iii) The richer dictionary does not increase computational complexity as the same number of occurrence ratio tables must be generated regardless of the number of words in the table.

Therefore, the CPU time should go down markedly for the same problems (as indicated by the comparisons between the 1,000 and 2,000 word dictionaries previously discussed.)

5. Summary and conclusions.

5.1. Summary. This paper investigated methods of computer construction of crossword puzzles. The initial input to the computer is a puzzle matrix with all the intended null or blank spaces filled in. An initial key word or words is also provided to establish a beginning point for the puzzle constructor.

A dictionary format and search structure is chosen. The format selected is that of a letter table. A letter table is essentially a tree construction with the root nodes of the tree beginning either the first or last letters of the words in the letter table. For speed of search, a tableau form of the letter table was adopted.

Two different approaches to constructing the puzzles were considered. These were: filling each possible word space immediately by a whole word, and constructing words by filling the puzzle's letter spaces one by one, in a nonserial manner.

Upon investigation, it was found that the whole word entry method was not suitable because it eventually became a solution by enumeration.

The letter by letter approach was successful. It was found that usually when a word was validly formed by the letter by letter puzzle constructor, it could remain permanently in the constructed puzzle. In addition, it was found that the number of iterations per letter space was bounded by a linear function of the number of letter spaces. This is important because it indicates that the effort expended by the constructor per space to be filled does not increase in a multiplicative manner as the size of the puzzle increases.

The puzzle constructor results described herein were performed using an IBM 360/50 in a 212K partition. Solution attempts of puzzle sizes from 3×3 to 13×13 were performed. Whether or not a puzzle was solved depended on dictionary richness and initial puzzle configuration rather than upon puzzle dimensions. CPU time consumption was approximately 2,400 iterations an hour. An iteration is defined as the generation of a new stage of puzzle completeness. The average ratio of blanks to iterations was 0.5806.

5.2. Conclusions. The letter by letter approach was a reasonably successful approach to the machine solution of crossword puzzles. Obvious improvements in performance lie in the areas of greater dictionary richness and statistics of a more task specific nature. In addition, it is possible that more effective heuristics to further prune the decision tree can be developed.

REFERENCES

[1] EDWIN S. SPIEGELTHAL, *Redundancy exploitation in the computer construction of double-crostics*, Proc. EJCC, 1960, pp. 39–56.
[2] The London Traveler Magazine, (1971), pp. 1–2.
[3] H. A. BAUER, *A program to solve crossword puzzles*, Masters thesis, Northwestern Univ., Evanston, Ill., 1973.
[4] ZELLIG HARRIS, *Mathematical Structures of Language*, John Wiley, New York, 1968.
[5] G. HERDAN, *Quantitative Linguistics*, Butterworth, Belfast, 1964.
[6] GODFREY DEWEY, *Relative Frequency of English Speech Sounds*, Harvard University Press, Cambridge, 1923.
[7] LUBOMIR DOLEZEL AND RICHARD W. BAILEY, *Statistics and Style*, American Elsevier, New York, 1969.
[8] YEHOSHUA BAR-HILLELL, *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley, Reading, Mass., 1964.
[9] NOAM CHOMSKY, *Explanatory models in linguistics*, Logic, Methodology and Philosophy of Science: Proceedings of the International Congress, E. Nagel, P. Suppes and A. Tarski, eds., Stanford University Press, Stanford, Calif., 1962, pp. 528–550.
[10] A. NEWELL, J. C. SHAW AND H. SIMON, *Empirical Explorations with the Logic Theory Machine*, Proc. Western Joint Computer Conference, Vol. 15, pp. 218–239, 1957.
[11] DAVID G. HAYS, *Introduction to Computational Linguistics*, American Elsevier, New York, 1967.
[12] WALTER JACOBS, *A structure for systems that plan abstractly*, Proc. Spring Joint Computer Conference, 1971.
[13] C. E. SHANNON, *Programming a digital computer for playing chess*, Philos. Mag., 41 (1950), pp. 356–375.
[14] A. L. SAMUEL, *Some studies in machine learning using the game of checkers*, IBM J. Res. Develop., (1959), pp. 211–229.
[15] J. SLAGEL, *A heuristic program that solves symbolic integration problems in freshman calculus.* Computers and Thought, Edward A. Feigenbaum and Julian Feldman, eds., McGraw-Hill, New York, 1963.
[16] FRANK KREITH, *Heat Transfer*, International Textbook, Scranton, Pa., 1962.
[17] JOHN D. KRAUS, *Electromagnetics*, McGraw-Hill, New York, 1962.
[18] E. SANDEWALL, *Heuristic search: Concepts and methods*, Artificial Intelligence and Heuristic Programming, N.V. Findler and Bernard Meltzer, eds., American Elsevier, New York, 1971, pp. 81–100.
[19] A. NEWELL AND H. SIMON, *GPS, a program that simulates human thought*, Lernevde Autmaten, R. Oldenbourg Kg, Munich, 1961.
[20] *Webster's Elementary Dictionary*, Merriam, Springfield, Mass., 1971.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

SCHEDULING GRAPHS ON TWO PROCESSORS*

RAVI SETHI†

Abstract. Consider a directed acyclic graph (dag) D with n nodes and e edges. D represents a task system; a node corresponds to a task, and an edge (x, y) means that task x must be finished before task y can be started. We shall restrict attention to systems in which all tasks require the same processing time. Coffman and Graham give an algorithm for determining nonpreemptive schedules on two processors for such systems. The first part of their algorithm assigns unique labels to nodes. The second part uses these labels to construct a list schedule (in the sense of Graham). The time taken for each part is $O(n^2)$. We give an algorithm to determine the labeling in $O(n + e)$ steps. Similar algorithms have independently been devised to test undirected graphs for chordality. We give a second algorithm to construct a schedule from the labeling in $O(n\alpha(n) + e)$ steps. $\alpha(n)$ is an almost constant function of n.

Key words. graph labeling, breadth first search, nonpreemptive scheduling, list scheduling, set merging

1. Introduction. Coffman and Graham [2] define the following labeling for nodes in a directed acyclic graph.

LABELING L. Given a directed acyclic graph (dag) D, we shall assign a label $L(x)$ to each node x in D.

1. A *terminal* node in D is a node with no edges leaving it.[1] The label 1 is assigned to one of the terminal nodes in D.

2. Suppose labels $1, 2, \cdots, j-1$ have been assigned. Let S be the set of nodes x such that all successors of x have been assigned labels. (Note that x may have no successors.)

For each node x in S define a list $l(x)$ as follows: Let $y_1, y_2, \cdots, y_k$ be all the nodes such that (x, y_i) is an edge, for $1 \leqq i \leqq k$. Then $l(x)$ is the decreasing sequence of integers formed by ordering the set $\{L(y_1), L(y_2), \cdots, L(y_k)\}$.

Let x be an element of S such that for all x' in S, $l(x)$ is lexicographically smaller than $l(x')$. (Break ties at will.) Define the label $L(x)$ to be j. □

Lexicographic order is dictionary order, so that (5, 4, 3) is smaller than (6, 2) and (5, 4, 3, 2). Figure 1 gives an example of a dag labeled using labeling L.

In the scheduling application [2], once the dag has been labeled, a list of nodes in order of decreasing label is formed. This list is used to construct a schedule for the dag as follows:

> *List scheduling* (see [6], [7]): Whenever a processor becomes available, the list is scanned from left to right; the first unexecuted task that is ready to execute is assigned to the processor.

A list schedule for the task system in Fig. 1 is given in Fig. 4(e) (see § 3). Quite different algorithms for scheduling dags on two processors may be found in [3], [13]. See also [1], [15].

In § 2 we give an algorithm for performing labeling L in $O(n + e)$ steps for a dag D with n nodes and e edges. Section 3 gives an algorithm to determine a

* Received by the editors April 16, 1974, and in revised form February 27, 1975.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. This work was supported in part by the National Science Foundation under Grant GJ-28290.

[1] Appendix A contains graph related definitions.

schedule from the labeling in $O(n\alpha(n) + e)$ steps, where $\alpha(n)$ is an almost constant function of n. Section 4 considers the relationship of the algorithm in § 2 with other graph algorithms in the literature.

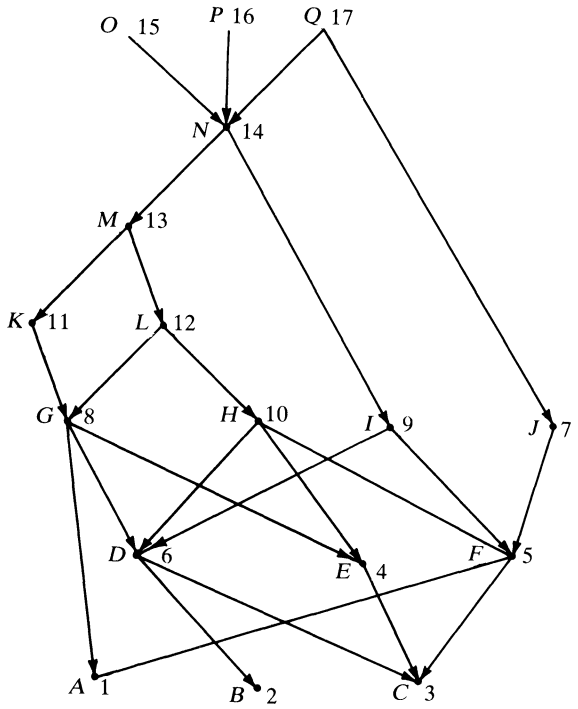


FIG. 1. *A dag labeled according to labeling L*

2. A linear labeling algorithm. A terminal node is defined to be at *level* 1. The *level* of a nonterminal node x is given by $1 + \max \{\text{level of } y | (x, y) \text{ is an edge}\}$. The dag in Fig. 1 has been drawn so that nodes at the same level appear on the same horizontal line.

Examination of the labels in Fig. 1 shows that nodes at higher levels have higher labels than nodes at lower levels. In fact, the following lemma appears as an exercise in [1].

LEMMA 1. *Let x and y be nodes in a dag D. If x is at a lower level than y, then $L(x) < L(y)$.*

Proof. Exercise. □

Given Lemma 1, we can make the following statement: Labeling L labels all nodes at level 1, then it labels all nodes at level 2 and so on. Thus if there are j nodes at levels $1, 2, \cdots, i - 1$ and k nodes at level i, then nodes at level i are assigned labels $j + 1, j + 2, \cdots, j + k$. Thus G, H, I and J at level 3 in Fig. 1 will be assigned labels 7 through 10 in some order.

All that remains to be done is to determine how nodes at a given level are to be labeled. The terminal nodes have no successors so they can be labeled in any

order. Having started the algorithm, suppose levels 1 and 2 in Fig. 1 have been labeled. Consider $S = \{G, H, I, J\}$, the set of nodes at level 3. Clearly, labels assigned to nodes in S will depend on the edges leaving these nodes.

We shall not construct the sequences $l(x)$ defined in the specification of labeling L. Instead we shall examine the destination nodes of the edges leaving nodes in S and infer the lexicographic ordering information.

Labels 1 through 6 have been assigned, with node D having the highest label, 6. Consider the edges between nodes in S and D. G, H and I have edges to D. Since $L(D) = 6$ is the highest label assigned so far, the decreasing sequences $l(G)$, $l(H)$ and $l(I)$ must all have 6 as the first element. Moreover, since there is no edge from J to D, $l(J)$ must start with a smaller integer. Clearly, $l(J)$ is lexicographically smaller than $l(G)$, $l(H)$ and $l(I)$. Hence J is assigned a smaller label than the other nodes.

Having partitioned the set S into two smaller sets, we can continue the process with the smaller sets. Consider $S' = \{G, H, I\}$. The next highest label, 5, belongs to node F, so consider the edges from nodes in S' to F. There is an edge from H and I to F, but not from G. Thus the next element in $l(H)$ and $l(I)$ is 5, but the next element in $l(G)$ is smaller. Thus G will be assigned a lower label than H and I. This process can be continued until all labels have been determined.

In order to carry out the process of partitioning the sets or equivalence classes efficiently, we will do two things: (i) Examine only the edges that play a part in the partitioning; and (ii) manage the partitioning so that the cost of keeping track of the smaller classes is low.

We will do (i) above by creating a stack that contains all edges leaving level i. Edges will be placed on the stack while the nodes at lower levels are being labeled, so all edges from nodes at level i to a node y appear closer to the top of the stack than all edges to z if $L(y) > L(z)$. For level 3 in Fig. 1, the stack might finally contain (G, A), (H, E), (G, E), (I, F), (J, F), (H, F), (G, D), (H, D), (I, D). The top of the stack is at the right.

For (ii) above, suppose the stack of edges has been created. We shall use a tree to represent equivalence classes of nodes that have to be labeled together.

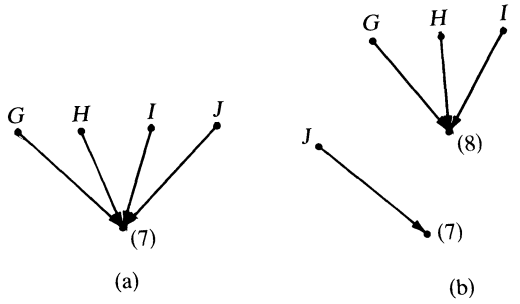


FIG. 2. *Labels are assigned level by level. Thus nodes at a level start out in the same equivalence class. The number in parentheses gives the lowest label to be assigned to the predecessors of a node. The equivalence classes are partitioned until finally labels can be assigned.*

All nodes in a given equivalence class will be *siblings* i.e., have the same immediate successor. Initially, as in Fig. 2(a) all nodes at level i have the same immediate successor. When the stack of edges is examined it turns out that there is no edge from J to node D with label 6. In order to partition the original equivalence class we introduce an auxiliary node that becomes the immediate successor of G, H and I as in Fig. 2(b). The number in parentheses at the auxiliary node keeps track of the lowest label that will eventually be assigned to the predecessors of the node.

We shall now give two routines that formalize the above discussion of the labeling process.

ROUTINE LABEL. This routine starts with a dag D and labels nodes according to labeling L. A data structure used in the process is a stack of edges leaving level i, called EDGES(i), for each level i.

1. Determine the level of each node in D.
2. Since all nodes at level 1 are terminals, EDGES(1) is empty. Since no labels have been assigned, j the lowest label to be assigned is 1. Do 3 with $i = 1, j = 1$ and EDGES(1) empty.
3. Use Routine Partition to determine the labels for nodes at level i. Do 3.1 for all nodes y at level i, in order of increasing label.
   3.1. Scan the edges entering node y. For all edges (x, y), with node x at level k, place (x, y) on top of EDGES(k).
4. Repeat 3 until all levels have been labeled. □

ROUTINE PARTITION. This routine labels nodes at a given level i. The routine has three inputs

(a) a level, i,
(b) EDGES(i), a stack of edges leaving level i,
(c) j, the lowest label to be assigned at level i.

The stack EDGES(i) has a useful property. Edges are placed on the stack while the destination nodes are being labeled. Hence all edges from level i to a node y appear closer to the top of the stack than all edges to node z if $L(y) > L(z)$.

1. Construct a new node w, such that w becomes the immediate successor of all nodes x at level i. Call w the *tree successor* of x in order to distinguish it from nodes in the dag D. Let LOW_LABEL(w) = j. Note that j is one of the inputs.
2. While EDGES(i) is nonempty do 2.1. Otherwise do 3.
   2.1. Let the edge on top of stack EDGES(i) have y as the destination node. Do 2.1.1 followed by 2.1.2 with y as the destination node.
      2.1.1. While the edge on top of EDGES(i) has destination y, do 2.1.1.1. Otherwise do 2.1.2.
         2.1.1.1. Let the edge on EDGES(i) be (x, y). Let w be the tree successor of x. Enter x into a temporary list for node w. Pop (x, y) from the stack EDGES(i). End 2.1.1.1.
      2.1.2. For all nodes w such that 2.1.1.1 places a node in the temporary list for w, construct a new auxiliary node v. Node v becomes the new tree successor of all nodes x in the temporary list for w. The edge from x to w is deleted. If there are now k nodes left with w as their tree successor, then LOW_LABEL(v) = LOW_LABEL(w) + k.

3. Visit the nodes w that have been created in steps 1 and 2, in any order. If j = LOW_LABEL(w), and there are $k \geqq 1$ nodes with w as their tree successor, then assign labels $j, j+1, \cdots, j+k-1$ in any order.
RETURN. □

We shall first demonstrate that the routines label dags correctly. Then we shall show that the labeling takes linear time.

THEOREM 1. *Routines Label and Partition assign labels according to labeling L.*

Proof. Nodes at level 1 are assigned correctly since they get the lowest labels. It is easy to see that all edges leaving level i appear on EDGES(i) when Routine Partition is called for level i. Moreover, since the edges are stacked in the order in which the destination nodes are labeled, the edges in the stack are ordered on the label of the destination node.

In order to complete the proof we need to prove the following statement:

(*) *If at some stage in the execution of step 2, nodes $x_1, x_2, \cdots, x_k$ have w as their tree successor, with j = LOW_LABEL(w), then nodes $x_1, x_2, \cdots, x_k$ are eventually assigned labels $j, j+1, \cdots, j+k-1$, in some order. Moreover, if $\sigma(x) = \{y \mid x, y)$ is an edge$\}$, and S is the set of destination nodes for which step 2.1 has been executed, then $S \cap \sigma(x_1) = S \cap \sigma(x_2) = \cdots = S \cap \sigma(x_k)$.*

It is easy to prove (*) by induction on the number of executions of step 2.1. □

Now we shall show that the two routines take time linear in the number of edges in the dag. We assume that the dag has been specified by giving a list of edges, or lists of immediate predecessors or successors of each node, as in [16].

THEOREM 2. *Routines Label and Partition take $O(n+e)$ steps for a dag with n nodes and e edges.*

Proof. The level of a node can be determined in linear time by visiting the nodes in any order that visits successors before predecessors. The terminals are at level 1. During the visit if edge (x, y) is examined, then set the level of x to 1 + level of y, unless x has already been placed at a higher level. The other steps in Routine Label clearly take linear time.

The interesting step in Routine Partition is 2.1.2 because it introduces auxiliary nodes and edges. An auxiliary node is introduced only when there is an edge from a node at level i to a node at a lower level. When an auxiliary node is visited the cost of the visit can be charged to the edge that caused the visit. It can be shown that each edge is charged a constant number of times. Thus execution of the two routines takes $O(n+e)$ steps. The n appears in $O(n+e)$ since a label is assigned to each node. □

3. Constructing a schedule. Once the dag has been labeled, the next step is to construct a schedule from the labeling. As Fig. 3 shows, list scheduling as considered in § 1 may sometimes be inefficient. Consider the list of nodes from which a schedule for the dag in Fig. 3 is constructed. If the list is scanned from left to right in order to find the first ready node, then successive scans will have to skip over $n-1$, $n-2, \cdots, 1$ nodes, respectively. The construction of the schedule will therefore take $O(n^2)$ steps.

In the sequel we shall use the terms *task* and *task system* interchangeably

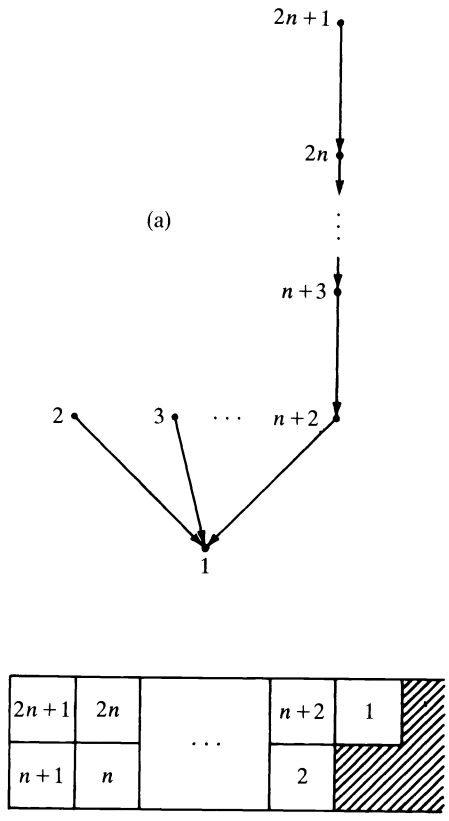


FIG. 3. (a) *A labeled task system.* (b) *A schedule on two processors constructed from the labeling.*

with node and dag. We shall see that it is possible to construct a list schedule in almost linear time. The algorithm exploits the following two facts. (i) Each task has an execution time of one unit. (ii) In the lists we shall deal with predecessors appear before their successors. Given the list $(T_n, T_{n-1}, \cdots, T_1)$, all the predecessors of task T_i appear before T_i in the list.

Intuitively, the algorithm builds up a schedule by successively constructing schedules for the lists (T_n), (T_n, T_{n-1}), $\cdots$ until the schedule for the entire list $(T_n, T_{n-1}, \cdots, T_1)$ is constructed.

Consider the task system in Fig. 1. Schedules corresponding to some partial lists for the task system have been shown in Fig. 4. When the lists (17) and (17, 16) are considered we get the schedules in Fig. 4(a) and (b), with tasks 17 and 16 on processors 1 and 2. The next list to be considered is (17, 16, 15). Task 15 is ready to begin execution during the first time unit, but there is no free processor. Thus task 15 is assigned in the next time unit. As the next lemma shows, there is a simple algorithm for constructing the schedule for an augmented list.

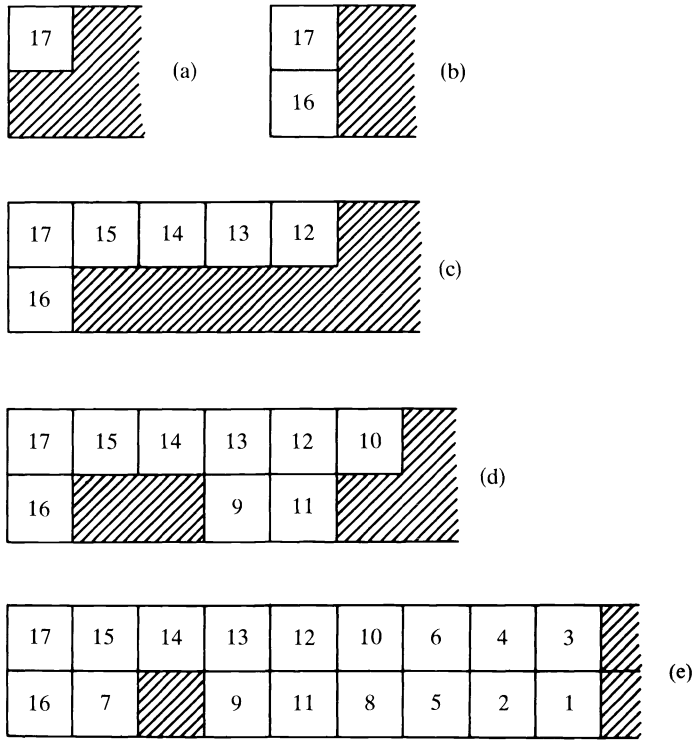


FIG. 4. *The schedule for the task system in Fig.* 1 *is constructed by successively considering the lists* (17), (17, 16), $\cdots$, (17, 16, $\cdots$, 1).

LEMMA 2. *Let* S_{i+1} *be the list schedule on* m *processors for* $(T_n, T_{n-1}, \cdots, T_{i+1})$. *The list schedule* S_i *for* $(T_n, T_{n-1}, \cdots, T_i)$ *can be constructed from* S_{i+1} *as follows: Let* t *be the first time unit by which all predecessors of* T_i *have been executed. Assign* T_i *to a processor during the first time unit* u, $u \geqq t$ *at which a processor is available.*

Proof. For all time units before t, the statement of the lemma assures us that T_i is not ready to be executed. So even if it is reached in a scan of the list $(T_n, T_{n-1}, \cdots, T_i)$ it cannot be assigned. Thus until time unit t, the two schedules will be the same. After time unit t, in constructing schedule S_i, task T_i will be reached when there is a processor free and no other task can be assigned. Clearly, this situation corresponds to the first idle time after t in schedule S_{i+1}. The lemma must therefore be true. □

From Lemma 2 it seems that we can trade repeated scans of the list for scans to determine a time unit during which a processor is free. It turns out that by using an efficient algorithm for merging disjoint sets we can cut down on the rechecking of time units during which all processors are busy.

Suppose all processors are busy during time unit i and will remain busy until j. Then for all t, $i \leqq t \leqq j$, the first time unit at which a processor is available is j. We collect all such t into a set identified by j. In general we choose names of sets so that if t is in set u, then u is the first time unit at or after t at which a processor

is available. In order to construct schedules as outlined in Lemma 2, we therefore need to be able to determine the set that a time unit is in, and also the first time unit at which a task becomes ready.

ROUTINE SCHEDULE. Let D be a dag labeled according to labeling L. This routine constructs a list schedule for the task system represented by D using the list of tasks in order of decreasing label.

By way of data structures we need a set for each time unit. Initially SET(t) $= \{t\}$ for each time unit.

1. All initial tasks in D are said to be *ready* at time unit 1.
2. For all tasks x in D, do step 3 in order of decreasing label.
3. Consider task x. Let x be ready at time unit t. If t is in SET(u), then assign x to a processor during time unit u. If u now has no free processors, look at time unit $u + 1$. Let $u + 1$ be in SET(v). Merge SET(u) and SET(v) calling the new set SET(v).
   Examine all y such that y is an immediate successor of x. If all predecessors of y have now been assigned, then task y is said to become *ready* at $u + 1$. □

In order to show that Routine Schedule is correct, we shall first show that the set operations are performed correctly.

LEMMA 3. *Let time unit t be in* SET(u) *at some stage. Then u is the first time unit, $u \geqq t$ at which a processor is available.*

Proof. Initially, time unit t is in SET(t), and all processors are free. Suppose that for some u, all processors have just been assigned tasks during u. Both u and $u + 1$ then have the same unit as the first free time unit. If $u + 1$ is in SET(v), the algorithm then merges SET(u) and SET(v), and calls the new set SET(v). The reader is invited to construct a more rigorous proof based on the number of executions of step 3 of the routine. □

THEOREM 3. *Routine Schedule constructs a list schedule using the list of tasks in order of decreasing label.*

Proof. Since tasks appear in order of decreasing label, when task x is considered, all predecessors of x have already been assigned. Thus the time unit at which x becomes ready is known. From Lemma 3 the algorithm correctly locates the next free time unit. Since schedules are augmented as in Lemma 2, the theorem must be true. □

We shall avoid the details of how the set operations are performed. The reader is referred to [9], [17]. We do note that step 3 of the routine looks at each edge in the task system exactly once. And, the number of set operations is proportional to n, the number of tasks. It has been shown in [17] that n set operations can be done in time almost linear in n. More precisely, n operations can be done in $O(n\alpha(n))$ time, where $\alpha(n)$ is a functional inverse of Ackermann's function.

4. Discussion. In many applications the directed graphs that occur have relatively few edges. An adjacency matrix requires space $O(n^2)$ and hence at least $O(n^2)$ time is required if any algorithm uses an adjacency matrix as a data structure. A list of edges, or a list of predecessors or successors for each node is proportional in length to the number of edges. Such a data structure is used in a number of graph algorithms [8], [10], [11], [16], [18].

The algorithm in § 2 examines the directed graph representing a task system. While similar in principle to the linear graph marking algorithms in [8], [16], [18] it differs in the important respect that the traversal of the graph depends on the labels assigned. The searches in [8], [16], [18] are such that the property of the graph that is being determined is independent of the order in which the nodes are searched. The level of a node is such a property.

The level of each node being independent of the search makes it possible to construct a linear labeling algorithm. During the labeling, it is possible to collect all edges leaving a certain level; and have them ordered on the destination node. A more direct solution is frustrated by the fact that an edge leaving level j may be scanned before the nodes at level $j - 1$ have been labeled.

Similar algorithms have been independently devised [12], [14] to test if an undirected graph is chordal. A *chordal* graph has been defined in [4], [5] as an undirected graph in which every simple cycle $v_1, v_2, \cdots, v_n, v_1$, for $n > 3$ has an edge between two nonconsecutive vertices, i.e., there is an edge between v_i and v_j, $j \neq i + 1$. As shown in [4] polynomial time algorithms exist for problems like the minimal coloring of a chordal graph.

It has been shown in [14] that a lexicographic ordering policy very similar to labeling L leads to a simple test for chordality. The ideas in this paper can be adapted to perform their lexicographic ordering.

Given the utility of lexicographic orderings in two quite different applications, a natural approach for further study is an investigation of the structure imposed by lexicographic orderings on directed and undirected graphs.

An interesting type of breadth first search called *structured breadth first search* (SBFS) is introduced in [12]. SBFS uses a mechanism similar to the partitioning illustrated in Fig. 2 to limit the number of possible breadth first searches of a graph. SBFS is applied in [12] to test an undirected graph for chordality.

Appendix A. A *directed graph* is a pair (N, E), where N is a set of *nodes* and E is a set of pairs of nodes called *edges*. An edge (x, y), is said to be *from* (*leave*) x *to* (*enter*) y. x is called the *origin* node and y the *destination* node of the edge. y is an *immediate successor* of x, and x is an *immediate predecessor* of y.

A sequence of edges $S = (x_1, x_2)\ (x_2, x_3) \cdots (x_{k-1}, x_k)$ is called a *path of length k, from* x_1 *to* x_k. If S is a path from x to y, then x is a *predecessor* to y and y is a *successor* of x. A node with no predecessors is an *initial* node, a node with no successors is a *terminal* node. A path from a node x to x is called a *cycle*. A directed acyclip graph (dag), is a directed graph with no cycles. A nonterminal node x is at *level k* if a longest path from x to a terminal node is of length k. The *level* of a terminal node is defined to be 1.

REFERENCES

[1] E. G. Coffman, Jr. and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] E. G. Coffman, Jr. and R. L. Graham, *Optimal scheduling for two processor systems*, Acta Informatica, 1 (1972), pp. 200–213.

[3] M. Fujii, T. Kasami and K. Ninomiya, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789, Erratum, 20 (1971), p. 141.

[4] F. GAVRIL, *Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph*, this Journal, 1 (1972), pp. 180–187.
[5] ———, *The intersection graphs of subtrees in trees are exactly the chordal graphs*, J. Combinatorial Theory, 16 (1974), pp. 47–56.
[6] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.
[7] ———, *Bounds for certain multiprocessing anomalies*, Bell System Tech. J., 45 (1966), pp. 1563–1581.
[8] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.
[9] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, this Journal, 2 (1973), pp. 294–303.
[10] J. E. HOPCROFT AND J. K. WONG, *Linear time algorithm for isomorphism of planar graphs*, 6th Annual ACM Symp. on Theory of Computing, Seattle (May 74), pp. 172–184.
[11] T. KAMEDA AND I. MUNRO, *A $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs*, Computing, 12 (1974), pp. 91–98.
[12] G. S. LUEKER, *Structured breadth first search and chordal graphs*, Tech. Rep 158, Princeton Univ., Princeton, N.J., 1974.
[13] Y. MURAOKA, *Parallelism, exposure and exploitation in programs*, Doctoral thesis, Univ. of Illinois, Urbana, 1971.
[14] D. J. ROSE AND R. E. TARJAN, *Algorithmic aspects of vertex elimination on graphs*, Memorandum, Univ. of California, Berkeley, 1974.
[15] R. SETHI, *Algorithms for Minimal Length Schedules*, Computer and Job-Shop Scheduling Theory, E. G. Coffman, ed., John Wiley, New York, 1975 to appear.
[16] R. E. TARJAN, *Depth first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.
[17] ———, *Efficiency of a good but not linear set union algorithm*, Memorandum ERL-M434, Univ. of California, Berkeley, 1974.
[18] L. E. THORELLI, *Marking algorithms*, BIT, 12 (1972), pp. 555–568.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

NEW BOUNDS ON THE COMPLEXITY OF THE SHORTEST PATH PROBLEM*

MICHAEL L. FREDMAN†

Abstract. It is shown that $O(N^{5/2})$ comparisons and additions suffice to solve the all-pairs shortest path problem for directed graphs on N vertices with nonnegative edge weights. In conjunction with preprocessing, this result is exploited to produce an $o(N^3)$ algorithm for solving the shortest path problem.

Key words. graph, shortest path, complexity, sorting, decision tree

Introduction. In this paper, we present two new upper bounds on the complexity of the shortest path problem. Let G be a complete directed graph on N vertices whose edges are assigned nonnegative weights. Between all pairs of vertices (v_i, v_j), $i \neq j$, there exists a directed path of minimum weighted length, the value of which we denote by L_{ij}. The computation of the $N(N-1)$ values L_{ij}, $1 \leqq i \neq j \leqq N$, is referred to as the all-pairs shortest path problem. The computation of the values L_{1j}, $1 < j \leqq N$, is referred to as the single source shortest path problem. These problems and variations of them have been investigated by a number of authors. We are concerned here with the all-pairs problem with nonnegative weights. At least two essentially different approaches to the problem have been developed, in particular that of Dijkstra [1] and that of Floyd [2]. Dijkstra's method solves N separate single source problems, while Floyd's method is matrix oriented. All known methods have worst-case running times that are $O(N^3)$ on the random access machine under the uniform cost criterion (memory accesses, additions, branching instructions, etc., each require unit time). A variant of Dijkstra's approach developed by Spira [3] has an expected running time of $O(N^2(\log N)^2)$. Regarding worst-case complexity, under the assumption that the only permissible operations are $a + b$ and $\min(a, b)$ in a straight line computation, it has been shown by Kerr [5] that the $O(N^3)$ bound is best possible. If we allow comparisons between sums of edge costs (decision tree complexity), Spira and Pan [4] have shown that cN^2 comparisons are necessary in the worst case to solve the single source problem, and this certainly applies to the all-pairs problem. In this paper, we show that $O(N^{5/2})$ comparisons between sums of edge costs suffice to solve the all-pairs problem. However, our method does not seem to readily lend itself to a practical algorithm with a $O(N^{5/2})$ running time, although in principle for some $c > 0$, we can construct for each N a separate algorithm A_N whose running time is $< cN^{5/2}$. We can however, mildly exploit this method as a preprocessing technique and construct a single algorithm with running time $o(N^3)$ for all values of N. More precisely, we describe an $O(N^3 (\log\log N)^{1/3}/(\log N)^{1/3})$ algorithm.

These results are primarily of theoretical interest, particularly when contrasted with the results of Spira and Pan, and Kerr quoted above.

* Received by editors January 3, 1975, and in revised form March 17, 1975.

† Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This work was supported in part by the Office of Naval Research under Contract N0014-67-A-0204-0067.

1. Decision tree complexity. In this section, we show that $O(N^{5/2})$ comparisons and additions suffice to solve the shortest path problem. First we show that $O(N^{5/2}(\log N)^{1/2})$ comparisons suffice, using a method which yields an explicit comparison tree. Secondly, we refine this result and prove the existence of an $O(N^{5/2})$ comparison method which we do not explicitly exhibit.

Our method is matrix oriented. We use the theorems of Munro [6], Furman [7] and Fischer and Meyer [8], which imply (explicitly stated in [12]) that the complexity of the shortest path problem is of the same order of magnitude as the complexity of min/plus multiplication of $N \times N$ matrices. (Given $A = (a_{ij})$, $B = (b_{ij})$, the min/plus product $C = AB$ is defined by $C = (c_{ij})$ where $c_{ij} = \min \{a_{ik} + b_{kj}, 1 \leqq k \leqq N\}$.)

THEOREM 1. *For some $c > 0$, $cN^{5/2}(\log N)^{1/2}$ comparisons and additions suffice to solve the all-pairs shortest path problem for directed graphs with non-negative weighted edges.*

Proof. As discussed above, it suffices to show that $O(N^{5/2}(\log N)^{1/2})$ comparisons and additions are sufficient to compute the min/plus product of two $N \times N$ matrices. Let $A = (a_{ij})$ and $B = (b_{ij})$. As will be seen below, all comparisons are performed between a sum or difference of two items in the set $\{a_{ij}, b_{ij}; 1 \leqq i, j \leqq N\}$. Consequently, we need only to count comparisons to prove the theorem.

We begin by partitioning A into $N \times m$ submatrices and B into $m \times N$ submatrices (see Fig. 1), where m will be specified later.

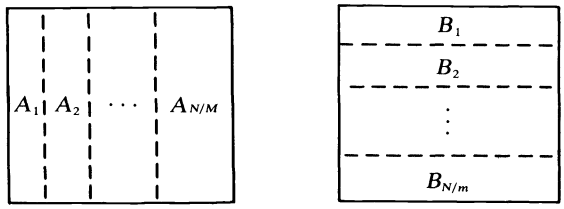


FIG. 1

We have $AB = \min(A_1B_1, A_2B_2, \cdots, A_{N/m}B_{N/m})$ where the min operation is a componentwise minimization. Once the N/m matrices A_jB_j are computed, each of dimension $N \times N$, this min operation can be computed with $N^2(N/m) = N^3/m$ comparisons. Next we show how to compute the product A_1B_1 with $O(m^2N \log N)$ comparisons. By computing the products A_jB_j, $j > 1$, in the same manner, we will conclude that a total of $O(mN^2 \log N)$ comparisons suffice to compute all the products A_jB_j.

To compute A_1B_1, for each pair of indices r, s, $1 \leqq r < s \leqq m$, we sort the $2N$ differences $a_{ir} - a_{is}$, $1 \leqq i \leqq N$, and $b_{sj} - b_{rj}$, $1 \leqq j \leqq N$, with $2N \log_2 N + O(N)$ comparisons. Doing this for each of the pairs r and s requires a total of $O(m^2N \log N)$ comparisons. Having done this, we symbolically form the product A_1B_1 without further involvement with the values of the a_{ij}, b_{ij} quantities. (Letting $(c_{ij}) = A_1B_1$, for each i and j, we can determine a number $t = t(i, j)$ such that $c_{ij} = a_{it} + b_{tj}$.) This follows because

$$(1) \qquad a_{ir} + b_{rj} \leqq a_{is} + b_{sj} \quad \text{if and only if} \quad a_{ir} - a_{is} \leqq b_{sj} - b_{rj}.$$

Since all inequalities on the right of (1) have been determined, those on the left can be deduced, and therefore the numbers t such that $c_{ij} = a_{it} + b_{tj}$ can be deduced. This can all be done symbolically once the above sorting has taken place.

We conclude that AB can be computed with $O(N^3/m + mN^2 \log N)$ comparisons. Setting $m = N^{1/2}/(\log N)^{1/2}$ yields our theorem.

In the remainder of this section, we show how to refine the method of Theorem 1 to obtain a $O(N^{5/2})$ bound. Referring to the proof of Theorem 1, we show that the product A_1B_1 can be computed with $O(m^2N)$ comparisons, as opposed to $O(m^2N \log N)$ comparisons. With this improvement, we can choose $m = N^{1/2}$ to obtain the $O(N^{5/2})$ bound.

When computing A_1B_1 in the proof of Theorem 1, we sort each of the sets, $D_{rs} = \{a_{ir} - a_{is}, b_{sj} - b_{rj},\ 1 \leqq i \leqq N,\ 1 \leqq j \leqq N\}$, for $1 \leqq r < s \leqq m$. Let L_{rs} denote the list of $2N$ items that results when D_{rs} is sorted, and let L denote the concatenation of all these lists,

$$L = L_{12}L_{13} \cdots L_{1m}L_{23} \cdots L_{2m} \cdots L_{m-1m}.$$

Each list L_{rs} is clearly restricted to be one of $(2N)!$ arrangements when written symbolically, and so L is restricted to be one of $(2N)!^{\binom{m}{2}} = O(c^{m^2N \log N})$ (for some c) arrangements. The following lemma, however, shows that the number of realizable arrangements is in fact much smaller.

LEMMA A. *Let Γ denote the set of possible arrangements that can result when forming the list L. Then*

$$\log |\Gamma| = O(mN \log N).$$

Proof. A well-known combinatorial theorem (see Buck [9]) states that n hyperplanes partition k-dimensional space into at most

$$\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{k} \leqq (k+1)n^k \tag{2}$$

regions. In the $k = 2mN$-dimensional space whose coordinates are the $2mN$ components of the matrices A_1 and B_1, the regions determined by the hyperplanes defined by the equations

$$(S)\quad \begin{aligned} &a_{ir} - a_{is} = a_{i'r} - a_{i's}, && 1 \leqq i < i' \leqq N,\quad 1 \leqq r < s \leqq m \\ &b_{si} - b_{ri} = b_{si'} - b_{ri'}, && 1 \leqq i < i' \leqq N,\quad 1 \leqq r < s \leqq m, \\ &a_{ir} - a_{is} = b_{sj} - b_{rj}, && 1 \leqq i < N,\quad 1 \leqq j \leqq N,\quad 1 \leqq r < s \leqq m, \end{aligned}$$

correspond to the possible arrangements of the list L. The number of planes here is

$$n = \binom{m}{2}\binom{2N}{2} = O(m^2N^2).$$

Substituting in (2) with these values for n and k and taking logarithms yields the lemma.

We are now ready to prove the $O(N^{5/2})$ bound. The author's search theorem

discussed in [10] and [11] implies that the list L can be formed with at most

$$\log_2 |\Gamma| + 2|L| \tag{3}$$

comparisons, where $|L| = 2\binom{m}{2}N$ denotes the length of L. Lemma A implies that when $m = N^{1/2}$, the quantity in (3) is dominated by the second term, and therefore $O(m^2N)$ comparisons suffice to form L. But we have seen that the determination of L is tantamount to computing A_1B_1. As described in the discussion preceding Lemma A, this is what we require to conclude the following.

THEOREM 2. *For some $c > 0$, $cN^{5/2}$ comparisons and additions suffice to solve the all-pairs shortest path problem for directed graphs with nonnegative weighted edges.*

2. Preprocessing as a means for constructing $o(N^3)$ algorithms. Along the lines of the last section, it is possible to construct a branching algorithm or comparison tree which operates on inputs consisting of a pair of matrices, one $N \times m$ and the other $m \times N$, and symbolically outputs the $N \times N$ product in time $O(m^2N)$. ($m = N^{1/2}$.) Assuming that we have constructed such a tree, we can utilize it to multiply matrices of much larger dimension. Let A^* and B^* be M by M matrices with M much larger than N. By regarding A^* and B^* as partitioned into $N \times N$ submatrices, we can form their product by performing $(M/N)^3$ products on $N \times N$ submatrices. Using our tree, each submatrix product is done in time $O(N^{5/2})$ (assuming we choose $m = N^{1/2}$), and so the entire multiplication A^*B^* is performed in time

$$O(M^3/N^{1/2}). \tag{4}$$

To obtain $o(M^3)$ performance, we would want to allow N to grow with M, and incorporate into an algorithm as preprocessing a mechanism to build the comparison tree which would be used for the multiplication of $N \times N$ submatrices. We want to choose N as large as possible, but constrained so that the preprocessing time doesn't dominate the overall timing. In Theorem 2, we chose m to grow as $N^{1/2}$ in order to optimize our result. In the context of this discussion, it is conceivable that by choosing m in a different manner, while the time to multiply $N \times N$ submatrices may increase, this is more than compensated for by a substantial decrease in preprocessing time. In fact this doesn't happen, and the $m = N^{1/2}$ choice is still near-optimal.

The time to build an explicit comparison tree is at least as large as the number of leaves it contains. Let us assume for the moment that the comparison tree implicit in Theorem 2 for multiplying $N \times m$ with $m \times N$ matrices can be constructed in time which is polynomial in the size of the tree, the latter represented by the quantity $|\Gamma|$ of Lemma A. Then the preprocessing time for our proposed

algorithm would be less than

$$c^{N^{3/2}\log_2 N} \quad \text{for some } c \geqq 2, \tag{5}$$

and choosing $N = (1/\log_2 c)(\log_2 M)^{2/3}/(\log_2 \log_2 M)^{2/3}$, (5) becomes less than M, and (4) becomes $O(M^3 (\log \log M)^{1/3}/(\log M)^{1/3})$.

To complete our description of this $O(M^3(\log \log M)^{1/3}/(\log M)^{1/3})$ algorithm, we have to verify that we can build, within the time given by (5), a comparison tree that multiplies $N \times N^{1/2}$ matrices with $N^{1/2} \times N$ matrices in time $O(N^2)$, as described in the discussion leading up to Theorem 2. We show how to tabulate all of the lists L enumerated in Lemma A within the above time constraint. Once these lists are tabulated, the construction described in [10] can be performed in time which is polynomial in the number of lists. Note that with each leaf of our tree we associate the $N \times N$ symbolic matrix product, which is precomputed using a conventional algorithm.

To tabulate the lists L, we proceed as follows. In the context of Lemma A, we form a collection P of pairs of matrices A_1, B_1 of respective dimension $N \times m$ and $m \times N, m = N^{1/2}$. For each of the regions discussed in the proof of Lemma A, the collection P will contain at least one pair of matrices A_1, B_1 in the interior of the given region. The size of P will be $\leqq c^{N^{3/2}\log N}$. Using this collection, the lists L can be tabulated in the following manner. Using conventional sorting techniques, given a pair in P, we form the associated list L, symbolically as well as numerically. If some of the quantities within a set D_{rs} (see previous section) are equal, we eliminate the resulting list since it would not have arisen from a pair in P which lies in the interior of a region defined by the system (S), and there would be ambiguity in forming the list L which is troublesome in the context of forming a comparison tree. Having constructed the symbolic lists L for each remaining pair in P, we next remove duplicates. This is easily accomplished by sorting the collection of lists according to a lexicographic ordering. This completes the tabulation process once the set P has been constructed. Given P, the time required to perform the above processing is:

(A) To convert a particular pair in P to a list L can be done in time $O(N^2 \log N)$. For all pairs, the time is $O(|P|N^2 \log N)$.

(B) Having formed the lists described in A, to sort them according to a lexicographic ordering requires time $O(N^2|P| \log |P|)$ since $|P| \log |P|$ comparisons between lists need to be performed, each comparison performed in time $O(|L|) = O(N^2)$.

As we will see, $|P| \leqq C^{N^{3/2}\log N}$, so that times given in (A) and (B) are

$$\leqq C^{N^{3/2}\log N + O(\log N)} \leqq C_1^{N^{3/2}\log N}.$$

Now we describe a method to form the set P. From the nature of the equations defining the hyperplanes (S) of Lemma A, each region formed by these planes contains points with nonnegative coordinates satisfying the following condition.

(C) All of the quantities in any one of the sets D_{rs} differ from one another in absolute value by at least 1.

Our set P will contain one such point from every region. Consider the following

collection of planes.

$$
(S')\qquad
\begin{aligned}
&a_{ir} - a_{is} + 1 = a_{i'r} - a_{i's},\\
&\quad a_{ir} - a_{is} = a_{i'r} - a_{i's} + 1,\\
&b_{si} - b_{ri} + 1 = b_{si'} - b_{ri'},\\
&\quad b_{si} - b_{ri} = b_{si'} - b_{ri'} + 1, \qquad 1 \leqq i < i' \leqq N, \quad 1 \leqq r < s \leqq m,\\
&a_{ir} - a_{is} + 1 = b_{sj} - b_{rj},\\
&\quad a_{ir} - a_{is} = b_{sj} - b_{rj} + 1,\\
&\qquad\qquad 1 \leqq i \leqq N, \quad 1 \leqq j \leqq N, \quad 1 \leqq r < s \leqq m,\\
&a_{ij} = 0, \qquad 1 \leqq i \leqq N, \quad 1 \leqq j \leqq m,\\
&b_{ij} = 0, \qquad 1 \leqq i \leqq m, \quad 1 \leqq j \leqq N.
\end{aligned}
$$

Clearly for each region R defined by the planes of (S), there is a region R' defined by the planes in (S') such that $R' \subseteq R$, and all points in R' have nonnegative coordinates and satisfy condition (C). The nonnegative requirement ensures that this region will have boundary vertex points. The set P will contain these boundary vertex points. We systematically attempt to solve every subset of $2mN$ equations of (S'). Since the total number of equations in (S') is $O(m^2N^2)$, this task can be accomplished in time $\leqq C^{mN \log N} = C^{N^{3/2} \log N}$. We include in P all the points we discover that are unique solutions to some particular subset of $2mN$ equations.

This concludes in outline form a method for computing the min/plus product of $M \times M$ matrices in time $O(M_3(\log\log M)^{1/3}/(\log M)^{1/3})$. As mentioned previously, this lends itself to a shortest path algorithm with an equivalent running time.

Acknowledgment. The author wishes to acknowledge A. Schönhage. An unpublished result of his on the min/plus convolution of two sequences inspired the method of Theorem 1.

REFERENCES

[1] E. W. DIJKSTRA, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[2] R. W. FLOYD, *Algorithm* 97: *Shortest path*, Comm. ACM, 5 (1962), p. 345.

[3] P. M. SPIRA, *A new algorithm for finding all shortest paths in a graph of positive arcs in average time* $O(n^2 \log^2 n)$, this Journal, 2 (1973), pp. 28–32.

[4] P. M. SPIRA AND A. PAN, *On finding and updating shortest paths and spanning trees*, Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 82–84.

[5] L. R. KERR, *The effect of algebraic structure on the computational complexity of matrix multiplications*, Ph.D. thesis, Cornell University, Ithaca, N.Y.,1970.

[6] I. MUNRO, *Efficient determination of the transitive closure of a directed graph*, Information Processing Letters, 1 (1971), pp. 56–58.

[7] M. E. FURMAN, *Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph*, Dokl. Akad. Nauk SSSR 194, 524 (in Russian), English translation in Soviet Dokl. 11:5, 1252.

[8] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 129–131.

[9] R. C. BUCK, *Partition of space*, Amer. Math. Monthly, 50 (1943), pp. 541–544.
[10] M. L. FREDMAN, *How good is the information theory bound in sorting?*, J. Theoretical Computer Science, to appear.
[11] ———, *Two applications of a probabilistic search technique; Sorting X + Y and building balanced search trees*, Proc. 7th SIGACT Conference.
[12] A. AHO, J. HOPCRAFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974, pp. 204–206.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

ON ALGORITHMS FOR ENUMERATING ALL CIRCUITS OF A GRAPH*

PRABHAKER MATETI† AND NARSINGH DEO‡

Abstract. A brief description and comparison of all known algorithms for enumerating all circuits of a graph is provided, and upper bounds on computation time of many algorithms are derived. The vector space method of circuit enumeration is discussed. It is proved that K_3, K_4, $K_4 - x$ and $K_{3,3}$ are the only undirected and reduced graphs which do not have any edge-disjoint unions of circuits.

Key words. algorithms, graph theory, circuits, dicircuits, cycles, circuit vector space, circuit-graph, adjacency matrix, graph search, backtrack

1. Introduction. Given an undirected graph, the problem of enumerating all its circuits has received much attention lately. The analogous problem for a directed graph is to enumerate all its directed circuits. The two problems have much in common, and we consider them in parallel.

In this paper, we survey all known algorithms for circuit enumeration and compare their efficiencies. In § 2, we classify and describe the basic ideas behind these algorithms. It is shown, in § 3, that only four graphs exist that have circuit vector spaces consisting of all circuits. The algorithms are compared in § 4, and conclusions are presented. To avoid misunderstanding, we define our terminology.

A *graph* G is a triple $\langle V, E, f \rangle$ where V is a finite set of *vertices*, E a finite set of *edges*, and f is a function. A graph is either *directed* (a *digraph*) when $f : E \to V \times V$, or is *undirected* when $f : E \to \{\{u, v\} | u, v \in V\}$. Observe that this definition permits the graph to have parallel edges (two edges e_1 and e_2 are *parallel* if $e_1 \neq e_2$ and $f(e_1) = f(e_2)$) and self-loops (an edge e is a *self-loop* if for some vertex $v, f(e) = (v, v)$ for digraph or $f(e) = \{v\}$ for undirected graph). A graph G is *simple* if it has neither self-loops nor parallel edges. An edge e of a digraph is said to be *incident out* of vertex v_1, and *incident into* a vertex v_2 if $f(e) = (v_1, v_2)$. The edge e is *incident with* both vertices v_1 and v_2. The *in-degree* of a vertex v of a digraph is the number of edges incident into v, and the *out-degree* of v is the number of edges incident out of v. An edge e of an undirected graph is *incident with* vertices v_1 and v_2 if $f(e) = \{v_1, v_2\}$. The *degree* of a vertex v in an undirected graph is the number of edges incident with this vertex, self-loops being counted twice.

Two vertices v_1 and v_2 of a directed graph G are *adjacent* if there is some edge e so that $f(e) = (v_1, v_2)$ or (v_2, v_1). An *edge-sequence* from vertex v_0 to v_k in a digraph is an alternating sequence of vertices and edges, $v_0 e_1 v_1 e_2 v_2 \cdots v_{k-1} e_k v_k$, such that for $1 \leqq i \leqq k$, $f(e_i) = (v_{i-1}, v_i)$. This edge sequence is said to *pass through* the vertices v_i, $0 < i < k$. An edge sequence is *simple* if it does not pass through a vertex more than once. A *dipath* from v_0 to v_k is an edge-sequence from v_0 to v_k where all vertices are distinct, except possibly that $v_0 = v_k$. When $v_0 = v_k$,

---

* Received by the editors August 13, 1973, and in final revised form April 2, 1975. This work was supported in part by the National Science Foundation under Grant GJ-31222.

† Department of Computer Science, University of Illinois, Urbana, Illinois 61801.

‡ Department of Electrical Engineering, Indian Institute of Technology, Kanpur, India. Now at Computer Science Department, Washington State University, Pullman, Washington, 99163.

the dipath is called a *rooted dicircuit* rooted at v_0. The rooted dicircuit considered as a subgraph of G is called a *dicircuit*. The *length* of an edge-sequence is the number of edges in it. By *deleting an edge* e we obtain a new graph $G' = \langle V, E', f' \rangle$ where $E' = E - \{e\}$ and f' is a restriction of f to E'. By *deleting a vertex* v from a graph G we obtain a new graph $G' = \langle V', E', f' \rangle$, where $V' = V - \{v\}$ and $E' = \{e \in E | f(e) \neq (v, u) \text{ or } (u, v) \text{ for } u \in V\}$ and $f' : E' \to V' \times V'$ is a restriction of f to E'. By *ignoring* the direction of an edge e we mean that the ordered pair $f(e) = (u, v)$ be considered as a set $\{u, v\}$. The corresponding definitions for an undirected graph follow if the directions of the edges are ignored.

An undirected graph is *connected* if there exists a path between every pair of vertices. A vertex is a *cut-vertex* if it lies on every path between a pair of vertices. A connected undirected graph is *separable* if it has at least one cut-vertex. A maximal, connected subgraph is called a *component* of the graph. A maximal, nonseparable subgraph of a graph is called a *block* of the graph. A directed graph is *strongly-connected* if there exists a dipath between every pair of vertices. A maximal, strongly-connected subgraph of a digraph is called a *fragment*. The *undirected version* G of a digraph D is obtained by ignoring the direction of the edges of D. The *directed version* $D = \langle V, E_D, f_D \rangle$ of an undirected graph $G = \langle V, E, f \rangle$ is obtained by introducing two edges e' and e'' into E_D such that $f_D(e') = (u, v)$ and $f_D(e'') = (v, u)$ iff $e \in E$ and $f(e) = \{u, v\}$. A *spanning tree* of a connected undirected graph is a maximal subgraph which has no circuits. The unique circuit obtained by adding a nontree edge to the spanning tree is called a *fundamental circuit* (with respect to that spanning tree). The *ring-sum* of two circuits $C_1 = \langle V_1, E_1, f_1 \rangle$ and $C_2 = \langle V_2, E_2, f_2 \rangle$ of an undirected graph G is an undirected graph $S = C_1 \oplus C_2 = \langle V', E', f' \rangle$ where $E' = E_1 \cup E_2 - E_1 \cap E_2$, and for all $e \in E'$, $f'(e) = f(e)$, and $V' = \{u, v \in V | \text{ for some } e \in E', f'(e) = \{u, v\}\}$. A *variable adjacency matrix* A of a digraph $D = \langle V, E, f \rangle$ is an $n \times n$ symbolic matrix in which the (i, j)-element $A_{ij} = \sum_{\text{all } e_k} e_k$, where $e_k \in E$ such that $f(e_k) = (v_i, v_j)$. If there is no such e_k, then $A_{ij} = 0$. The powers of A, $p \geqq 1$, are defined as follows: $A^1 \equiv A$, and for $p > 1$, $A^p \equiv A^{p-1} \cdot A = A \cdot A^{p-1}$, employing the usual symbolic multiplication. To obtain a binary adjacency matrix, replace all nonzero entries A_{ij} by 1's.

We shall use n to denote the number of vertices, e the number of edges, and c the number of circuits of the given graph G. We shall refer to an edge-sequence of length k as a k-sequence. Similarly we use k-dipath and k-dicircuit, etc.

2. Outline of circuit enumeration algorithms. In principle, any algorithm enumerating all dicircuits of a digraph can be used to enumerate all circuits of an undirected graph, and vice versa. (However, see § 4 for computational complexity considerations.) The algorithms are therefore considered together. Every circuit enumeration algorithm proposed can be put into one of the following four classes, depending on the underlying approach:

1. circuit vector space for undirected graphs;
2. search algorithms;
3. powers of adjacency matrix;
4. edge-digraph.

In the following subsections, we attempt to describe the basic ideas behind various algorithms belonging to each class. More details may be found in Mateti and Deo [19].

2.1. Circuit vector space algorithms. It is well known that the set of all circuits and edge-disjoint unions of circuits of an undirected graph is a vector space over $GF(2)$, with the vector addition corresponding to the ring-sum operation on subgraphs. To obtain all circuits, one can start from a basis for the circuit vector space. The dimension of this space is $\mu = e - n + 1$, for a connected graph of n vertices and e edges. All the $2^\mu - \mu - 1$ vectors (excluding the basic circuits themselves, and the null element) are computed. Since not every vector is a circuit, a test is necessary to determine if the vector generated is a circuit. Generally only a small fraction of the $2^\mu - \mu - 1$ vectors are circuits, the rest being edge-disjoint unions of circuits. In fact, we will prove in § 3 that there are only four undirected graphs having $2^\mu - 1$ circuits, and we will exhibit classes of graphs for which the ratio of number of circuits to the number of vectors goes asymptotically to zero as the number of vertices increases.

An attempt to compute only a subset of all vectors and yet enumerate all circuits was made by Welch [36], and later by Hsu and Honkanen [15]. Gibbs [13] showed that Welch's attempt led to incorrect results, and Mateti and Deo [19] give a class of graphs for which Hsu and Honkanen's algorithms end up computing all $2^\mu - \mu - 1$ vectors.

Algorithms belonging to the vector space class are: Maxwell and Reed [20], Welch [36], Gibbs [13], Rao and Murti [26], Hsu and Honkanen [15], Mateti and Deo [19].

2.2. Search algorithms. These algorithms search for circuits in an appropriate search space which is a super set containing all circuits. The efficiency of such an algorithm depends on (i) the size of this super set, (ii) the effort it takes to compute an element in the search space, and (iii) a test to find if the element is indeed a circuit. The algorithms of Char [6], and Chan and Chang [5] have the set of all permutations of vertices of the graph as the search space.

Several algorithms since Floyd [12] have used backtracking to generate dipaths of the graphs, and then identified if it is a dicircuit. The algorithm of Tarjan [32] uses improved pruning methods to decrease the size of the subset of dipaths generated considerably. Further improvements in this algorithm are made by Johnson [16], Read and Tarjan [28] and Szwarcfiter and Lauer [31].

The algorithms of Roberts and Flores [29], Floyd [12], Tiernan [33], Berztiss [2], [3], Weinblatt [34], and Ehrenfeucht et al. [11] are also basically backtrack algorithms.

2.3. Algorithms using powers of adjacency matrix. Let A be the variable adjacency matrix of the given digraph. Then a nonzero element A^p_{ij}, $p \geqq 1$, is the set of all p-sequences from vertex v_i to vertex v_j. Since no dicircuit can be of length greater than n, we need compute only up to A^n. Observe that an edge-sequence need neither be a dipath nor a dicircuit. The crux of the problem is to avoid such

nonsimple edge-sequences. It is in this avoidance that different algorithms in this class vary. Note that all algorithms using this method find rooted dicircuits.

Algorithms belonging to this class are: Ponstein [25], Yau [37], Kamae [17], Danielson [9], Ardon and Malik [1].

2.4. Algorithm using edge-digraph of a digraph. Consider a digraph $E(D)$ derived from the given digraph D as follows: the vertex-set of $E(D)$ is the edge-set of D, and $E(D)$ contains an edge q between its vertices e_1 and e_2 iff e_1 and e_2 (which are edges of D) are adjacent in D.

The edge-digraph of a p-dipath is a $(p-1)$-dipath; the edge-digraph of a p-dicircuit is also a p-dicircuit. Thus we see that there is a one-to-one correspondence between the dicircuits of D and $E(D)$. An edge of $E(D)$ will correspond to a rooted 2-dicircuit. Enumerate and delete all 2-dicircuits of $E(D)$, and call the resulting graph D^1. Now, take the edge-digraph of D^1. $E(D^1)$ has only dicircuits of length $p \geqq 3$. Enumerate and delete all dicircuits of length 3; call the resulting digraph D^2, and so on until a D^p, for some p, is empty.

An algorithm using this approach is due to Cartwright and Gleason [4].

3. Circuit vector space of undirected graph. All circuit vector space algorithms, known so far, compute all of the 2^μ vectors, in the worst case. But there are classes of graphs for which the ratio of number of circuits c to vectors approaches zero for large μ. In fact, there are only four graphs with $c = 2^\mu - 1$. These four graphs are shown in Fig. 1. To prove this, it is useful to have the following definition.

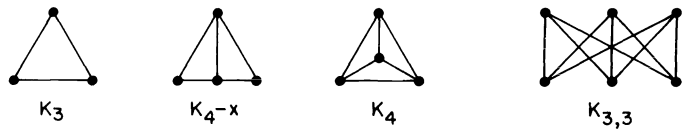


FIG. 1. *Graphs all of whose vectors are circuits*

DEFINITION 1. A graph G is said to be *reduced* if

(i) G is simple,

(ii) G has no vertex of degree 0 or 1, and,

(iii) for every vertex v of degree 2, the two vertices adjacent to v are themselves adjacent.

LEMMA. *Let G be a reduced graph with the number of vertices $n \geqq 6$. If G has no edge-disjoint union of circuits, then the number of edges $e \geqq n + 3$. The converse is not true.*

Proof. Since G is reduced, and has no edge disjoint union of circuits, G must be connected. No vertex in G can be of degree less than 3; for otherwise, G is either not reduced or has an edge-disjoint union of circuits. Therefore $e \geqq 3n/2 = n + n/2 \geqq n + 3$, since $n \geqq 6$.

The graphs in Fig. 2 have $e = n + 3$ and yet have edge-disjoint unions of circuits.

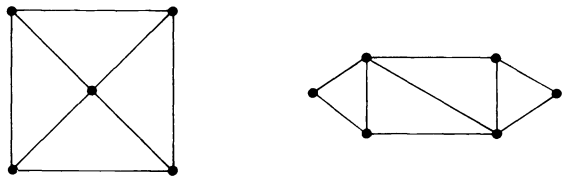


FIG. 2. *Graphs with $e = n + 3$ and yet having edge-disjoint unions of circuits*

THEOREM 1. *Only the four reduced graphs K_3, K_4, $K_4 - x$ and $K_{3,3}$ have all vectors as circuits.*

Proof. That the theorem is true for graphs with at most five vertices and seven edges can be seen by enumerating reduced graphs. It is also true for graphs with n vertices $n \geqq 6$, and e edges, $e \leqq n + 2$ by the lemma. For graphs with $n \geqq 5$, and $e \geqq n + 3$, the theorem will be proved by contradiction. We will first assume that there exists a graph G which has $n \geqq 5$, is not $K_{3,3}$, and yet has all vectors as circuits, and then show a contradiction. If G is disconnected or separable, we are done because every reduced graph which is separable or disconnected will contain one or more edge-disjoint union of circuits. Therefore G is connected and is non-separable. Graph G is either planar or nonplanar.

Case 1. G is planar, $n \geqq 5$, $e \geqq n + 3$; that is, nullity $\mu = e - n + 1$ is at least 4.

There are at least 4 finite regions and one infinite region defined by a plane representation of the graph G (see Deo [10, Chap. 5]). If any two finite regions are not adjacent, the circuits defining these regions are edge-disjoint. Likewise, every finite region must also be adjacent to the infinite region; otherwise the circuit defining the finite region is disjoint with the circuit defining the infinite region. The dual of G is therefore a complete graph $K_{\mu+1}$ of $\mu + 1$ vertices, and $\mu + 1 \geqq 5$ which is impossible. Thus there must exist a pair of regions which are not adjacent, and hence an edge-disjoint union of circuits.

Case 2. G is nonplanar, and therefore G either has a subgraph homeomorphic to $K_{3,3}$, or has a subgraph homeomorphic to K_5.

Case 2.1. Let G have a subgraph homeomorphic to K_5. An edge-disjoint union of circuits of K_5 shown in Fig. 3 is homeomorphic to a subgraph of G.

Case 2.2. G has a proper subgraph g homeomorphic to $K_{3,3}$. Since g is a proper subgraph, there exists an additional path between two vertices u and v. If u and v correspond to vertices in the same independent set of vertices of $K_{3,3}$,

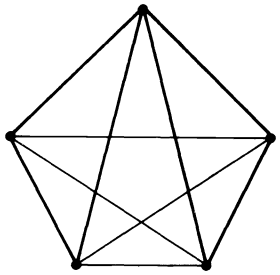


FIG. 3. *An edge-disjoint union of circuits in a complete graph of five vertices*

there exists an edge-disjoint union of circuits (see Figure 4(a)). If u and v correspond to vertex u' in one independent set of vertices of $K_{3,3}$ and to v' in the other independent set, respectively, an edge-disjoint union of circuits exists (see Figure 4(b)).

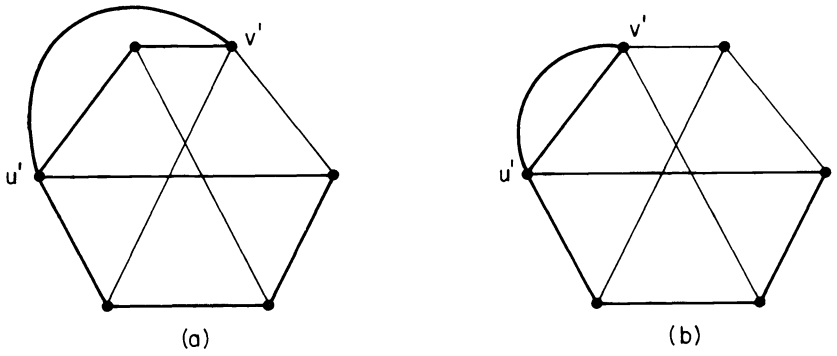
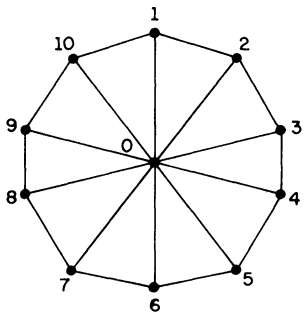


FIG. 4. *Edge-disjoint unions of circuits in a nonplanar graph having a proper subgraph homeomorphic to* $K_{3,3}$

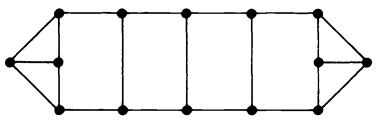
Observe that the Theorem 1 does not give us information as to how many edge-disjoint unions of circuits a given graph can have. In general, a large fraction of the vectors are edge-disjoint unions of circuits. The ratio of circuits to all vectors in the vector space for numerous classes of graphs tends to zero. Three such classes are shown in Fig. 5.



$\mu = n - 1$

$c = \mu(\mu - 1) + 1$

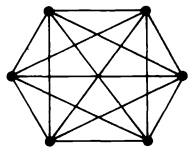
(a) *A wheel graph*



$\mu = \frac{1}{2}(n + 2)$

$c = \frac{\mu^2}{2} + \frac{5\mu}{2} - 3$

(b) *A modified ladder graph*



$$\mu = \frac{n^2}{2} - \frac{3n}{2} + 1$$

$$c = \frac{1}{2}\sum_{s=3}^{n} \binom{n}{s} (s-1)!$$

(c) *A complete graph*

FIG. 5. *Graphs for which* $\lim_{n\to\infty} (c/2^{\mu}) \to 0$

4. Analyses of algorithms. We take as our data unit a single vertex, or an edge. Other data objects like paths, circuits, or graphs are composed of these basic data units. The space bounds for various algorithms are given in terms of these data units. Any operation performed on an edge, between two edges, or between two vertices takes a unit of time. Thus adding an edge to a partly constructed path, testing if the label of a vertex is greater than that of another vertex, given e finding $f(e)$, etc., all take a unit of time. All other operations performed in the actual execution of an algorithm are assumed to take no time. The time bounds of the algorithms are given in terms of these time units.

In what follows we give summaries of analyses performed on the algorithms. Space bounds are rather obvious and are given in Table 1 at the end of this section. Since the number of time units consumed by some of these algorithms depends intricately on the structure of the graph whose circuits are being enumerated, the worst-case time bounds given are in general pessimistic bounds. Thus if T_A and T_B are upper bounds on the time required by two algorithms A and B, respectively, and if $T_A < T_B$, it cannot be said with certainty that algorithm A is faster than B. The algorithms A and B are often directly compared with each other to make a statement to the effect that A is faster than B on any given graph. In general, the worst-case graphs for two algorithms are not the same. For a discussion on related matters, see Chase [8].

It will be computationally advantageous to perform some initial simplifications on the given graph which do not disturb the circuit structure of the graph before proceeding to enumerate all circuits. Self-loops can be enumerated initially. Then if the given graph is a directed graph, take its fragments, take the undirected version of each fragment and decompose it into blocks, and enumerate the dicircuits of the subdigraph corresponding to each of these blocks. A set of p parallel edges can be replaced by a single new edge. For each dicircuit containing this new edge, enumerate p dicircuits, each containing an edge from the set of edges replaced. Vertices v of in-degree and out-degree 1 may be deleted by adding an edge connecting the two adjacent vertices of v. Note, however, that this might result in parallel edges. Similar editing operations may be performed on undirected graphs. This is advantageous because all these editing operations have upper bounds of $O(n + e)$ (see Hopcroft and Tarjan [14]). And even the fastest known algorithm (Johnson's) has an upper time bound[1] of $O((n + e)(c + 1))$. For the purpose of time analysis, we will therefore assume that the graph given is an "edited" graph.

[1] For some positive constant K, and given function $\Psi(\cdot)$, $O(\Psi(\cdot))$ means that the number being represented by $O(\Psi(\cdot))$ is at most $K \cdot \Psi(\cdot)$; $T = O(\Psi(\cdot))$ means that $T \leqq K \cdot \Psi(\cdot)$; and $\Psi(\cdot) = O(T)$ means that $T \geqq K \cdot \Psi(\cdot)$.

If the size of the input is a and the size of the output is b, then any upper bound cannot be less than $O(a + b)$. We insist that each circuit be enumerated as a sequence of vertices or edges. Thus $b = O(n \cdot c)$. Johnson's algorithm approaches this bound.

It may be pointed out that an algorithm A enumerating all the dicircuits of a digraph may be used on an undirected graph G by taking the directed version D of G. The digraph D is obtained from G by replacing each edge e of G by two oppositely directed edges e' and e''. For each circuit of G, we now have two dicircuits in D, and, in addition, e 2-dicircuits are created, where e is the number of edges in G. Thus if c_D is the number of dicircuits in D, and c_G is the number of circuits in G, $c_D = 2c_G + e$. (However, an algorithm for undirected graphs, such as a circuit vector space algorithm, should not be applied on an undirected version G of a digraph D to enumerate the dicircuits because the number of circuits in G could be exponential in the number of dicircuits of D.) Thus circuit vector space algorithms should not be used even to enumerate the circuits of an undirected graph until progress has been made in the pruning methods used to avoid fruitless computations of noncircuit vectors giving a circuit vector space algorithm having an upper time bound of $O((n + e)c)$. When this is achieved, the multiplying constants, which depend on the implementation, may make one algorithm better to use than the other. Such a progress appears promising if it is remembered that the backtrack algorithms for digraphs have been improved from exponential (Tiernan [33]) to polynomial in output (Johnson [16]) in three years.

TABLE 1
Upper bounds on time and space

Algorithm of	Time bound	Space bound	Method used
Ardon and Malik	$n(\text{const.})^n$	n^2	Powers of adjacency matrix
Berztiss	$n(\text{const.})^n$	$n + e$	Backtrack
Cartwright and Gleason	$n(\text{const.})^n$	$n(\text{const.})^n$	Edge-digraph
Chan and Chang	$\sum_{i=1}^{n-1} i!i$	$n + e$	Searches permutations
Char	$\sum_{i=3}^{n} n!/(n-i)!$	$n + e$	Searches permutations
Danielson	$n(\text{const.})^n$	$n(\text{const.})^n$	Powers of adjacency matrix
Ehrenfeucht *et al.*	$n^3 + n \cdot e \cdot c$	$n + e$	Backtrack
Hsu and Honkanen	$n \cdot 2^{2\mu}$	$e \cdot 2^{\mu}$	Circuit vector space
Johnson	$(n + e)c$	$n + e$	Backtrack
Kamae	—	$n(\text{const.})^n$	Powers of adjacency matrix
Mateti and Deo	$\mu^2 \cdot 2^{\mu}$	μ^2	Circuit vector space
Maxwell and Reed	$n \cdot 2^{2\mu}$	$e \cdot 2^{\mu}$	Circuit vector space
Ponstein	—	$n(\text{const.})^n$	Powers of adjacency matrix
Rao and Murti	$e \cdot \mu^2 \cdot 2^{\mu}$	$\mu \cdot n$	Circuit vector space
Read and Tarjan	$(n + e)c$	$n + e$	Backtrack
Szwarcfiter and Lauer	$(n + e)c$	$n + e$	Backtrack
Tarjan	$n \cdot e \cdot c$	$n + e$	Backtrack
Tiernan (RFFT)	$n(\text{const.})^n$	$n + e$	Backtrack
Weinblatt	$n(\text{const.})^n$	$n \cdot c$	Backtrack
Welch-Gibbs	$n \cdot 2^{2\mu}$	$e \cdot 2^{\mu}$	Circuit vector space
Yau	—	$n(\text{const.})^n$	Powers of adjacency matrix

5. Conclusions. Of all the algorithms analyzed, the Johnson's algorithm [16] having an upper bound of $O((n + e)c)$ on the time is the asymptotically fastest algorithm. (Szwarcfiter and Lauer [31] give a similar algorithm with the same upper bound.) The success of this backtrack algorithm is due to a very effective pruning technique which avoids much of the fruitless search present in earlier algorithms.

Little has been done in circuit vector space algorithms by way of pruning unnecessary computations. The algorithms of Hsu and Honkanen [15], and Mateti and Deo [19] are the fastest among circuit vector space algorithms. It cannot be said of these two algorithms that one is faster than the other.

Acknowledgments. The authors wish to thank the referees for making valuable suggestions. The authors are grateful to Professor Jurg Nievergelt for many insightful discussions.

REFERENCES

[1] M. T. Ardon and N. R. Malik, *A recursive algorithm for generating circuits and related subgraphs*, 5th Asilomar Conf. on Circuits and Systems 5, Pacific Grove, Calif., Nov. 1971, pp. 279–284.

[2] A. T. Berztiss, *Data Structures: Theory and Practice*, Academic Press, New York, 1971.

[3] ———, *A K-tree algorithm for simple cycles of a directed graph*, Tech. Rep. 73-6, Dept. of Computer Sci., Univ. of Pittsburgh, 1973.

[4] D. Cartwright and T. C. Gleason, *The number of paths and cycles in a digraph*, Psychometrika, 31 (1966), pp. 179–199.

[5] S. G. Chan and W. T. Chang, *On the determination of dicircuits and dipaths*, Proc. 2nd Hawaii Internat. Conf. System Sci., Honolulu, Hawaii, 1969, pp. 155–158.

[6] J. P. Char, *Master circuit matrix*, Proc. IEE (London), 115 (1968), pp. 762–770.

[7] ———, *Master circuit matrix*, Ibid., 117 (1970), pp. 1655–1656.

[8] S. M. Chase, *Analysis of algorithms for finding all spanning trees of a graph*, Ph.D. thesis, Rep. 401, Dept. of Computer Sci., Univ. of Illinois, Urbana, Ill., 1970.

[9] G. H. Danielson, *On finding the simple paths and circuits in a graph*, IEEE Trans. Circuit Theory, CT-15 (1968), pp. 294–295.

[10] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[11] A. Ehrenfeucht, L. D. Fosdick and L. J. Osterweil, *An algorithm for finding the elementary circuits of a directed graph*, Tech. Rep. CU-CS-024-73, Dept. of Computer Sci., Univ. of Colorado, Boulder, 1973.

[12] R. W. Floyd, *Nondeterministic algorithms*, J. Assoc. Comput. Mach., 14 (1967), pp. 636–644.

[13] N. E. Gibbs, *A cycle generation algorithm for finite undirected linear graphs*, Ibid., 16 (1969), pp. 564-568.

[14] J. Hopcroft and R. Tarjan, *Efficient algorithms for graph manipulation*, Comm. ACM, 16 (1973), pp. 372–378.

[15] H. T. Hsu and P. A. Honkanen, *A fast minimal storage algorithm for determining all the elementary cycles of a graph*, Computer Sci. Dept., Pennsylvania State Univ., University Park, 1972.

[16] D. B. Johnson, *Finding all the elementary circuits of a directed graph*, this Journal, 4 (1975), pp. 77–84.

[17] T. Kamae, *A systematic method of finding all directed circuits and enumerating all directed paths*, IEEE Trans. Circuit Theory, CT-14 2 (1967), pp. 166–171.

[18] J. W. LaPatra and B. R. Meyers, *Algorithms for circuit enumeration*, IEEE Internat. Conv. Record, part 1, 12 (1964), pp. 368–373.

[19] P. Mateti and N. Deo, *On algorithms for enumerating all circuits of a graph*, UIUCDCD-R-73-585 (revised), Dept. of Computer Sci., University of Illinois, Urbana, Ill., 1973.

[20] L. M. MAXWELL AND G. B. REED, *Subgraph identification—Segs, circuits and paths*, 8th Midwest Symp. on Circuit Theory, Colorado State Univ., Fort Collins, Colo., June 1965, pp. 13-0–13-10.

[21] R. L. NORMAN, *A matrix method for location of cycles of a directed graph*, Amer. Inst. Chem. Engrs. J., 11 (1965), pp. 450–452.

[22] K. PATON, *An algorithm for finding a fundamental set of cycles for an undirected linear graph*, Comm. ACM, 12 (1969), pp. 514–518.

[23] ———, *An algorithm for the blocks and cut-nodes of a graph*, Ibid., 14 (1971), pp. 468–476.

[24] M. PRABHAKER, *Analysis of algorithms for finding all circuits of a graph*, Master's tech. thesis, Dept. of Electrical Engrg., Indian Inst. of Tech., Kanpur, India, 1972.

[25] J. PONSTEIN, *Self-avoiding paths and adjacency matrix of a graph*, SIAM J. Appl. Math., 14 (1966), pp. 600–609.

[26] V. V. B. RAO AND V. G. K. MURTI, *Enumeration of all circuits of a graph*, Proc. IEEE, 57 (1969), pp. 700–701.

[27] V. V. B. RAO, K. SANKARA RAO, R. SANKARAN AND V. G. K. MURTI, *Planar graphs and circuits* Matrix Tensor Quart., 18 (1968), pp. 88–91.

[28] R. C. READ AND R. E. TARJAN, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, *Networks* (to appear); ERL Memo. M 433, Electronics Research Laboratory, Univ. of California, Berkeley, 1973.

[29] S. M. ROBERTS AND B. FLORES, *Systematic generation of Hamiltonian circuits*, Comm. ACM, 9 (1966), pp. 690–694.

[30] M. M. SYSLO, *The elementary circuits of a graph*, *Algorithm 459*, Comm. ACM, 16 (1973), pp. 632–633; Errata: Ibid., 18 (1975), pp. 119.

[31] J. L. SZWARCFITER AND P. E. LAUER, *Finding the elementary cycles of a directed graph in $O(n + m)$ per cycle*, no. 60, Univ. of Newcastle upon Tyne, Newcastle upon Tyne, England, May 1974.

[32] R. TARJAN, *Enumeration of the elementary circuits of a directed graph*, this Journal, 2 (1973), pp. 211–216.

[33] J. C. TIERNAN, *An efficient search algorithm to find the elementary circuits of a graph*, Comm. ACM, 13 (1970), pp. 722–726.

[34] H. WEINBLATT, *A new search algorithm for finding the simple cycles of a finite directed graph*, J. Assoc. Comput. Mach., 19 (1972), pp. 43–56.

[35] J. T. WELCH, *Cycle algorithms for undirected linear graphs and some immediate applications*, Proc. 1965 ACM Nat. Conf., P-65, pp. 296–301.

[36] ———, *A mechanical analysis of the cyclic structure of undirected linear graphs*, J. Assoc. Comput. Mach., 13 (1966), pp. 205–210.

[37] S. S. YAU, *Generation of all Hamiltonian circuits, paths, and centers of a graph, and related problems*, IEEE Trans. Circuit Theory, CT-14 (1967), pp. 79–81.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

ON THE EVALUATION OF POWERS*

ANDREW CHI-CHIH YAO†

Abstract. It is shown that for any set of positive integers $\{n_1, n_2, \cdots, n_p\}$, there exists a procedure which computes $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ for any input x in less than $\lg N + c\sum_{i=1}^{p} [\lg n_i/\lg\lg(n_i + 2)]$ multiplications for some constant c, where $N = \max_i \{n_i\}$. This gives a partial solution to an open problem in Knuth [3, § 4.6.3, Ex. 32] and generalizes Brauer's theorem on addition chains.

Key words. addition chains, Brauer's theorem

1. Introduction. An *addition chain* (of length r) is a sequence of $r + 1$ integers $a_0, a_1, a_2, \cdots, a_r$ such that (i) $a_0 = 1$ and (ii) for each i, $a_i = a_j + a_k$ for some $j \leqq k < i$. It is clear that, for any r and any set of integers $\{n_1, n_2, \cdots, n_p\}$, there exists an addition chain of length r which contains the values $n_1, n_2, \cdots, n_p$ if and only if there exists a procedure which, for any input x, computes $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ in r operations using only multiplications. A theorem by Brauer [1], [3, pp. 398–418] states that, for any n, there exists an addition chain of length[1] $\lg n + O(\lg n/\lg\lg n)$ which contains the value n; this implies the existence of a corresponding procedure to compute x^n in $\lg n + O(\lg n/\lg\lg n)$ multiplications. Furthermore, it was shown by Erdös [2], [3, pp. 398–418] that the above result is asymptotically with probability 1 nearly the best possible. In an open problem posed in Knuth [3, § 4.6.3, Ex. 32], it is asked if there are fast procedures to compute $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ for $p \geqq 2$. This problem cannot be solved by a direct extension of the technique used by Brauer in the proof of his theorem.

In this paper we show that for any positive integers $n_1, n_2, \cdots, n_p$, there exists a procedure using only multiplications which, for any input x, computes $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ in $\lg N$ + constant $\times \sum_{i=1}^{p} [\lg n_i/\lg\lg(n_i + 2)]$ multiplications where $N = \max_i \{n_i\}$. This gives a solution to Knuth's problem and leads to a corresponding theorem on addition chains which generalizes Brauer's theorem mentioned earlier.

2. Definition. Let e_i, $1 \leqq i \leqq p$, and f_j, $1 \leqq j \leqq q$, be positive integers. We shall say that $\{x^{e_1}, \cdots, x^{e_p}\}$ *is computable from* $\{x^{f_1}, \cdots, x^{f_q}\}$ *in r multiplications* ($r \geqq 0$) if there exists a set of r positive integers, $\{f_{q+1}, \cdots, f_{q+r}\}$, such that

(i) for all $i = q + 1, \cdots, q + r$,

$$x^{f_i} = x^{f_j} \cdot x^{f_k} \quad \text{for some } j \leqq k < i.$$

(ii) $\{x^{e_1}, \cdots, x^{e_p}\} \subset \{x^{f_1}, \cdots, x^{f_{q+r}}\}$.

---

* Received by the editors August 29, 1974.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This research was supported by the National Science Foundation under Grant GJ-41538.

[1] lg is logarithm to the base 2.

Since the exponents are added when two powers of x are multiplied, the above definition is a natural generalization of the definition of addition chains (cf. § 1). The exponents appearing in $\{x^{f_1}, \cdots, x^{f_q}\}$ correspond to a set of numbers initially available in the chain, as opposed to a single number, 1, in the earlier definition.

3. The computation of $\{x^{n_1}, \cdots, x^{n_p}\}$. The following lemma is well known [3, pp. 398–418).

LEMMA 1. *For any integer $i > 0$, $\{y^i\}$ is computable from $\{y\}$ in at most $2\lfloor \lg i \rfloor$ multiplications.*

Proof. Let the binary representation of i be

$$i = \sum_{j=0}^{v} b_j \cdot 2^j, \tag{1}$$

where $v = \lfloor \lg i \rfloor$. Then,

$$y^i = \prod_{b_j = 1} y^{2^j}. \tag{2}$$

Thus, we can first compute $y^2, y^4, y^8, \cdots, y^{2^v}$ sequentially in v multiplications and then compute y^i by (2) in no more than v multiplications. The total number of multiplications is no greater than $2v$. □

THEOREM 2. *For any integers m, n where $0 < m \leqq n$, $\{x^m\}$ is computable from $\{x, x^2, x^4, x^8, \cdots, x^{2^{\lfloor \lg n \rfloor}}\}$ in less than $c \lg n / \lg \lg (n + 2)$ multiplications for some constant c.*

Proof. Assume $n \geqq 4$. Define the following quantities:

$$k = \lceil (\lg \lg n)/2 \rceil, \tag{3}$$

$$D = 2^k, \tag{4}$$

$$t = \lfloor \log_D n \rfloor, \tag{5}$$

Let the D-ary representation of m be

$$m = \sum_{j=0}^{t} a_j D^j,$$

where

$$0 \leqq a_j \leqq D - 1 \quad \text{for } j = 0, 1, \cdots, t. \tag{6}$$

We partition the set of integers $\{0, 1, \cdots, t\}$ into D disjoint subsets $S(0), S(1), \cdots, S(D-1)$ by letting

$$S(i) = \{l | a_l = i\} \quad \text{for } i = 0, 1, \cdots, D - 1.$$

It follows from (6) that

$$m = \sum_{i=1}^{D-1} i \cdot \left[\sum_{l \in S(i)} D^l \right] = \sum_{i=1}^{D-1} i \cdot m_i, \tag{7}$$

where

$$m_i = \sum_{l \in S(i)} D^l. \tag{8}$$

From (7) and (8), we obtain the following two equations:

$$x^{m_i} = \prod_{l \in S(i)} x^{D^l} \quad \text{for } i = 1, 2, \cdots, D-1, \tag{9}$$

$$x^m = \prod_{i=1}^{D-1} (x^{m_i})^i. \tag{10}$$

Since all the x^{D^l} in (9) are available in the set $\{x, x^2, x^4, x^8, \cdots, x^{2^{\lfloor \lg n \rfloor}}\}$, we can construct a procedure to compute x^m as follows.

Step 1. For $i = 1, 2, \cdots, D-1$ do the following:

(a) Compute x^{m_i} from (9) in fewer than $|S(i)|$ multiplications.

(b) Compute $(x^{m_i})^i$ in at most $2\lfloor \lg i \rfloor$ multiplications (by Lemma 1).

Step 2. Compute x^m from (10) in $D-2$ multiplications.

Let M be the total number of multiplications in the above procedure. Then,

$$\begin{aligned} M &< \sum_{i=1}^{D-1} (|S(i)| + 2\lfloor \lg i \rfloor) + D - 2 \\ &\leqq \sum_{i=1}^{D-1} |S(i)| + 2(D-1)\lg(D-1) + D - 2. \end{aligned} \tag{11}$$

Noting that the $S(i)$'s form a partition of the set $\{0, 1, \cdots, t\}$, we obtain from (11) that

$$M < t - 1 + 2(D-1)\lg(D-1) + D - 2, \tag{12}$$

which together with equations (3), (4) and (5), implies that

$$M < 2(\lg n/\lg\lg n) + 1 + 4(\lg n)^{1/2} \lg\lg n + 2(\lg n)^{1/2}. \tag{13}$$

It follows from (13) that there exists a constant c such that

$$M < c \lg n/\lg\lg(n+2). \tag{14}$$

Thus the theorem is true if $n \geqq 4$. Obviously we can choose c so that the theorem is also true for $n = 1, 2, 3$. □

THEOREM 3. *For any set of positive integers* $\{n_1, n_2, \cdots, n_p\}$, $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ *is computable from input* $\{x\}$ *in less than* $\lg N + c\sum_{i=1}^{p} [\lg n_i/\lg\lg(n_i+2)]$ *multiplications for some constant* c, *where* $N = \max_i \{n_i\}$.

COROLLARY. $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ *is computable from* $\{x\}$ *in less than* $\lg N + cp \lg N/\lg\lg(N+2)$ *multiplications.*

Proof of Theorem 3 *and Corollary.* First we compute $\{x, x^2, x^4, x^8, \cdots, x^{2^{\lfloor \lg N \rfloor}}\}$ from input x in $\lfloor \lg N \rfloor$ multiplications. For each i, according to Theorem 2, x^{n_i} is computable from $\{x, x^2, x^4, \cdots, x^{2^{\lfloor \lg N \rfloor}}\}$ in $c \lg N/\lg\lg(N+2)$ multiplications for some constant c. The theorem and corollary then follow immediately. □

In terms of addition chains, Theorem 3 and its corollary give the following generalization of Brauer's theorem [1], [3, pp. 398–418].

THEOREM 4. *For any positive integers* $n_1, n_2, \cdots, n_p$, *there exists an addition chain of length less than* $\lg N + c\sum_{i=1}^{p} \lg n_i/\lg\lg(n_i+2)$ *containing the values* $n_1, n_2, \cdots, n_p$ *for some constant* c, *where* $N = \max_i \{n_i\}$.

COROLLARY. *For positive integers* $n_1, n_2, \cdots, n_p$, *there exists an addition chain of length less than* $\lg N + cp \lg N/\lg\lg(N+2)$ *containing* $n_1, n_2, \cdots, n_p$.

4. Conclusion. We have shown that $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ can be computed in $\lg N + cp \lg N/\lg\lg(N+2)$ multiplications for input x where $N = \max_i \{n_i\}$ and c is a constant. On the other hand, it is well known that to evaluate $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ by arithmetic operations, at least $\lg N$ operations are necessary. Thus our procedures for evaluating $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ are nearly the best possible when $p \ll \lg\lg(N+2)$. It remains an interesting open problem to determine the complexity of computing $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ for general p.

Note added in proof. (A) By choosing the value of k in (3) more carefully, say $k = \lceil \lg\lg n - 3\lg\lg\lg n \rceil$, our algorithm in Theorem 3 takes at most $\lg N + p \lg N/\lg\lg N +$ (smaller terms) multiplications as $N \to \infty$. For fixed p, these leading terms are almost the best possible since, as observed by Larry Stockmeyer (private communication), the lower bound of Erdös [2] can be generalized straightforwardly. (B) Nicholas Pippenger proved the following (private communication): $\{x^{n_1}, x^{n_2}, \cdots, x^{n_p}\}$ can be computed from x in $\min\{(p + 2^l)\lceil \lg N/l \rceil \mid l \text{ is a positive integer}\}$ multiplications, and for some $c_1 > 0$ and every N, p, $c_1 p \lg N/(\lg P + \lg\lg N)$ multiplications are needed for some set of $\{n_1, n_2, \cdots, n_p\}$ with $\max\{n_i\} \leqq N$. For large p ($p \geqq \lg N$), this determines the worst-case complexity to be $p \lg N/\lg p$ up to a constant factor. (C) A related theorem on power evaluation may be found in Schönhage [4].

REFERENCES

[1] A. Brauer, *On addition chains*, Bull. Amer. Math. Soc., 45 (1939), pp. 736–739.
[2] P. Erdös, *Remarks on number theory—On addition chains*, Acta Arith., 6 (1960), pp. 77–81.
[3] D. E. Knuth, *The Art of Computer Programming*, vol. 2, Addison-Wesley, Reading, Mass., 1969.
[4] A. Schönhage, *Eine untere Schranke für die Lange von Additionsketten*, preprint, 1974.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

b-MATCHINGS IN TREES*

S. GOODMAN†, S. HEDETNIEMI† AND R. E. TARJAN‡

Abstract. We develop linear-time algorithms to find maximum weighted and unweighted degree-constrained subgraphs (b-matchings) of a tree. We use a generalization of an algorithm for finding a maximum 2-matching in a tree.

Key words. tree, matching, degree-constrained subgraph, postorder numbering, Hamiltonian cycle

Let T be a tree with n vertices. For any vertex v, let $d(v)$ denote the *degree* (number of incident edges) of v. For each vertex v, let a *bound* $b(v)$ such that $0 \leqq b(v) \leqq d(v)$ be given, and for each edge $(u, v) \in T$, let a real-valued *cost* $c(u, v)$ be given. If $S \subseteq T$ is a subset of the edges of T and $|\{(u, v) \in S\}| \leqq b(v)$ for all v, then S is a *b-matching* of T. We consider the problem of finding

(i) a b-matching S such that $|S|$ is maximum (S is called a *maximum (unweighted) b-matching*);

(ii) a b-matching S such that $\sum_{(u,v)\in S} c(u, v)$ is maximum (S is called a maximum weighted b-matching);

(iii) a maximum (unweighted) b-matching S such that $\sum_{(u,v)\in S} c(u, v)$ is maximum. Efficient algorithms exist to solve these problems in arbitrary graphs [3], [7]; we develop $O(n)$ algorithms to solve them in trees by generalizing an $O(n)$ algorithm for finding maximum 2-matchings in trees devised by Goodman and Hedetniemi [5].

Let $C = \sum_{(u,v)\in T} c(u, v)$ and for each edge $(u, v) \in T$ let $c'(u, v) = 1 + c(u, v)/(2C)$. Then $\sum_{(u,v)\in T} (c'(u, v) - 1) = \frac{1}{2}$. If S is any b-matching of T, then S is a b-matching such that $\sum_{(u,v)\in S} c'(u, v)$ is maximum if and only if S is a maximum unweighted b-matching such that $\sum_{(u,v)\in S} c(u, v)$ is maximum. Thus by changing the cost function we can convert a problem of type (iii) into a problem of type (ii).

Let $\alpha: \{1, 2, \cdots, n\} \leftrightarrow \{v \text{ is a vertex of } T\}$ be a bijection such that, for all vertices v, $F(v) = \{w | (v, w) \in T \text{ and } \alpha^{-1}(w) > \alpha^{-1}(v)\}$ has no more than one element. Such a numbering of the vertices of T may be computed in $O(n)$ time by converting T into a directed, rooted tree and numbering the vertices in postorder (see [6], [9], [10]). If $|F(v)| = 1$, let $f(v)$ be the unique element of $F(v)$; otherwise let $f(v) = 0$. Henceforth, assume that vertices are identified by number (i.e., $\alpha(i) = i$).

Our algorithm to find unweighted b-matchings is based on the following simple observation. If (u, v) is the only edge incident to u in T, $b(u) > 0$, and $b(v) > 0$, then there is a maximum b-matching which contains (u, v). The following algorithm

* Received by the editors October 4, 1974.

† Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, Virginia 22901.

‡ Department of Computer Science, Stanford University, Stanford, California 94305. This research was supported in part by the National Science Foundation under Grant GJ-35604X and a Miller Research Fellowship at University of California, Berkeley, and by the National Science Foundation under Grant GJ-36473X at Stanford University.

computes a maximum b-matching using this observation. The edges $(i, f(i))$ such that $s(i) =$ **true** when the algorithm finishes are the edges in the b-matching. For each vertex i, COUNT(i) is the number of edges incident to i currently in the b-matching. To handle vertex n, we set COUNT(0) := $b(0) := 0$ by convention, since $f(n)$ is defined to be 0.

```
        ALGORITHM TREEMATCH: begin
        initialization: for i:=1 until n do COUNT(i):=0;
          COUNT(0):=b(0):=0;
mainloop: for i:=1 until n do
            if (COUNT(i) = b(i)) or (COUNT(f(i)) = b(f(i))) then
              s(i):=false;
            else begin
              s(i):=true;
              COUNT(f(i)):=COUNT(f(i))+1;
            inc: COUNT(i):=COUNT(i)+1;
            end;
      end TREEMATCH;
```

To prove that TREEMATCH works, we must show that the edges $(i, f(i))$ with $s(i) =$ **true** when the algorithm concludes from a maximum b-matching S of T. We show by induction on j that there is some maximum b-matching $S(j)$ of T such that, for all $k \leqq j$, $(k, f(k)) \in S(j)$ if and only if $S(k) =$ **true** when TREEMATCH finishes.

The hypothesis is clearly true for $j = 0$; $S(0)$ can be *any* maximum b-matching. Thus suppose $S(j-1)$ satisfies the hypothesis for $j-1$. Consider the iteration of *mainloop* having index $i = j$. If COUNT(j) = $b(j)$ or COUNT($f(j)$) = $b(f(j))$ when *test* is executed in this iteration of *mainloop*, then $(j, f(j)) \notin S(j-1)$ and $S(j) = S(j-1)$ satisfies the hypothesis for j. (Note that this case includes the case when $j = n$, since COUNT($f(n)$) = COUNT(0) = 0 = $b(0)$.)

Suppose, on the other hand, that COUNT(j) < $b(j)$ and COUNT($f(j)$) < $b(f(j))$. If $(j, f(j)) \in S(j-1)$, then $S(j) = S(j-1)$ satisfies the hypothesis for j. If $(j, f(j)) \notin S(j-1)$, then $S(j-1)$ has COUNT(j) $\leqq b(j) - 1$ edges incident to j, and there is some edge $(k, f(j)) \in S(j-1)$ such that $(k, f(j))$ is not of the form $(l, f(l))$ for any $l < j$. Then $S(j) = S(j-1) - (k, f(j)) + (j, f(j))$ satisfies the hypothesis for j. By induction, the hypothesis is true in general, and TREEMATCH works correctly. TREEMATCH obviously requires $O(n)$ time and space. The algorithm will find maximum b-matchings in forests as well as trees. Statement *inc* can be deleted without affecting the final values of $s(i)$.

If $b(i) = 2$ for all i, a maximum b-matching is called a *maximum 2-matching*. A maximum 2-matching can be used to find the minimum number of edges which must be added to a tree so that it has a Hamiltonian cycle. (This number, the *Hamiltonian completion number* of T, is equal to n minus the number of edges in a maximum 2-matching of T.) See [1], [4].

Our algorithm for finding maximum weight b-matchings is an extension of TREEMATCH and is based on the following observation: Let v be a vertex of T

adjacent to at most one vertex w of degree higher than one. Let $u_1, u_2, \cdots, u_k$ be the vertices of degree one adjacent to v, in decreasing order of $c(u_i, v)$. Then there exists some maximum weight b-matching containing the edges $(u_1, v), \cdots, (u_{b(v)-1}, v)$ and in addition either $(u_{b(v)}, v)$ or (w, v). Thus, if T' is the tree formed from T by deleting edges $(u_1, v), \cdots, (u_k, v)$ and c' is a cost defined on the edges of T' by $c'(w, v) = c(w, v) - c(u_{b(v)}, v)$ and $c'(x, y) = c(x, y)$ if $(x, y) \neq (w, v)$, then any maximum weight b-matching of T' may be converted into a maximum weight matching of T by adding edges $(u_1, v), \cdots, (u_{b(v)-1}, v)$, and $(u_{b(v)}, v)$ if (w, v) is not already in the b-matching.

The following algorithm implements this idea. Let the vertices of T be numbered as before. For each vertex i, let $A(i) = \{j | (i, j) \in T \text{ and } \alpha^{-1}(i) > \alpha^{-1}(j)\}$. The algorithm changes the costs of edges according to the scheme given above. The program uses the variable CEE(i) to denote the current value of $c(i, f(i))$. To insure proper behavior at vertex n, we set CEE(n) = 0. The maximum weight b-matching consists of the edges $(i, f(i))$ such that $s(i)$ = **true** when the algorithm finishes. For each vertex v, $p(v)$ is computed to be the vertex $u \in A(v)$ with $b(v)$th largest value of CEE(u), if this vertex has CEE(u) < CEE(v). Otherwise, $p(v) = 0$. Further explanation follows the program.

```
        ALGORITHM WTREEMATCH: begin
initialization: for i:=1 until n-1 do begin
                    CEE(i):=c(i, f(i));
                    p(i):=0;
                end;
                CEE(n):=p(n):=0;
   mainloop: for i:=1 until n do begin
          select: find the element j ∈ A(i) with b(i)th largest value of CEE(j);
          order: order the elements of A(i) so that j occurs in the b(i)th
                 position and any k ∈ A(i) occurring before j has
                 CEE(k) ≧ CEE(j);
          check: if (CEE(i) > CEE(j)) then begin
                     CEE(i):=CEE(i)-CEE(j);
                     if CEE(j) > 0 then p(i):=j;
                 end else CEE(i):=CEE(j):=0;
                 flag:=true;
            add: for k ∈ A(i) do begin
                     if (j≠k) or (CEE(i) = 0) then begin
                         if CEE(k) > 0 then s(k):=flag else s(k): = false;
                         m:=k;
      backtrack: while p(m) ≠ 0 do begin
                         s(p(m)) = ¬s(m);
                         m:=p(m);
                     end end;
                     if j:=k then flag:=false;
                 end end;
             end WTREEMATCH;
```

In this algorithm, statement *select* finds the $j \in A(i)$ with $b(i)$th largest value of CEE(j), and *order* orders $A(i)$ so that the k's in $A(i)$ with the $b(i) - 1$ largest values of CEE(k) occur first, followed by j, followed by the rest of $A(i)$. Statement *check* updates the value of CEE(j) if there is a choice of whether to add (j, i) or $(i, f(i))$ to the b-matching. This step also sets $p(i) = j$ so that adding $(i, f(i))$ to the b-matching later will force (j, i) to remain out of the b-matching, and leaving $(i, f(i))$ out will force (j, i) to be added. If there is no choice, then (j, i) must be added to the b-matching, and statement *check* sets CEE(i): = CEE(j) = 0. Statement *add* decides whether to add (k, i) to the matching for each $k \in A(i)$. *Add* adds (j, i) to the matching if CEE(i) = 0. After a decision whether to add (k, i) to the b-matching is made, statement *backtrack* makes the delayed decisions which are now forced, by following p pointers. The program never adds an edge of zero or negative cost to the b-matching.

It is not hard to prove that WTREEMATCH works correctly in a way similar to that used in the correctness proof of TREEMATCH. We must use a double induction instead of a single induction to take care of the delayed decisions which *backtrack* makes; this is the only complication. WTREEMATCH also works for forests.

Statements *select* and *order* can be implemented to run in $O(|A(i)|)$ time using a complicated but linear-time recursive procedure (see [2]). If there is a constant B such that $b(i) \leqq B$ for all i, then it is much easier to implement *select* to run in linear time (where the constant factor depends on $\log B$). Each edge of the tree is examined once in *select*, once in *add*, and once in *backtrack*. Thus it is clear that the total time and space requirements of the algorithm are $O(n)$.

We have presented linear-time algorithms for finding unweighted and weighted b-matchings in trees. Here are two open questions for future research. Do these algorithms generalize to (circuit-free) matroid problems? Are there fast algorithms for finding maximum b-matchings on more complicated kinds of graphs, such as k-trees [8]? Is there a linear-time algorithm to find a b-matching of fixed (not necessarily maximum) cardinality and maximum weight?

REFERENCES

[1] F. T. BOESCH, S. CHEN AND J. A. M. MCHUGH, *On covering the points of a graph with point disjoint paths*, presented at the Capitol Conference on Graph Theory and Combinatorics, Washington, D.C., 1973.

[2] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST AND R. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[3] J. EDMONDS, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[4] S. E. GOODMAN AND S. T. HEDETNIEMI, *On the Hamiltonian completion problem*, presented at the Capitol Conference on Graph Theory and Combinatorics, Washington, D.C., 1973.

[5] ———, *A linear algorithm for the Hamiltonian completion problem for trees*, unpublished manuscript, 1974.

[6] D. E. KNUTH, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968, pp. 334–338.

[7] E. L. LAWLER, *Combinatorial Optimization: Matroids and Networks*, Holt, Rinehart and Winston, to be published in 1975.

[8] D. ROSE, *On simple characterizations of k-trees*, Discrete Math., 7 (1974), pp. 317–322.
[9] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.
[10] ———, *A note on finding the bridges of a graph*, Information Processing Letters, 2 (1974), pp. 160–161.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

BOUNDS FOR SELECTION*

LAURENT HYAFIL†

Abstract. In this paper we show that the minimum number of comparisons necessary for the computation of the kth element of a totally ordered set of size n, $V_k(n)$, is bounded below by $n - k + (k-1)\lceil \log_2 (n/(k-1))\rceil$. For $3 < k < n/4$, this bound is an improvement on the best lower bound presently known. A new algorithm which yields an upper bound that is better than the currently known bound for a large range of values of n will also be presented.

Key words. selection, sorting, analysis of algorithm

1. Introduction. The selection problem is to determine the kth element of a totally ordered set P of size n. Two efficient algorithms for solving this problem are presently known. When k is small with respect to n, $k < 4n/\lg n$,[1] A. Hadian and M. Sobel's algorithm [3], which needs at most $n - k + (k-1)\lceil \lg (n - k + 2)\rceil$ comparisons, is adequate. Another method, using at most $5.43n$ comparisons,[2] was discovered by M. Blum et al. [1]. This method is more efficient than Hadian and Sobel's method for $k \geqq 4n/\lg n$.

Let $V_k(n)$ denote the minimum number of comparisons necessary for finding the kth element of a set of size n. The exact values of $V_k(n)$ are known for $k = 1 (V_1(n) = n - 1)$, and $k = 2 (V_2(n) = n - 2 + \lceil \lg n \rceil)$ (Schreier and Kislitsyn). For $k = 3$, F. Yao [6] has obtained a lower bound which is equal to the upper bound of Hadian and Sobel for infinitely many values of n. V. Pratt and F. Yao [5] also showed that:

$$\text{for } k \leqq \lg n/2 \lg\lg n, \qquad V_k(n) \geqq n - k + (k-1)[\lg n - (k-1)\lg^* n - 2\lg((k-1)!)],$$

$$\text{for } k \leqq n/3, \qquad V_k(n) \geqq n + 2k - \lg n,$$

$$\text{for } n/3 < k < \lfloor n - 3/2 \rfloor, \qquad V_k(n) \geqq (3n + k)/2 - \lg n - O(1),$$

improving the bound due to Blum et al., except when $\lg n/2 \lg\lg n < k < \lg n$.

In this paper we first present a new lower bound for $V_k(n)$, namely: $n - k + (k-1)\lceil \lg (n/k - 1)\rceil \leqq V_k(n)$. When $3 < k < n/4$, this bound is strictly greater than the best previously known bound. For instance, for $k = 5$, this result together with the best known upper bound enables us to determine the value of $V_5(n)$ within a gap of at most 8, while the previously known bounds leave a gap of at

* Received by the editors July 17, 1974, and in revised form May 15, 1975.

† Institut de Recherche d'Informatique et d'Automatique, Domaine de Voluceau-Rocquencourt, 78150 Le Chesnay, France.

[1] lg stands for $\log_2$.

[2] M. Paterson, N. Pippinger, A. Schönhage have set up a new method which needs only $3n$ comparisons.

least 80. Furthermore, this result shows that, for a fixed value of k, the tree selection algorithm is asymptotically optimal.[3]

We then present a new algorithm for selecting the kth largest element of a set of size n which yields an upper bound that improves strictly the previously known bound when $2 \lg n < k < 4n/\lg n$ and when

$$2^i + k - 2 < n \leqq 2^{\lceil((k-2)/(k-1))i\rceil - 1} + 2^i$$

for some integer i. Specifically, for $k = 3$, the new upper bound is

$$V_3(n) \leqq n - 3 + \lceil \lg (n-1) \rceil + \lceil \lg (n - 2^{\lfloor \lg n)/2 \rfloor}) \rceil.$$

Following the original formulation of the selection problem by Rev. C. L. Dodgson (better known as Lewis Caroll) [2], we call an element of the set P a "player" and a comparison between two players a "match" which must be won by one of the two players. A procedure for selecting the kth largest element will be referred to as a "tournament" for determining the kth best player.

2. The lower bound. We want to show that for any algorithm that computes the kth best player among n players, there exists a ranking of the players such that this algorithm must perform at least $n - k + (k-1)\lceil \lg (n/(k-1)) \rceil$ comparisons. The idea of using an oracle in our proof is due to Knuth [4], who gave a new proof of Kislitsyn's lower bound for $k = 2$. Here we extend this idea for all values of k. Our oracle is basically a deterministic process which builds up a ranking among the players, while the algorithm tries to find out the solution. This ranking, which must satisfy transitivity and antisymmetry, will force the algorithm to perform at least $V_k(n)$ comparisons. A correct algorithm cannot stop before the kth player is uniquely determined by the oracle. As a direct consequence, the set of the $k-1$ best players must also be uniquely determined.

We describe the oracle $\mathcal{O}$ as an automaton whose states are represented by ordered pairs. To be specific, the state vector S_t before the tth match is (φ_t, E_t) where φ_t is a mapping from P to $\mathbb{N}$, and E_t is a totally ordered subset of P. The initial state is $S_1 = (I, \varnothing)$ where I is the constant mapping such that $\forall x \in P$, $I(x) = 1$. Roughly speaking, the players in E_t are the top players, specifically the ith player to enter the set E_t is the ith best player. Candidates for entering E_t are selected according to the values of φ_t.

The input to the oracle at time t is an unordered pair of players $\{x, y\}$, who are engaged in the tth match according to the selection procedure. The oracle decides the winner of the match and enters state S_{t+1} according to the following rules:

R1—If $x \in E_t$ and $y \in E_t$, then x wins if and only if $x > y$ (E_t is an ordered set). Moreover, $S_{t+1} := S_t$.

R2—If $x \in E_t$ and $y \notin E_t$, then x wins and $S_{t+1} := S_t$.

R3—If $x \notin E_t$ and $y \notin E_t$, then if $\varphi_t(x) > \varphi_t(y)$, x wins; and if $\varphi_t(x) = \varphi_t(y)$, an arbitrary decision compatible with transitivity will be made. In both

[3] This result has been obtained independently by D. Kirkpatrick of the University of Toronto.

cases, suppose the winner is x; if $\varphi_t(x) + \varphi_t(y) \geqq n/(k-1)$, then $\varphi_{t+1} := \varphi_t$, $E_{t+1} := E_t \cup \{x\}$ and x becomes the smallest element of E_{t+1}. If $\varphi_t(x) + \varphi_t(y) < n/(k-1)$, then $E_{t+1} := E_t$, $\varphi_{t+1}(y) := 0$, $\varphi_{t+1}(x) := \varphi_t(x) + \varphi_t(y)$ and $\forall z \neq x, y, \varphi_{t+1}(z) := \varphi_t(z)$.

Being given that x dominates only x at time 1, we say that x dominates y at time $t+1$ if x dominates y at time t, or if x has beaten y in the tth match, or if x dominates z and z dominates y. Clearly, if x dominates y, x is a better player than y.

THEOREM. *The number* $V_k(n)$ *satisfies* $n - k + (k-1)\lceil \lg(n/(k-1))\rceil \leqq V_k(n)$.

We first prove the following lemma.

LEMMA. *Using oracle* $\mathcal{O}$, *the* $k-1$ *best players will have played at least* $(k-1)\lceil \lg(n/(k-1))\rceil$ *matches when the tournament is completed.*

Proof. The lemma follows from the facts listed below.

Fact 1. The number of matches won by x by time t is greater or equal to $\lceil \lg \varphi_t(x)\rceil$.

Fact 2. Let $e_i \in E_t$ be the ith player ($1 \leqq i \leqq |E_t|$) to enter E_t. Then e_i can be dominated only by e_j with $j \leqq i$.

Fact 3. $\sum_{x \in P} \varphi_t(x) = n$.

We denote with W_t the set of players x such that $x \notin E_t$ and $\varphi_t(x) > 0$.

Fact 4. $|E_t| + |W_t| > k - 1$.

This is a consequence of Fact 3 and of the fact that $\forall x \in P$, $\varphi_t(x) < n/(k-1)$.

Fact 5. At the end of the tournament, $|E_t| \geqq k - 1$.

Since the players in W_t can be dominated only by the players in E_t, if $|E_t| < k-1$, then any player in E_t or W_t can be one of the $k-1$ best players. Contradiction results from Fact 4.

Fact 6. At the end of the tournament, the $k-1$ best players are the $k-1$ top players in E_t.

This is a consequence of Facts 2 and 5.

Since x entering E_{t+1} by defeating y implies $\varphi_t(x) + \varphi_t(y) \geqq n/(k-1)$ and $\varphi_t(x) \geqq \varphi_t(y)$, the result is a direct consequence of Facts 1 and 6. □

Proof of theorem. According to the lemma, the $k-1$ best players have played at least $(k-1)\lceil \lg(n/(k-1))\rceil$ matches. Clearly, any player who is not among the k best players has lost at least one match against a player which is not among the $k-1$ best. Thus there are $n-k$ additional matches which were not included in the count of the matches played by the $k-1$ top players. This completes the proof of the theorem.

3. Improving the upper bound. Since Hadian and Sobel's algorithm needs at most $n - k + (k-1)\lceil \lg(n-k+2)\rceil$ comparisons, the new lower bound presented above enables us to determine $V_k(n)$ to within a gap of at most $(k-1)\lceil \lg(k-1)\rceil$ comparisons. The new algorithm we present reduces that gap when $\lg n < k/2$ and when

$$2^i + k - 2 < n \leqq 2^i + 2^{\lceil ((k-2)/(k-1))i\rceil - 1} \quad \text{for any integer } i.$$

We describe the algorithm in a pseudo-ALGOL dialect including set operations ($\cup$, $\cap$, $-$) and list operations (first, last, ", " for concatenation).

We first describe the procedure BEST(i, S), which is a tree selection algorithm used to determine the ordered list of the i best players of the set S. The set S is initially divided into two disjoint subsets S_1 and S_2, such that $S = S_1 \cup S_2$ and $|S_1| = 2^{\lceil \lg |S| \rceil - 1}$. Furthermore, each set is associated with a list TOP(S), which is initially empty.

```
list procedure WINNER(list L1, list L2) := if last (L1) > last (L2)
        then L1 else L2;
comment: WINNER uses one comparison except if one of the two
        lists is empty;
list procedure BEST(integer i, set S);
begin if S ≠ ∅ then
    begin for j = |TØP(S)|+1 until i do
        begin W:=WINNER (BEST (1, S1), BEST (1, S 2));
            if TØP(S1) = W then
            begin TØP(S1):=∅; S1:=S1−W; end;
                        else
            begin TOP(S2):=∅; S2:=S2−W; end;
            TØP(S):=TØP(S), W;
        end;
    end;
    TØP(S);
end.
```

This tree selection algorithm performs at most $|S| - i + (i-1)\lceil \lg |S| \rceil$ comparisons (see, for instance, [4] for further details). The new algorithm is an extension of this tree selection algorithm. Let P be the initial set of players which is divided into two disjoint subsets P_1 and P_2 such that $P_1 \cup P_2 = P$ and $|P_1| = 2^{\lceil \lg |P| \rceil - 1}$. The procedure BEST applied to P selects top players one by one in P_1 and P_2. The new algorithm uses two sequences of positive integers $\{u_\alpha\}$ and $\{v_\alpha\}$, and a characteristic step is to select either the u_h top players of P_1 or the v_j top players of P_2, according to the results of previous comparisons.

```
list procedure SELECT(integer k, set P);
begin h:=1; j:=1; A:=u1 + v1;
    while A ≦ k do
L1: begin W:=WINNER(BEST(uh, P1), BEST (vj, P2));
        if TØP(P1) = W then
        begin TØP(P1):=∅; P1:=P1 − W;
            h:=h + 1; A:= A+ uh;
        end;
                    else
        begin TØP (P2):=∅; P2:=P2 − W;
            j:=j + 1; A:=A + vj;
        end;
        R:=k−A+uh+vj;
        TØP(P):=TØP(P), PICK(BEST(R, P1), BEST(R, P2));
```

comment: TOP(P) contains the k best players of P, furthermore the kth element of TOP(P) is the kth player of P;

end.

list procedure PICK (**list** L_1, **list** L_2)

comment: selects the top R players from the ordered lists L_1 and L_2 of length R using R comparisons;

Remark: It is possible (and sometimes more efficient) to use the procedure SELECT recursively instead of the procedure BEST. In that case, since the result of SELECT is not an ordered list, it is also necessary to replace the procedure PICK.

Analysis of the algorithm. An exhaustive analysis of the algorithm to determine the best possible choices of $\{u_\alpha\}$ and $\{v_\alpha\}$ for given values of n and k being quite tedious, we restrict our study to particular values of $\{u_\alpha\}$ and $\{v_\alpha\}$.

A comparison performed when line L1 of the algorithm is executed, or a comparison performed in the procedure PICK, clearly determines at least one new element of the pool of the k best players. Such a comparison will be referred to as an *active* comparison.

Case 1. $u_\alpha = v_\alpha = a$, $a \in N$, *for all integer* α. Assuming that $k = ta$, $t \in N$, $a + t - 1$ active comparisons are performed and at most $n - 2 + (k + a - 2)(\lceil \lg n \rceil - 2)$ inactive ones. So the difference between the number of comparisons performed by tree selection and the number of comparisons performed by this algorithm is clearly equal to

$$k - [(a - 1)(\lceil \lg n \rceil - 1) + k/a].$$

The choice $a = 2$ shows that this algorithm strictly improves on tree selection if $k > 2(\lceil \lg n \rceil - 1)$. In fact, there is an optimal manner of choosing a which is the closest integer to

$$\sqrt{k/(\lceil \lg n \rceil - 1)}.$$

For instance, suppose we are to select the 90th player among a set of 2048. The choice $u_\alpha = v_\alpha = 3$, for all α, in our algorithm will save 39 comparisons over tree selection.

Case 2. We want to choose $\{u_\alpha\}$ and $\{v_\alpha\}$ such that, in the worst case, the number of inactive comparisons is equal to $n - 2k + (k - 1)\lceil \lg n \rceil$. Such a choice guarantees that the algorithm is not worse than tree selection.

Assume that $n = 2^{i_1} + 2^{i_2}$, with $i_1 > i_2$. The values of $\{u_\alpha\}$ must satisfy the relation

$$n - 2 - \left(1 - \sum_{i \leqq \alpha \leqq h} u_\alpha\right)(i_1 - 1) + \left(k - 1 - \sum_{i \leqq \alpha \leqq h} u_\alpha\right)(i_2 - 1) \leqq n - 2k + (k - 1)(i_1 + 1);$$

that is,

$$u_k \leqq 1 + \frac{i_1 - i_2}{i_1 - 1}\left(k - 1 - \sum_{1 \leqq \alpha \leqq h-1} u_\alpha\right).$$

The choice $v_\alpha = 1$ for all integer α appears to be always convenient, and a simple calculation yields that the algorithm improves strictly on tree selection if

$$i_2 < \frac{(k-2)i_1 + 1}{k-1}.$$

For instance, for $k = 7$, using the sequence $u_1 = 4$, $u_2 = 2$, $u_3 = 1$, saves 3 comparisons on tree selection if

$$2^{i_1} < n \leqq 2^{i_1} + 2^{\lfloor (i_1+1)/2 \rfloor}.$$

For $k = 3$, using $u_1 = 2$ and $u_2 = 1$ saves one comparison on tree selection if

$$2^{i_1} < n \leqq 2^{i_1} + 2^{\lceil (i_1+1)/2 \rceil - 1},$$

and the new upper bound for $V_3(n)$ is

$$V_3(n) \leqq n - 3 + \lceil \lg (n-1) \rceil + \lceil \lg (n - 2^{\lfloor (\lg n)/2 \rfloor}) \rceil.$$

Acknowledgments. I am very grateful to C. L. Liu, J. A. Koch and E. M. Reingold for their comments and suggestions during the preparation of this paper.

REFERENCES

[1] M. Blum, R. Floyd, V. Pratt, R. Rivest and R. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[2] C. L. Dodgson, St. James's Gazette, August 1, 1883, pp. 5–6.

[3] A. Hadian and M. Sobel, *Selecting the t-th largest using binary errorless comparisons*, Tech. Rep. 121, Dept. of Statist., Univ. of Minnesota, Minneapolis, 1969.

[4] D. Knuth, *The Art of Computer Programming*, vol. 3, Addison-Wesley, 1973, pp. 209–220.

[5] V. Pratt and F. Yao, *On lower bounds for computing the i-th largest element*, Proc. 14th Symp. on Switching and Automata Theory, Iowa City, Iowa, 1973, pp. 70–81.

[6] F. Yao, *On lower bounds for selection problems*, Tech. Rep. MAC TR-121, Mass. Inst. of Tech., Cambridge, 1974.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

ON FINDING LOWEST COMMON ANCESTORS IN TREES*

A. V. AHO†, J. E. HOPCROFT‡ AND J. D. ULLMAN¶

Abstract. Trees in an n-node forest are merged according to instructions in a given sequence, while other instructions in the sequence ask for the lowest common ancestor of pairs of nodes. We show that any sequence of $O(n)$ such instructions can be processed "on-line" in $O(n \log n)$ steps on a random access computer.

If we can accept our answer "off-line", that is, no answers need to be produced until the entire sequence of instructions has been seen, then we may perform the task in $O(n\alpha(n))$ steps, where $\alpha(n)$ is the very slowly growing inverse Ackermann function defined in [14].

A third algorithm solves a problem of intermediate complexity. We require the answers on-line, but we assume that all tree merging instructions precede the information requests. This algorithm requires $O(n \log \log n)$ time.

We apply the first on-line algorithm to a problem in code optimization, that of computing immediate dominators in a reducible flow graph. We show how this computation can be performed in $O(n \log n)$ steps.

Key words. algorithms, computational complexity, graphs, trees, first common ancestor, code optimization, dominators

1. Introduction. Suppose that we are running the following genealogy service. During the course of a day, we receive new information concerning the ancestry relationships among a fixed set of men. (E.g., "B is a son of A.") We also receive requests asking for the closest common male ancestor of pairs of men. (E.g., "Who is the most recent common male parent of C and D?") Our problem is to process each new request in turn using the most current information.

We can abstract our problem as follows. We have n nodes in a finite set of trees (see [1] for definitions), hereafter called a *forest*. We receive a sequence of instructions to execute. The instructions are of two types:

1. The instruction LINK(u, v) makes node u a son of node v. We assume that at the time this instruction is received, nodes u and v are on different trees and that u is a root. Thus, after executing this instruction, the nodes will remain a forest.

2. The instruction LCA(u, v) prints the lowest common ancestor of nodes u and v.

Example 1. Suppose that we initially have a forest consisting of eight isolated

* Received by the editors May 3, 1973.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Department of Computer Science, Cornell University, Ithaca, New York 14850. The work of this author was supported by the Office of Naval Research under Grant N00014-67-A-0077-0021.

¶ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540. The work of this author was supported in part by the National Science Foundation under Grant GJ-1052.

nodes $u_1, u_2, \cdots, u_8$ and we receive the following sequence of instructions.

LINK(u_1, u_2)
LINK(u_3, u_4)
LINK(u_5, u_6)
LINK(u_7, u_8)
LINK(u_2, u_4)
LINK(u_6, u_8)
LCA(u_5, u_7)
LINK(u_4, u_6)
LCA(u_2, u_3).

When the instruction LCA(u_5, u_7) is received, the forest is as shown in Fig. 1(a). Thus u_8, the lowest common ancestor of u_5 and u_7, is printed.

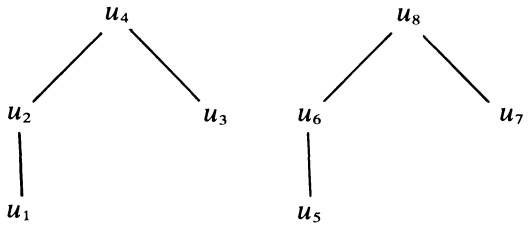


FIG. 1(a). *Tree structures after* LCA(u_5, u_7) *instruction*

Fig. 1(b) shows the forest when LCA(u_2, u_3) is executed. This instruction causes u_4 to be printed.

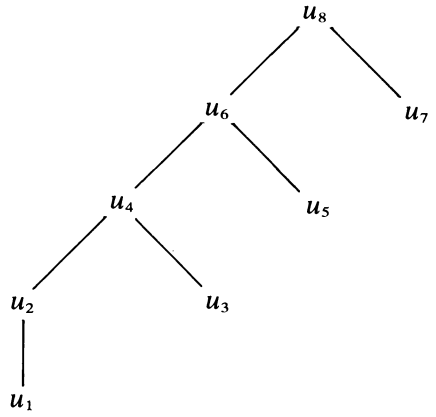


FIG. 1(b). *Tree structure after* LCA(u_2, u_3) *instruction*

In this paper we shall consider the problem of executing a sequence of $O(n)$ LINK and LCA instructions on a forest with n nodes. We shall hereafter refer to this sequence as σ. If we execute $O(n)$ LINK instructions, trees with paths of length $n - 1$ can develop. Consequently, if we execute the LCA instructions in the obvious way, we could spend $O(n)$ time on each LCA instruction, or $O(n^2)$ time in total when there are $O(n)$ LCA instructions.

We shall first give an on-line algorithm that requires $O(n \log n)$ steps to execute σ. We shall also provide an asymptotically faster off-line algorithm and an algorithm of intermediate complexity which solves an intermediate problem. We then apply the on-line algorithm to compute the immediate dominators of an n-node reducible program flow graph in $O(n \log n)$ steps.

2. A useful data structure for forests. In our on-line algorithm, we define two forests with information attached to the various nodes. These two forests are used together to find the least common ancestors. To distinguish these forests from the actual forest that would be constructed by the LINK instructions, we shall refer to the latter forest as the *implied forest*.

The first defined forest, which we call the *A-forest*, has the same structure as the implied forest. For the A-forest we shall maintain an array ANCESTOR[u, i], where u is a node and i an integer such that $0 \leqq i < \log n$.[1] At all times, ANCESTOR[u, i] will be either the (2^i)th ancestor of node u in the implied forest or will be undefined. ANCESTOR[u, i] could be undefined even though u has a (2^i)th ancestor, since we shall not compute ancestor information until needed. Maintenance of ancestor information will be discussed in § 4.

The second forest, called the *D-forest*, has nodes grouped into the same trees as the implied forest, but the internal structure of corresponding trees will in general be different. The sole purpose of the D-forest is to keep track of the depth of nodes in the implied forest.

In what follows, we shall refer to a node in the A- and D-forests merely by its name in the implied forest. We trust no confusion will result. It should be borne in mind, however, that "the depth of u" always refers to the depth of u in the implied forest or, equivalently, to the depth of u in the A-forest.

3. Maintaining the D-forest. Our first algorithm uses the D-forest to compute the depth of nodes in the implied forest. We attach an integer WEIGHT[u] to each node u in the D-forest. To find the depth of a node, we find the representative of the node in the D-forest and trace the path from this node to its root, summing the weights of the nodes along this path. Then, except for the root, we make each node along this path be a son of the root, updating the weights of the nodes appropriately.

ALGORITHM 1.

procedure DEPTH(u):
begin
1. Find the path from node u to its root in the D-forest. Suppose $u_k, u_{k-1}, \cdots, u_0$ is that path, where u_0 is the root and u_k is u.
2. $sum \leftarrow$ WEIGHT[u_0] + WEIGHT[u_1] + $\cdots$ + WEIGHT[u_k].
3. Make each of $u_2, u_3, \cdots, u_k$ a son of u_0 in the D-forest.
4. **for** $i = 2$ **until** k **do**
   WEIGHT[u_i] $\leftarrow$ WEIGHT[u_i] + WEIGHT[u_{i-1}];
5. **return** *sum*

end

[1] All logarithms in this paper are to the base 2.

The root of each tree in the D-forest also has an associated COUNT, giving the number of nodes in the tree. Initially each node in the D-forest is in a tree by itself, having a COUNT of 1 and a WEIGHT of 0.

We now give an algorithm to merge the two corresponding trees in the D-forest when a LINK(u, v) instruction is executed.

ALGORITHM 2 (Merging trees in the D-forest).

1. Find the roots x and y of the trees holding u and v, respectively, by executing steps 1, 3 and 4 of Algorithm 1.

2. Compute DEPTH(v) using Algorithm 1.

3. If COUNT[x] $\leqq$ COUNT[y], then make x a son of y in the D-forest and do the following:

COUNT[y] $\leftarrow$ COUNT[y] + COUNT[x];

WEIGHT[x] $\leftarrow$ WEIGHT[x] + DEPTH[v] + 1 $-$ WEIGHT[y].

4. Otherwise, if COUNT[x] $>$ COUNT[y], make y a son of x in the D-forest and counts appropriately.

COUNT[x] $\leftarrow$ COUNT[y] + COUNT[x];

WEIGHT[x] $\leftarrow$ WEIGHT[x] + DEPTH[v] + 1;

WEIGHT[y] $\leftarrow$ WEIGHT[y] $-$ WEIGHT[x].

In steps 3 and 4 we merge the smaller tree into the larger, adjusting the weights and counts appropriately.

LEMMA 1. *Suppose Algorithm* 2 *is used every time trees must be merged by a* LINK *instruction and Algorithm* 1 *is used every time we wish to compute* DEPTH[u]. *Then*

(a) *each value* DEPTH[u] *found in Algorithm* 1 *is correct, and*

(b) *if* $O(n)$ *tree merges and* $O(n)$ *depth computations are done, the total time spent in Algorithms* 1 *and* 2 *is* $O(n\alpha(n))$.[2]

Proof. Observe that Algorithm 1 does not change the sum of the weights along the path from any node to its root. In particular, step 4 of Algorithm 1 adds the sum of the weights of $u_1, u_2, \cdots, u_{i-1}$ to u_i for each node u_i moved in step 3. This adjustment corrects for the fact that the path from u_i to u_0 no longer passes through $u_1, u_2, \cdots, u_{i-1}$.

Algorithm 2 adds the value DEPTH(v) + 1 to paths from nodes in the tree containing u and does not change other paths. Since u is the root (in the A-forest) of its tree, we may conclude (a). For part (b) observe that the number of steps taken by Algorithm 1 is proportional to that of the set merging algorithm of [2]; [14] shows this algorithm takes $O(n\alpha(n))$ time. □

It is important that the weights in the D-forest do not grow too large since we are assuming that arithmetic on integers can be accomplished in one step. Should numbers grow larger than say $O(n)$, we would have to consider the cost of multiple-precision arithmetic. The following bound, however, justifies our ignoring the cost of arithmetic.

LEMMA 2. *No weight in the D-forest exceeds n in magnitude.*

Proof. Suppose that u is a node in the D-forest and $u, v_1, v_2, \cdots, v_k$ is the

[2] $\alpha(n)$ is the inverse Ackermann function defined in [14].

path from u to its root. Then the difference in the depth of nodes u and v_1 is easily seen to be WEIGHT[u]. Since no depth exceeds n, the difference of two depths cannot exceed n. Also, the depth of a node represented by a root in the D-forest is never greater than n. Thus $\lceil \text{WEIGHT}[u] \rceil \leqq n$ for all u. □

4. Computing ancestor information. Maintaining the structure of the A-forest is easy, since a LINK(u, v) instruction can be executed by attaching an additional pointer to node u, i.e., setting ANCESTOR[u, 0] to v. What is difficult is the maintenance of the ancestor information. We shall define a recursive routine INSTALL(u, i) which inserts the (2^i)th ancestor of u into the ANCESTOR array. This routine will be called at various times when ancestor information is needed to execute an LCA instruction. It is written under the assumption that ANCESTOR[u, 0] $\neq$ *undefined* for any u to which the routine INSTALL will be applied.

```
procedure INSTALL(u, i):
begin
    if ANCESTOR[u, i − 1] = undefined then INSTALL(u, i − 1);
    if ANCESTOR[ANCESTOR[u, i − 1], i − 1] = undefined then
        INSTALL(ANCESTOR[u, i − 1], i − 1);
    ANCESTOR[u, i] ← ANCESTOR[ANCESTOR[u, i − 1], i − 1]
end
```

Given the assumption that ANCESTOR[v, 0] is correctly defined for all v and that u has a (2^i)th ancestor, a straightforward induction on $i \geqq 1$ shows that INSTALL(u, i) correctly computes ANCESTOR[u, i].

We shall also define a procedure FIND(u, v, i, d) which takes as arguments two distinct nodes u and v of equal depth d such that $2^{i-1} \leqq d$ and such that the (2^i)th ancestors of u and v are the same or neither exists. The result of FIND(u, v, i, d) is the lowest common ancestor of u and v; FIND works by repeatedly halving the range in which the length of the path from u to the lowest common ancestor of u and v is known to lie.

```
procedure FIND(u, v, i, d):
if i = 0 then return ANCESTOR[u, 0];
else
    begin
        if ANCESTOR[u, i − 1] = undefined then INSTALL(u, i − 1);
        if ANCESTOR[v, i − 1] = undefined then INSTALL(v, i − 1);
        if ANCESTOR[u, i − 1] = ANCESTOR[v, i − 1] then return
            FIND(u, v, i − 1, d)
        else
            begin
                j ← min (i − 1, ⌊log (d − 2^(i−1))⌋);
                return FIND(ANCESTOR[u, i − 1],
                    ANCESTOR[v, i − 1], j, d − 2^(i−1))
            end
    end
```

It is straightforward to show that FIND works correctly given that $2^{i-1} \leqq d$. The selection of j on the next to last line of the procedure insures that $2^{j-1} \leqq d - 2^{i-1}$.

We can now give an algorithm to compute the lowest common ancestor of an arbitrary pair of nodes.

ALGORITHM 3 (Lowest common ancestor of u and v).

1. Use Algorithm 1 to compute DEPTH(u) and DEPTH(v). Assume without loss of generality that DEPTH(u) $\geqq$ DEPTH(v).

2. Find the ancestor a of u having the same depth as v by the following procedure:

```
begin
    a ← u;
    d ← DEPTH(u) − DEPTH(v);
    while d ≠ 0 do
        begin
            j ← ⌊log d⌋;
            if ANCESTOR[a, j] = undefined then INSTALL (a, j);
            a ← ANCESTOR[a, j];
            d ← d − 2^j
        end;
    return a
end
```

3. If $a = v$, then a is the lowest common ancestor of u and v. Otherwise, execute FIND(a, v, i, d), where d = DEPTH(v) and $i = \lfloor \log d \rfloor$.

5. An on-line algorithm. We now utilize Algorithms 1, 2 and 3 to execute the sequence σ on-line, that is, providing the answer to the ith instruction in σ before the $(i + 1)$st instruction is read.

ALGORITHM 4 (On-line execution of σ).

1. Initialize the D-forest with all nodes in separate trees, having counts of 1 and weights of 0.

2. Initialize the A-forest with all nodes in separate trees and with ANCESTOR[u, i] = *undefined* for all u and i.

3. Execute an LCA(u, v) instruction by applying Algorithm 3 to u and v. Print the resulting lowest common ancestor.

4. Execute a LINK(u, v) instruction as follows:

(a) Set ANCESTOR[$u, 0$] = v.

(b) Use Algorithm 2 to merge the trees in the D-forest holding u and v.

THEOREM 1. *If Algorithm 4 is applied to execute σ, the execution of the algorithm requires at most $O(n \log n)$ steps of a random access computer.*[3]

Proof. Algorithm 4 results in $O(n)$ calls of Algorithm 2. There are also $O(n)$ calls to Algorithm 3, which result in $O(n)$ calls to Algorithm 1. By Lemma 1, all calls to Algorithms 1 and 2 are handled in $O(n\alpha(n))$ steps.

Exclusive of calls to FIND and INSTALL, Algorithm 3 clearly requires

[3] See [1] for a discussion of the formal "RAM" model.

$O(\log n)$ steps per call, for a total of $O(n \log n)$ steps. Since FIND calls itself with the third argument decreased by at least 1 each time, at most $O(n \log n)$ calls to FIND may be made. Since each call of FIND requires constant time exclusive of calls to itself or to INSTALL, the total cost of FIND exclusive of calls to INSTALL is $O(n \log n)$.

Since INSTALL(u, i) is called only if ANCESTOR$[u, i]$ = *undefined*, and ANCESTOR$[u, i]$ will be defined after this call to INSTALL(u, i), we see that no more than $O(n \log n)$ calls of INSTALL can occur. Since each call of INSTALL requires constant time exclusive of calls to itself, the cost of INSTALL is $O(n \log n)$.

Thus Algorithm 4 requires at most $O(n \log n)$ steps on any sequence of $O(n)$ LINK and LCA instructions. □

One might argue that the "obvious" method of executing σ has an expected time of $O(n \log n)$, since a random sequence of LINK instructions might produce paths of length $O(\log n)$, rather than $O(n)$. In this case, however, it is easy to bound the expected (not worst-case) time taken by Algorithm 4 at $O(n \log \log n)$. In fact, if the expected path length in trees is $f(n)$, then our algorithm will run in $O(\max \{n \log f(n), n\alpha(n)\})$ steps. In § 7 we shall see that in the case where all LINK instructions precede all LCA instructions, $O(n \log \log n)$ is an upper bound on the running time of a modified algorithm, as well as its expected time.

6. An off-line algorithm. Algorithm 4 produces an answer to the ith instruction in σ before the $(i + 1)$st is read. If we are willing to wait until all of σ has been seen before producing any answers, however, we can do better than $O(n \log n)$; an $O(n\alpha(n))$ algorithm exists.

To begin, we use the $O(n\alpha(n))$ set merging algorithm of [2] to check that no LCA(u, v) instruction in σ has u and v on different trees. Having thus assured ourselves that we have a legal sequence of instructions, we may build the forest required by the LINK instructions in σ. If there is more than one tree in the final forest at the end, we can make all roots be sons of a new node, so that exactly one tree T results. For each LCA(u, v) instruction in σ, the lowest common ancestor of u and v in T will be their lowest common ancestor in the forest built by the LINK instructions preceding that LCA instruction in σ.

We shall number the nodes of T so that if we visit them in preorder, we visit them in the order $1, 2, \cdots$. For example, the nodes in Fig. 1(b) would be numbered as shown in Fig. 2.

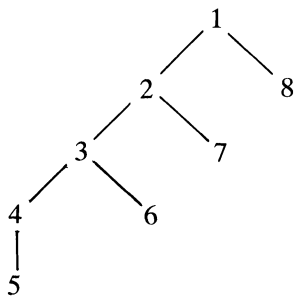


FIG. 2. *Preorder numbering*

The construction of T and the preorder numbering of the nodes can clearly be done in $O(n)$ steps.

Note that if a and b are preordered nodes, then $a < b$ if and only if

(i) a is an ancestor of b, or

(ii) a is to the left of b.

We shall now identify each LCA(u, v) instruction in σ with a distinct *object* X with which we shall associate the integer pair (i, j), such that $i < j$, and i and j are the numbers associated with nodes u and v. Let L and R be the projection functions such that $L(X) = i$ and $R(X) = j$.

We wish to generate the answers to the LCA instructions in σ. To do this, we shall first derive from the tree T a sequence τ of new instructions ENTER(X) and REMOVE(i), where X is an object and i an integer. We can think of these instructions as entering and removing objects from a "bin" which is initially empty.

1. The instruction ENTER(X) places object X in the bin.

2. The instruction REMOVE(i) removes from the bin all objects X such that $L(X) \geqq i$. In addition, for each object X removed, we set $A[X] = i$, where A is an array indexed by the objects. We shall see that i is the lowest common ancestor of the pair of nodes associated with the object X.

We shall subsequently show that the execution of the sequence τ can be simulated in $O(n\alpha(n))$ steps, off-line, by the $O(n\alpha(n))$ set merging algorithm of [2]. To begin, we show how the sequence τ is generated from the set of objects and the tree T.

ALGORITHM 5 (Generating τ).

1. For each node i, list those objects X for which $R(X) = i$.

2. Process each node of T in postorder. That is, node i is processed before node j if and only if i is to the left of j or a descendant of j. The nodes in postorder for the tree of Fig. 2 are:

$$5 \quad 4 \quad 6 \quad 3 \quad 7 \quad 2 \quad 8 \quad 1.$$

When at node i, do the following:

(a) Generate the instruction ENTER(X) for each X such that $R(X) = i$.

(b) Generate the instruction REMOVE(i).

We shall now prove an important property of the sequence τ of ENTER and REMOVE instructions generated by Algorithm 5.

LEMMA 3. *Object X is removed from the bin by the instruction* REMOVE(a) *in τ if and only if a is the lowest common ancestor of $L(X)$ and $R(X)$.*

Proof. If. Let X be an object such that a is the lowest common ancestor of $L(X)$ and $R(X)$. Object X will be placed in the bin by the ENTER(X) instruction generated when node $R(X)$ is processed. Because the nodes are processed in postorder, all nodes processed between $R(X)$ and a must either be ancestors of $R(X)$ and descendants of a or they must be descendants of a to the right of $R(X)$.

$L(X)$ is not the descendant of any node between $R(X)$ and a in the postorder. Thus $L(X) < u$ for all nodes u processed between $R(X)$ and a. Since $L(X) = a$ or $L(X)$ is a descendant of a, we must have $L(X) \geqq a$. Thus object X is removed from the bin by the REMOVE(a) instruction in τ.

Only if. Suppose object X is removed from the bin by the instruction REMOVE(a). Then $R(X)$ must precede a in the postorder, $L(X) \geqq a$, and for any node u between $R(X)$ and a in the postorder, $L(X) < u$.

Suppose $R(X)$ is not a descendant of a. Then $R(X)$ is to the left of a (since $R(X)$ precedes a in the postorder) and $R(X) < a$. But then $L(X)$ also must be less than a, a contradiction. Therefore $R(X)$ must be a descendant of a.

Let u be a descendant of a and an ancestor of $R(X)$. That is, u is a node between $R(X)$ and a in the postorder. $L(X)$ cannot be a descendant of u since $L(X) < u$. However, either $L(X) = a$ or $L(X)$ is to the left of $R(X)$. In addition, we know $L(X) \geqq a$. Thus $L(X)$ must be a descendant of a (or a itself). Hence, a is the lowest common ancestor of $L(X)$ and $R(X)$. □

We shall now give an algorithm that will simulate the execution of the sequence τ.

ALGORITHM 6 (Simulation of σ). We note that all instructions in σ are distinct, and that the last instruction in σ is REMOVE(1).

1. Suppose that there are k nodes in the tree T, so for all objects X, we have $1 \leqq L(X) < k$. (Note that $k \leqq n + 1$, where n is the number of nodes in the original forest.) For each i, $1 \leqq i \leqq k$, make a list $OBJ[i]$ of those objects X for which $L(X) = i$.

2. Create an "atom" e_X for each ENTER(X) instruction in τ and an atom r_i for each REMOVE(i) instruction in τ. Also create an initially empty set named S_i for each REMOVE(i) instruction in σ. Place e_X in S_i if $R(X) = i$. Place r_i in S_j if REMOVE(j) is the first REMOVE instruction in τ to follow REMOVE(i).

3. For $i = k, k - 1, \cdots, 1$ in turn do the following:

(a) For each X on $OBJ[i]$, find the set S_j of which e_X is currently a member. Then do (b) and (c).

(b) If $i \geqq j$, place X on list $REM[j]$, which will hold all objects that are removed from the bin when the instruction REMOVE(j) in τ is executed. Consider the next X in step 3(a).

(c) If $i < j$, merge set S_j with that set S_h such that r_j is in S_h. Call the new set S_h. Return to step (b) with j set to h.

4. Examine each REMOVE(i) instruction of τ in turn from the beginning. List those pairs (X, i) such that X is on $REM[i]$.

In step 1 of Algorithm 6 we create the list OBJ to sort the objects in terms of their first components. In step 2 we enter the atoms representing the objects into sets indexed by the second component of the object. We also include in set S_j the atom r_i corresponding to the instruction REMOVE(i) if node j follows node i in the postorder. In step 3, for each object X in set S_j, we locate via the r-atoms the first ancestor a of node j such that $L(X) \geqq a$. Node a is the lowest common ancestor of nodes $L(X)$ and $R(X)$.

The motivation behind Algorithm 6 is that each REMOVE(i) instruction in τ is presumed to remove from the bin all objects X such that the instruction ENTER(X) precedes REMOVE(i) in τ. If we are working on some X for which $L(X) > i$, however, then we will have already found all those objects which will be removed by the instruction REMOVE(i). We therefore "get rid of" the instruction REMOVE(i) by merging the set S_i with the set for the next remaining REMOVE instruction.[4] The atom r_i allows us to find the next REMOVE instruction, since r_i will always be in the set associated with that instruction.

[4] Note that the last instruction, REMOVE(1), can never disappear, so step 3(c) can always be carried out.

A formal proof that Algorithm 6 works correctly is quite similar to the proof regarding the "INSERT-EXTRACT" instructions in [2], and we omit it.

We now summarize the off-line LINK-LCA algorithm.

ALGORITHM 7 (Off-line simulation of the sequence σ of LINK and LCA instructions).

1. Test that when an LCA(u, v) instruction is encountered in σ, u and v are on the same tree, using the $O(n\alpha(n))$ set merging algorithm of [2].
2. Build the forest as dictated by the LINK instructions in σ. If necessary, add one root to make the final forest a tree T.
3. Number the nodes of T in preorder.
4. For each LCA(u, v) instruction, create an object $X = (i, j)$, where i and j are the preorder numbers of u and v.
5. Use Algorithm 5 to generate the sequence τ of ENTER and REMOVE instructions for T and the set of objects created in step 4.
6. Use Algorithm 6 to simulate the sequence τ.
7. Scan the output of Algorithm 6. For each (X, i) in the output, set $A[X] = i$.
8. Scan the LCA instructions in the original sequence σ. For each LCA(u, v) instruction, determine the corresponding object X; $A[X]$ is the lowest common ancestor of u and v.

THEOREM 2. *Algorithm 7 requires $O(n\alpha(n))$ steps on a random access computer.*

Proof. Step 1 can be done in $O(n\alpha(n))$ steps. Steps 2–4 are each easily seen to be $O(n)$, and the sequence of ENTER and REMOVE instructions generated is $O(n)$ in length.

Since k in Algorithm 6 is at most $n + 1$, steps 1 and 2 of Algorithm 6 can be done in $O(n)$ time. In step 2, $O(n)$ atoms (of the forms e_X and r_i) are created. As step 3(c) of Algorithm 6 can apply only $O(n)$ times, step 3 involves at most $O(n)$ operations of merging two sets or finding the set containing a given node. Thus step 3 can be performed in $O(n\alpha(n))$ steps if we use the $O(n\alpha(n))$ algorithm of [2] for the set merging and name finding operations.

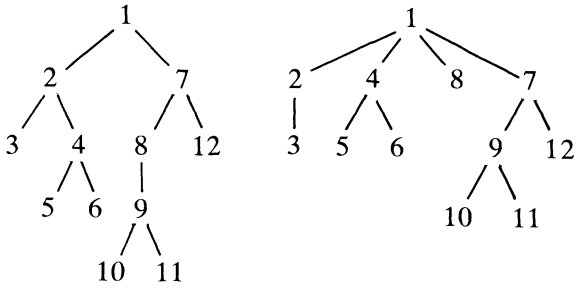
Finally, steps 7 and 8 are clearly $O(n)$. Thus the aggregate time required by Algorithm 7 is $O(n\alpha(n))$. □

7. An intermediate problem. Let us return to the on-line processing of σ, our original sequence of LINK and LCA instructions, but now assuming that all LINK instructions in σ precede all LCA instructions. In this case, we may build the implied forest first, and then process the LCA instructions without changing the forest. Before executing the LCA instructions, however, we shall modify the implied forest so that all paths in the forest are bounded by $O(\log n)$ in length. Then we can process each LCA instruction in at most $\log \log n$ steps.

Given a tree T in the forest, we shall construct from it a *virtual tree* V which has the same nodes as T but in which nodes have different fathers. The father of a node u in V is the lowest ancestor of u in T having at least twice as many descendants in T as does u. As a special case, if no such ancestor exists and u is not the root of T, we then make the root of T the father of u in V.

Example 2. A tree T is shown in Fig. 3(a). Its virtual tree is shown in Fig. 3(b). For example, node 9 has three descendants (we are assuming a node is a descendant

of itself). Node 8 has four, but node 7 has six, so node 7 becomes the father of node 9 in the virtual tree.



(a) *Tree* (b) *Virtual tree*

FIG. 3. *Tree and its virtual tree*

LEMMA 4. *Let T be a tree with n nodes and V its virtual tree. No path in V is longer than* $\log n$.

Proof. The ith node in a path beginning from a leaf has at least 2^{i-1} descendants in T, provided the ith node is not the root. If the path is longer than $\log n$, T would have more than n nodes, a contradiction. □

LEMMA 5. *Let T be a tree with n nodes and V its virtual tree. Let u and v be two nodes and let node a be their lowest common ancestor in T. Assume u, v and a are all distinct, and suppose v has at least as many descendants as u. Then f, the father of u in V, is a descendant of a in T (possibly a itself). Thus, in T, a is also the lowest common ancestor of f and v.*

Proof. Suppose not. Then node a would have fewer than twice as many descendants as u, a contradiction, since a has more descendants than u and v put together. □

An efficient off-line method for determining whether one of two nodes in a tree T is a descendant of the other is to preorder the nodes of T and to attach to each node i (that is, i is its number in preorder) the value HIGH[i] which is the highest numbered node that is a descendant of node i. Node i is a descendant of j if and only if $j \leqq i$ and HIGH[j] $\geqq$ HIGH[i]. It should be clear that HIGH can be computed for a tree with n nodes in $O(n)$ steps.

We also observe without proof that in $O(n)$ steps we can compute COUNT[u], the number of descendants of node u, for all nodes u. Furthermore, in $O(n)$ steps we can find for each node u, a son of u having the largest count.

We shall now outline an $O(n)$ algorithm to construct a virtual tree from a tree T with n nodes. The heart of the algorithm is a procedure BUILD(u) which finds fathers in the virtual tree for all nodes in the subtree T_u of T with root u. The result of BUILD is a queue Q of those nodes v in T_u such that $2 * \text{COUNT}[v] > \text{COUNT}[u]$.

A *queue* is a list of elements from which elements are removed from the front

and added to the rear; *front*(Q) is the first element of a queue Q.

```
procedure BUILD(u):
begin
    construct a list of nodes u_1, u_2, ..., u_k such that u_1 = u, u_k is a leaf, and
        u_{i+1} is a son of u_i with the largest count, for 1 ≦ i < k;
    Q ← u_k;
    for i = k − 1 step − 1 until 1 do
        begin
            while COUNT[u_i] ≧ 2 * COUNT[front (Q)] do
                begin
                    make u_i the father of front (Q) in the virtual tree;
                    delete front (Q) from Q
                end;
            add u_i to the rear of Q
        end
    for i = 1 until k − 1 do
        for each son v of u_i other than u_{i+1} do
            begin
                R ← BUILD(v);
                for each w on R do
                    make u_i the father of w in the virtual tree
            end;
    return Q
end
```

ALGORITHM 8 (Constructing the virtual tree).

1. Execute BUILD(u_0), where u_0 is the root of T.

2. For each node $v \neq u_0$ on the resulting queue, make u_0 the father of v in the virtual tree.

Example 3. After applying BUILD to node 7 of Fig. 3(a), Q contains nodes 8 and 7.

LEMMA 6. *Algorithm* 8 *requires* $O(n)$ *steps and correctly builds the virtual tree.*

Proof. For the linearity of the algorithm, it suffices to observe that BUILD takes time proportional to the number of nodes on the path $u_1, \cdots, u_k$ found in the first statement of BUILD, exclusive of recursive calls to itself. However, no node in T will be on the path created by two distinct calls of BUILD, except the first node of the path.

For the correctness of the algorithm, it is easy to determine that each node on the path $u_1, \cdots, u_k$ is given its correct father if that father is on T_u. Moreover, if v is a son of u_i and $v \neq u_{i+1}$, then COUNT[u_i] is no less than twice COUNT[v], so v's father in T_u is u_i. The same is clearly true of any descendant w of v remaining on the queue R when BUILD(v) is called. That is,

$$\text{COUNT}[u_i] \geqq 2 * \text{COUNT}[w] > \text{COUNT}[v]$$

We shall use the following strategy to simulate σ. After all LINK instructions in σ have been seen, we shall build the implied forest F and then from F a virtual forest V. Then, when we see an instruction LCA(u, v), we choose the one of u

and v having the smaller count, say u. We find the $\lfloor(\log n)/2\rfloor$th ancestor of u in V, say a. If $a = v$, LCA(u, v) is clearly v. If a is a proper ancestor of v in F, we repeat this procedure, finding the $\lfloor(\log n)/4\rfloor$th ancestor of u in V. If a is not an ancestor of v in V, we repeat the procedure, assuming the instruction was LCA(a, v) and beginning with the $\lfloor(\log n)/4\rfloor$th ancestor in V of the one of a and v having the smaller COUNT.

In log log n steps we shall converge upon a node a in V which is an ancestor of one of u and v in V. Since we are effectively following paths in F from u and v toward a root, and since we always move from the current ancestor of u or v having the smaller count, Lemma 5 guarantees us that a is the lowest common ancestor of u and v in F.

To implement this strategy, we shall use a procedure LOCATE(u, v, i, j) which finds the lowest common ancestor of u and v on F, given that

(a) the (2^i)th ancestor of u on V either does not exist or is an ancestor of v in F, and

(b) the (2^j)th ancestor of v in V either does not exist or is an ancestor of u in F.

In what follows, we assume that COUNT and HIGH refer to the implied forest F and ANCESTOR$[p, i]$ to the virtual forest V. We assume that this information has already been computed.

```
(1)  procedure LOCATE(u, v, i, j):
(2)  without loss of generality assume COUNT[u] ≦ COUNT[v]
(3)  otherwise (u, v, i, j) ← (v, u, j, i) in[5]
(4)      if i = 0 then return ANCESTOR[u, 0]
         else
             begin
(5)              a ← ANCESTOR[u, i − 1];
(6)              if v = a then return a
(7)              else if a = undefined or (v > a and HIGH[v] ≦ HIGH[a])
(8)                  then return LOCATE(u, v, i − 1, j)
(9)                  else return LOCATE(a, v, i − 1, j)
             end
```

We now summarize the entire algorithm.

ALGORITHM 9 (On-line execution of σ, assuming all LINK instructions in σ precede all LCA instructions).

1. As the LINK instructions are read, build the implied forest F in the obvious way.

2. When the first LCA instruction is encountered, do the following steps.

(a) For each tree in F, build a virtual tree by Algorithm 8. Call the resulting virtual forest V.

(b) Use the procedure INSTALL of § 4 to compute ANCESTOR$[u, i]$ for all nodes u and $0 \leqq i \leqq \lfloor\log(1 + \log n)\rfloor$.

(c) Preorder the nodes of F and compute COUNT$[u]$ and HIGH$[u]$ for all nodes u.

[5] This statement is a compile-time macro. See [1].

3. Now process each LCA instruction in turn. To compute LCA(u, v), we check whether one of u and v is a descendent of the other using the HIGH information. If so, the response is obvious. If not, we execute LOCATE(u, v, k, k), where $k = \lfloor \log(1 + \log n) \rfloor$ and print the result.

THEOREM 3. *Algorithm* 9 *correctly simulates* σ, *assuming that if the instruction* LCA(u, v) *is encountered*, u *and* v *are on the same tree.* (This condition can be checked in $O(n\alpha(n))$ time, as in Algorithm 7.)

Proof. The crux of the proof is showing that LOCATE works correctly. To do this, we shall show by induction on the sum $i + j$ that LOCATE(u, v, i, j) produces the lowest common ancestor of u and v, given that:

(a) the (2^i)th (resp. (2^j)th) ancestor of u (resp. v) in V is undefined or an ancestor of v (resp. u) in F, and

(b) COUNT[u] $\leqq$ COUNT[v],

(c) $i \geqq 0$ and $j \geqq 0$.

Basis. $i = j = 0$. Let f be the father of u in V, and let a be the lowest common ancestor of u and v. By hypothesis, f is an ancestor of v, and hence of a. By Lemma 5, f is a descendant of a, so $f = a$. Since line (4) makes the result of LOCATE be f in the case $i = 0$, we have the basis.

Inductive step. If $i = 0$, the argument is the same as for the basis. If $i \neq 0$, let a be the (2^{i-1})st ancestor of u in V, and let b be the lowest common ancestor of u and v. If $a = v$, then $b = v$ and this relationship is reflected in line (6) of LOCATE.

If a is a proper ancestor of v in F, or is undefined, then by the inductive hypothesis, LOCATE(u, v, $i - 1$, j) invoked on line (8) correctly produces b. If a is not an ancestor of v in F, then the lowest common ancestor of the pairs (u, v) and (a, v) are the same. Moreover, in F the (2^i)th ancestor of u is the (2^{i-1})st ancestor of a. Since a must be a descendant of b, and the (2^j)th ancestor of v, if it is defined, is an ancestor of b, it follows that the (2^j)th ancestor of v is an ancestor of a if it is defined. Thus the inductive hypothesis tells us that the result of LOCATE (a, v, $i - 1$, j) invoked on line (9) produces the correct result.

THEOREM 4. *Algorithm* 9 *operates in* $O(n \log \log n)$ *time.*

Proof. Steps 1, 2(a) and 2(c) are $O(n)$. Step 2(b) is $O(n \log \log n)$. Step 3 requires $O(n \log \log n)$ time since a call to LOCATE(u, v, k, k) can result in at most $2k + 1$ additional calls to LOCATE.

8. Dominators and reducible graphs. We shall now apply Algorithm 4 to a problem in code optimization. This section presents the basic definitions.

A *flow graph* is a triple $G = (N, E, u_0)$ where N is a finite set of nodes, E is a subset of $N \times N$ (the set of directed edges), and u_0 in N is the *initial node*. There is a path from u_0 to every node.

If each node has no more than two successors, we call G a *program* flow graph.

We say that node d *dominates* another node u if every path from u_0 *to* u passes through d. That is, if $u_0, u_1, \cdots, u_i$ is a path with $u_i = u$, then there exists an integer j, $0 \leqq j < i$ such that $u_j = d$. We say d *immediately dominates* u if d dominates u and every other dominator of u also dominates d. There are several interesting properties of the dominator and immediate dominator relations. The

following lemma is taken from [3].

LEMMA 7. (a) *Every node except the initial node has an immediate dominator.*

(b) *We may construct a tree (called the* dominator tree) *in which u is a son of d if and only if d immediately dominates u. The ancestors of u in the tree are precisely the dominators of u.*

Information about the dominator relation is useful for certain compiler code optimizations, such as those involving the detection of "loops". See [3], [4] for elaboration. By Lemma 7(b), the dominator information can be stored in a tree constructed knowing only the immediate dominators. If, as in [3], only a small number of dominators—the lowest ancestors in the tree—are used, we may not even need to construct the complete dominator relation.

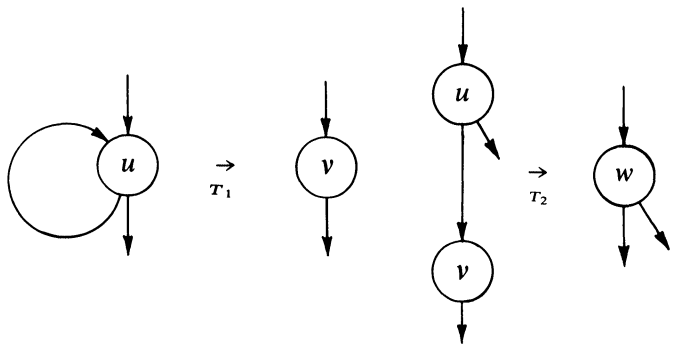
Algorithms to compute the dominator relation are given in [3] and [5]. Each requires $O(n^3)$ steps for program flow graphs, where n is the number of nodes in the graph. $O(n^2)$ algorithms for program flow graphs are found in [4] and [6], and these appear to be optimal, as it can take $O(n^2)$ time just to print the answer. To our knowledge, no one has developed a faster algorithm to compute only the immediate dominators. Here we do so for the important special case of reducible program flow graphs.

Reducible graphs were defined in [7]. They form a large class of graphs. For example, every rooted directed acyclic graph is reducible, and the flow graphs of gotoless programs are reducible. In fact, experiments have shown that the flow graphs of most programs written in practice are reducible. Moreover, any flow graph can be made reducible by judicious node splitting [8]. While this process could be expensive, those flow graphs which come "from nature" but which are not reducible readily yield to the node splitting technique. As a result, many code optimization algorithms such as

(i) eliminating common subexpressions [9], [10],
(ii) propagating constants [4],
(iii) eliminating useless definitions [4], and
(iv) finding active variables [11]

have been couched in terms of reducible flow graphs.

We shall give a definition of reducible flow graphs taken from [12]. This involves two transformations on directed graphs illustrated in Fig. 4 and defined



(a) *Transformation* T_1 (b) *Transformation* T_2

FIG. 4.

as follows:

T_1: Delete a loop.

T_2: Let node u be the lone predecessor of node v, where v is not the initial node. Merge u and v into a single node w. The predecessors of u become predecessors of w. The successors of u and v become successors of w. Note that w has a loop if there was formerly an edge to u from u or v. If u was the initial node, w becomes the new initial node. In this transformation we say v is *consumed*.

It is known that if T_1 and T_2 are applied to a given flow graph until no longer possible, a unique flow graph results. If this flow graph is a single node, we call the original graph *reducible*.

Example 4. Fig. 5 shows a sequence of reductions by T_1 and T_2. The initial node is u_0.

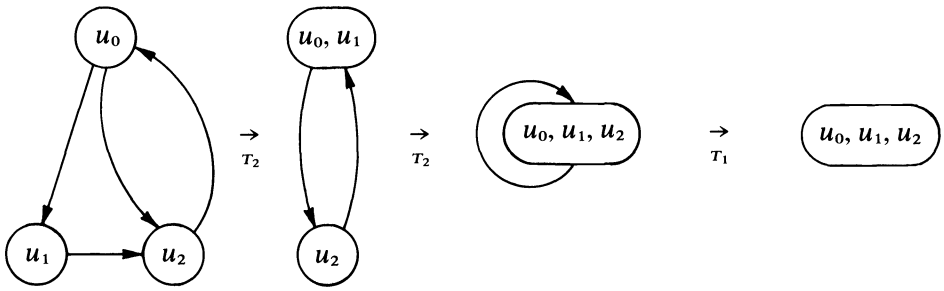


FIG. 5. *Reduction of flow graph.*

In the first step, T_2 is applied to u_1 with lone predecessor u_0. At the second step, T_2 is applied to u_2 with lone predecessor $\{u_0, u_1\}$. There is a loop introduced since $\{u_0, u_1\}$ is a successor of u_2.

A *region* R is a subset of the nodes of a flow graph such that there is a node h in R, called the *header*, having the property that every node in $R - \{h\}$ has all of its predecessors in R. Thus, the header dominates every other node in the region.

Example 5. Any node by itself is a region. In the previous example, $\{u_0, u_1\}$ is a region with header u_0, but $\{u_1, u_2\}$ is not since both u_1 and u_2 have a predecessor, u_0, outside the set.

Following [10], we may observe that as we reduce a flow graph by applying T_1 and T_2, each node of each successive graph *represents* a region in the following sense.

1. Initially, each node represents itself.
2. If we apply T_1 to a node, it continues to represent the same region as before.
3. If we apply T_2 when node u is the lone predecessor of node v, the resulting node represents the union of the regions represented by u and v. The header of the region represented by u is the header of the new region.

The following lemma is a restatement of the definitions of "dominator tree" and "region."

LEMMA 8. (a) *If u is a node in region R and u is not the header of R, then the immediate dominator of u is a member of R.*

(b) *If T_2 is applied to nodes u and v, with u the lone predecessor of v, and u and v represent regions R_u and R_v, respectively, then the immediate dominator of the header h of R_u is the lowest common ancestor (on the dominator tree of R_v) of the predecessors of h in the original graph.*

9. Dominators of a reducible flow graph. As a consequence of Lemma 8, the following algorithm may be used to construct the dominator tree for a reducible flow graph. The algorithm generates a sequence of LINK and LCA instructions for each reduction by T_2. These instructions will place u_0, the header of the region consumed by T_2, in its proper place on the dominator tree. Thus, if the flow graph is reducible, every node except the initial node will be the header of a region consumed by T_2, and its immediate dominator will be known. The details of the algorithm are as follows.

ALGORITHM 10 (Construction of dominator tree).

1. List the predecessors of each node of the original graph.

2. Use the algorithm of [13] to reduce the graph. Keep track of each region represented by the nodes of the "current" graph. Each time a node is consumed by T_2, create the sequence of instructions

$$\begin{aligned} &\mathrm{LCA}(u_1, u_2)\\ &\mathrm{LCA}(v_1, u_3)\\ &\quad\vdots\\ &\mathrm{LCA}(v_{k-2}, u_k)\\ &\mathrm{LINK}(u_0, v_{k-1}) \end{aligned}$$

and simulate them by Algorithm 4. Here u_0 is the header of the consumed region, $u_1, \cdots, u_k$ are all its predecessors, v_1 is the lowest common ancestor of u_1 and u_2, and v_i is the lowest common ancestor of v_{i-1} and u_{i+1} for $2 \leqq i < k$. If $k = 1$, the sequence is just $\mathrm{LINK}(u_0, u_1)$. (Note that neither Algorithm 7 nor 9 is sufficient. Direct on-line simulation is required.)

3. The desired dominator tree is the final A-forest (which must be a tree, since only one root, the initial node, remains).

THEOREM 5. *Algorithm* 10 *correctly constructs the dominator tree and requires $O(e \log e)$ steps on a flow graph with e edges.*

Proof. The correctness of the algorithm follows immediately from Lemma 8. By Lemma 8(a), the nodes of each region may be formed into a dominator tree with the header as root. By Lemma 8(b), when T_2 is applied, the dominator tree for the new region is created by making the header h of the consumed region a son of the lowest common ancestor of the predecessors of h in the original graph.

Step 1 requires $O(e)$ time. The reduction of the graph may be accomplished in $O(e\alpha(e))$ time by [13], and the sequence of instructions generated clearly has length $O(e)$. Thus step 2 requires $O(e \log e)$ time by Theorem 1. □

COROLLARY. *Algorithm* 10 *requires* $O(n \log n)$ *steps on an n-node program flow graph.*

Proof. An n-node program flow graph has no more than $2n$ edges. □

COROLLARY. *Algorithm* 10 *requires* $O(e \log e)$ *steps on a rooted directed acyclic graph with e edges.*

Proof. A rooted directed acyclic graph is reducible. □

After this paper was written, Tarjan [15] developed an $O(\max(n \log n, e))$ dominator algorithm for general graphs.

10. Conclusions. We have defined a problem that involves merging nodes into trees while retaining the ability to determine the lowest common ancestor of any two nodes. We have offered an $O(n \log n)$ algorithm to solve the problem on-line. We have shown how this algorithm provides a fast way of computing the dominator tree of a reducible flow graph. If an off-line solution is sufficient, the LINK-LCA problem can be solved in $O(n\alpha(n))$ steps. An on-line solution in the case where all LINK instructions precede all LCA instructions can be achieved in $O(n \log \log n)$ steps.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass, 1974.

[2] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, this Journal, 2 (1973), pp. 294–303.

[3] E. S. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[4] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling. Vol.* 2: *Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[5] M. SCHAEFER, *A Mathematical Theory of Global Flow Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[6] P. W. PURDOM AND E. F. MOORE, *Algorithm* 430: *Immediate predominators in a directed graph*, Comm. ACM, 15 (1972), pp. 777–778.

[7] F. E. ALLEN, *Control flow analysis*, SIGPLAN Not., 5 (1970), pp. 1–19.

[8] J. COCKE AND R. E. MILLER, *Some techniques for optimizing computer programs*, Proc. 2nd Internat. Conf. on System Sciences, Honolulu, Hawaii, 1969.

[9] J. COCKE, *Global common subexpression elimination*, SIGPLAN Not., 5 (1970), pp. 20–24.

[10] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, Acta Informatica, 2 (1973), pp. 191–213.

[11] K. KENNEDY, *A global flow analysis algorithm*, Internat. J. Comput. Math., 3 (1971), pp. 5–16.

[12] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

[13] R. E. TARJAN, *Testing flow graph reducibility*, J. Comput. System Sci., 9 (1974), pp. 355–365.

[14] ———, *On the efficiency of a good but not linear disjoint set union algorithm*, J. Assoc. Comput. Mach., 22 (1975). pp. 215–225.

[15] ———, *Finding dominators in directed graphs*, this Journal, 3 (1974), pp. 62–89.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

A FAST ALGORITHM FOR FINDING AN OPTIMAL ORDERING FOR VERTEX ELIMINATION ON A GRAPH*

TATSUO OHTSUKI†

Abstract. This paper gives a graph-theoretic approach to the problem of finding an optimal ordering for Gaussian elimination on a sparse matrix. A set of new "fill-ins" produced by Gaussian elimination on a matrix A is characterized by a *triangulation* induced by *vertex elimination* on a graph associated with A. A triangulation T is *minimal* (*minimum*) if there exists no triangulation $\hat{T}$ such that $\hat{T} \subset\subset T(|\hat{T}| < |T|)$, where $\subset\subset$ denotes the strict inclusion, and an ordering α is *optimal* (*optimum*) if a minimal (minimum) triangulation is generated by α. An optimum ordering is necessarily optimal but not conversely. An efficient algorithm for finding an optimal ordering in $O(M \cdot N)$ time is presented, where M is the number of vertices and N is the number of edges of the graph being considered.

Key words. algorithm, sparse matrix, vertex elimination, optimal ordering, triangulated graph, minimal triangulation, backward ordering scheme, search

1. Introduction. The problem of solving a large sparse system of linear equations arises in widespread applications. Since, in many cases, coefficient matrices have a fixed sparseness structure in the course of entire computation, it is crucial to a priori find a good pivoting order for Gaussian elimination on such matrices.

Whenever the coefficient matrix is structurally symmetric and diagonally dominant, the pivoting procedure can be characterized by the vertex elimination process on an undirected graph [1]. Recently, Rose introduced the concept of "triangulated graph" [2] and showed that an optimal pivoting order of a matrix, which produces a minimal set of fill-ins, can be obtained from a minimal triangulation of the graph associated with it [3]. This pioneering work is extended by Ohtsuki, Cheung and Fujisawa, who first gave a systematic method of finding a minimal triangulation, i.e., optimal vertex elimination ordering of a graph [4]. The word "optimal" should not be mistaken to mean "optimum" as in some engineering literatures. In terms of a set, "optimal" refers to a set property, whereas "optimum" refers to the size of the set.

It turns out, however, that the algorithm presented in [4] requires $O(M^2N)$ time for finding an optimal ordering, where M is the number of vertices and N is the number of edges of the graph being considered. This paper presents a new algorithm for finding an optimal ordering, in which only $O(M \cdot N)$ time and $O(M, N)$ memory space are required. The efficiency of the algorithm presented is based on the following strategy: *Suppose the vertices of a graph are to be eliminated in the order* $v_1, v_2, \cdots, v_M$. *Then the algorithm determines* v_M *first,* v_{M-1} *next, and so forth.* This "backward ordering scheme" with extensive use of the "search" technique of graphs [5] gives the remarkable improvement in running time.

2. Preliminaries. For our purpose a *graph* is a pair $G = (V, E)$, where V is a finite set of $M = |V|$ elements called *vertices* and

$$E \subset \{(u, v) | u, v \in V \text{ and } u \neq v\}$$

* Received by the editors December 20, 1974, and in revised form April 8, 1975.
† Central Research Laboratories, Nippon Electric Co., Ltd., Kawasaki, Japan.

is a set of $N = |E|$ unordered vertex pairs called *edges*. In order to avoid trivialities, we shall assume that the graph G is connected.

Given a subset $A \subset V$, the set

$$\mathrm{Adj}\,(A) = \{b \in V - A | (a, b) \in E \text{ for some } a \in A\}$$

is the set of vertices *adjacent* to A. If A consists of a single vertex a, the abbreviation $\mathrm{Adj}\,(a)$ will be used instead of $\mathrm{Adj}\,(\{a\})$. A *clique* of a graph is a subset of vertices which are pairwise adjacent. For a subset $A \subset V$, the *section graph* $G(A)$ is the subgraph

$$G(A) = (A, E(A)), \qquad E(A) = \{(u, v) \in E | u, v \in A\}.$$

A *separator* of a graph $G = (V, E)$ is a subset $S \subset V$ such that the section graph $G(V - S)$ consists of two or more *connected components*.

For a pair of distinct vertices $a, b \in V$, an a, b *chain* C is an ordered set of vertices

$$C = \{v_1, v_2, \cdots, v_L\}$$

such that $v_1 = a$, $v_L = b$ and $(v_l, v_{l+1}) \in E$; $l = 1, 2, \cdots, L - 1$. An a, b *separator* $S \subset V - \{a, b\}$ is a separator such that a and b are in distinct components of $G(V - S)$.

Given a vertex $v \in V$ of a graph $G = (V, E)$, the *vertex elimination* of v is the operation of (i) deleting v and its incident edges and (ii) adding edges so that $\mathrm{Adj}\,(v)$ becomes a clique. Consider, for a subset $A \subset V$, the graph obtained from G by successively eliminating all the vertices in $V - A$. It is clear that the result is independent of order in which the vertices are eliminated. The resultant graph is denoted by $G\langle A\rangle$ and called the *elimination graph* determined by A.

For a graph $G = (V, E)$ with $|V| = M$, an *ordering* of V is a bijective map

$$\alpha : \{1, 2, \cdots, M\} \to V.$$

Thus α indicates that the vertices are eliminated in the order $\alpha(1), \alpha(2), \cdots, \alpha(M)$. Let $E^{(i)}$; $i = 1, 2, \cdots, M$, be the set of edges of the elimination graph

$$G\langle\{\alpha(m)\}_{m=i}^{M}\rangle.$$

Then the set

$$\mathrm{Trg}\,(G\,;\alpha) = \bigcup_{i=1}^{M} E^{(i)} - E$$

is the set of edges added in the entire course of vertex elimination corresponding to the ordering α. The supergraph of G obtained by adding edges of $\mathrm{Trg}\,(G\,;\alpha)$ is denoted by $G[\alpha]$, i.e.,

$$G[\alpha] = (V, E \cup \mathrm{Trg}\,(G\,;\alpha)).$$

For a graph $G = (V, E)$ which is not necessarily triangulated, consider its supergraph $\hat{G} = (V, E \cup F)$; $E \cap F = \varnothing$. Then the set F is called a *triangulation* of G if $\hat{G}$ is triangulated. Rose [3] has shown that $G[\alpha]$ is triangulated for any ordering α. Thus $\mathrm{Trg}\,(G\,;\alpha)$ is called the *triangulation induced by* α. A triangulation $\hat{F}$ is said to be *minimal* if there exists no triangulation F such that $F \subset\subset \hat{F}$, where $\subset\subset$ denotes the strict inclusion. Similarly, an ordering $\hat{\alpha}$ is said to be *optimal* for

G if there exists no ordering α such that $\mathrm{Trg}(G;\alpha) \subset \subset \mathrm{Trg}(G;\hat{\alpha})$. Concerning minimal triangulation and optimal ordering, Ohtsuki, Cheung and Fujisawa [4] have obtained the following results.

THEOREM A. *A triangulation F of a graph $G = (V, E)$ is minimal if and only if there exists an optimal ordering α such that* $\mathrm{Trg}(G;\alpha) = F$.

THEOREM B. *A triangulation F of $G = (V, E)$ is minimal if and only if, for each $(x, y) \in F$, there exists no x, y separator S of G such that S is a clique of $\hat{G} = (V, E \cup F)$.*

THEOREM C. *Let $x \in V$ be a vertex of a graph $G = (V, E)$. Then there exists an optimal ordering α such that $\alpha(1) = x$ if and only if, for each pair of distinct vertices u, $v \in \mathrm{Adj}(x)$, there exists a u, v chain in the section graph $G(V - S)$, where $S = \{x\} \cup (\mathrm{Adj}(x) - \{u, v\})$.*

3. Backward ordering scheme. The following result, due to Rose [6, p. 198], plays a fundamental role in the proposed optimal ordering algorithm.

THEOREM D. *For any clique C of a graph $G = (V, E)$ with $M = |V|$, there exists an optimal (optimum) ordering*

$$\alpha : \{1, 2, \cdots, M\} \to V$$

for G such that the vertices of C are ordered last.

As a corollary of Theorem D, it is seen that, *for any vertex $v \in V$, there exists an optimal ordering α such that $\alpha(M) = v$.* It should be noted in comparison that there may not exist an optimal ordering α such that $\alpha(1) = v$ for a given vertex v. Thus, if we are to determine vertices in the order $\alpha(1), \alpha(2), \cdots, \alpha(M)$, a nontrivial operation to select a vertex in each elimination step must be involved. This consideration highlights the "backward ordering scheme", i.e., the strategy to determine vertices in the order $\alpha(M), \alpha(M-1), \cdots, \alpha(1)$. In this respect, the present algorithm is quite unlike the one given in [4], and unlike any existing heuristic ordering schemes such as *minimum degree scheme* and *minimum valency scheme* [7].

In order to establish the "backward ordering scheme", we shall first of all generalize Theorem D as follows.

DEFINITION 1. Let P be a proper subset of V for a graph $G = (V, E)$ and $G(X)$ be a connected component of $G(V - P)$. Then P is said to satisfy *Condition $\mathscr{A}$ with respect to $G(X)$* (denoted by $P \in \mathscr{A}_X(G)$) if there exists a u, v chain in $G(V - X - \mathrm{Adj}(X) \cup \{u, v\})$ for any distinct vertices u, $v \in \mathrm{Adj}(X)$. Furthermore, P is simply said to satisfy *Condition $\mathscr{A}$* (denoted by $P \in \mathscr{A}(G)$) if the condition holds with respect to every connected component of $G(V - P)$.

Remark. From the inherent property of the vertex elimination process, it is clear that

$$P \in \mathscr{A}(G) \Rightarrow P \in \mathscr{A}(G\langle \hat{P} \rangle),$$

for any elimination graph $G\langle \hat{P} \rangle$ of G such that $P \subset \hat{P} \subset V$.

THEOREM 1. *For a graph $G = (V, E)$, let P be a proper subset of V such that $P \in \mathscr{A}(G)$. Then there exists an optimal ordering of V such that the vertices of P are ordered last.*

Proof. Let $G(X_k), k = 1, 2, \cdots, K$, be the connected components of $G(V - P)$ and $\hat{G} = (V, E \cup F)$ be the supergraph of G obtained by adding the minimum set of edges so that $\mathrm{Adj}(X_k)$ becomes a clique for each $k = 1, 2, \cdots, K$. Furthermore,

let $G_0 = \hat{G}(P)$ and $G_k = \hat{G}(X_k \cup \mathrm{Adj}(X_k))$, $k = 1, 2, \cdots, K$. Since $\mathrm{Adj}(X_k)$ is a clique of G_k, Theorem D implies that there exists an optimal ordering of $X_k \cup \mathrm{Adj}(X_k)$ for G_k such that the vertices of $\mathrm{Adj}(X_k)$ are ordered last. Let β_k, $k = 1, 2, \cdots, K$, be such an ordering. Furthermore, let β_0 be an optimal ordering of P for G_0. We shall prove the theorem by showing that the ordering α of V is optimal for G, where α is determined as follows:

$$\alpha(i) = \beta_1(i), \qquad i = 1, 2, \cdots, |X_1|,$$

$$\alpha\left(\sum_{j=1}^{k-1} |X_j| + i\right) = \beta_k(i), \qquad i = 1, 2, \cdots, |X_k|, \quad k = 2, 3, \cdots, K,$$

$$\alpha\left(\sum_{j=1}^{K} |X_j| + i\right) = \beta_0(i), \qquad i = 1, 2, \cdots, |P|.$$

The triangulation of G induced by α is given by

$$\mathrm{Trg}(G\,;\alpha) = \bigcup_{k=0}^{K} \mathrm{Trg}(G_k\,;\beta_k) \cup F.$$

Due to Theorem A, it suffices for the proof to show that $\mathrm{Trg}(G\,;\alpha)$ is a minimal triangulation of G. Assume that $\mathrm{Trg}(G\,;\alpha)$ is not minimal. Then as Theorem B says, there exists, for at least one vertex pair $(x, y) \in \mathrm{Trg}(G\,;\alpha)$, an x, y separator S of G such that S is a clique of $G[\alpha]$. We have three cases to consider.

Case 1. $(x, y) \in F$. Without loss of generality, we assume that $x, y \in \mathrm{Adj}(X_1)$. Then S must contain a vertex (say z) in X_1, since $G(X_1)$ is connected, and another vertex (say w) in $V - X_1 - \mathrm{Adj}(X_1)$, since there exists an x, y chain in $G(V - X_1 - \mathrm{Adj}(X_1) \cup \{x, y\})$. Furthermore, $(z, w) \in \mathrm{Trg}(G\,;\alpha)$ since S is a clique of $G[\alpha]$. This contradicts the property of ordering α.

Case 2. $(x, y) \in \mathrm{Trg}(G_k\,;\beta_k)$, $k \in \{1, 2, \cdots, K\}$. According to Theorem A, $\mathrm{Trg}(G_k\,;\beta_k)$ is a minimal triangulation of G_k. Also, there exists no pair of distinct vertices $u, v \in \mathrm{Adj}(X_k)$ such that S is a u, v separator of G, since otherwise it causes the same contradiction as in Case 1. Therefore $S \cap (\mathrm{Adj}(X_k) \cup X_k)$ is an x, y separator of G_k and, simultaneously, is a clique of $G_k[\beta_k]$. Then, as Theorem B says, $\mathrm{Trg}(G_k\,;\beta_k)$ cannot be a minimal triangulation of G_k, which is a contradiction.

Case 3. $(x, y) \in \mathrm{Trg}(G_0\,;\beta_0)$. This case also leads us to a contradiction with an obvious modification of Case 2.

Now we have shown that α is an optimal ordering of V for G.

Remark. This theorem can be viewed as a generalization of the "if" part of Theorem C.

The basic idea of the proposed algorithm for obtaining an optimal ordering of V for a given graph $G = (V, E)$ is to decompose V into a number of disjoint subsets $V^0, V^1, \cdots, V^I$ in a sequence such that

$$\bigcup_{j=0}^{i} V^j \in \mathscr{A}(G), \qquad i = 0, 1, \cdots, I - 1.$$

Then, as Theorem 1 indicates, the sequence provides us with a set of optimal orderings such that the vertices in V^I are ordered first, those in V^{I-1} next, and so forth. Here we confront another problem, that is, to determine the local

orderings of V^j's which constitute an optimal ordering of V for G. To solve the problem, we shall present a scheme for decomposing V in such a way that the local ordering of each V^j can be determined arbitrarily.

DEFINITION 2. Let P and Q be disjoint subsets of V for a graph $G = (V, E)$. Then the ordered pair (P, Q) is said to satisfy *Condition* $\mathscr{B}$ (denoted by $(P, Q) \in \mathscr{B}(G)$) if $(u, v) \in \hat{E}$ for any distinct vertices u and v such that $u \in Q$ and $v \in Q \cup R$, where $\hat{E}$ is the set of edges of $G\langle P \cup Q\rangle$ and R is the set of vertices adjacent to Q in $G\langle P \cup Q\rangle$.

Remark. It is clear that

$$(P, Q) \in \mathscr{B}(G) \Rightarrow (P, Q) \in \mathscr{B}(G\langle U\rangle)$$

for any elimination graph $G\langle U\rangle$ of G such that $P \cup Q \subset U \subset V$.

Suppose the set of vertices V of a graph $G = (V, E)$ has been decomposed into disjoint subsets $V^0, V^1, \cdots, V^I$ which have the following properties:

(i) V^0 is a clique of G (of course, $V^0 \in \mathscr{A}(G)$);

(ii) $\bigcup_{j=0}^{i} V^j \in \mathscr{A}(G), i = 1, 2, \cdots, I - 1$;

(iii) $(\bigcup_{j=0}^{i-1} V^{(j)}, V^{(i)}) \in \mathscr{B}(G)$, $i = 1, 2, \cdots, I$ (this implies that V^I is also a clique of G).

Then the problem of obtaining an optimal ordering is easy. To be precise, we can simply order the vertices in V^I first, those in V^{I-1} next, $\cdots$, those in V^0 last, while the local ordering of each $V^j, j = 0, 1, \cdots, I$, is arbitrary. Note that property (iii) above implies that the local orderings make no difference in the triangulation of G.

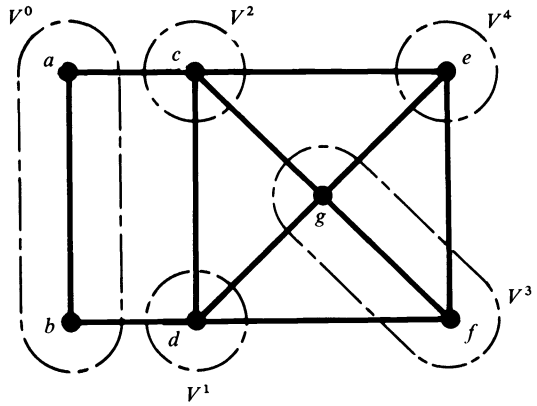


FIG. 1. *Backward ordering scheme*

Example 1. Consider the graph $G = (V, E)$ shown in Fig. 1, and decompose $V = \{a \sim g\}$ into five disjoint subsets: $V^0 = \{a, b\}$, $V^1 = \{d\}$, $V^2 = \{c\}$, $V^3 = \{f, g\}$, $V^4 = \{e\}$. Then the ordered sequence $\{V^0, V^1, V^2, V^3, V^4\}$ satisfies the desired properties and determines optimal orderings as $\{\alpha(6), \alpha(7)\} = \{a, b\}$, $\alpha(5) = d$, $\alpha(4) = c$, $\{\alpha(2), \alpha(3)\} = \{f, g\}$ and $\alpha(1) = e$. The corresponding minimal triangulation is

$$\mathrm{Trg}\,(G; \alpha) = \{(c, f), (a, d)\}.$$

The actual procedure to decompose the set of vertices V of a graph $G = (V, E)$ into the desired form is initiated by taking an arbitrary clique, which is assigned for V^0. The key problem to determine $V^1, V^2, \cdots, V^I$ one after another is as follows: *for given set* $P \subset V$ *such that* $P \in \mathscr{A}(G)$, *find another set* $Q \subset V - P$ *such that* (i) $P \cup Q \in \mathscr{A}(G)$ *and* (ii) $(P, Q) \in \mathscr{B}(G)$.

The following theorems combined lead to a systematic method to find such a set Q. In the case where a vertex q of a connected component $G(X)$ of $G(V - P)$ is adjacent to all the vertices in $\mathrm{Adj}(X)$, Theorem 2 below says that $Q = \{q\}$ is a desired set. Whenever this is not the case, Theorem 4 below guarantees the existence of a subset Q of $V - P$ satisfying all the conditions of Theorem 3 below, which says that Q possesses the desired property.

THEOREM 2. *For a graph* $G = (V, E)$, *let* P *be a proper subset of* V *such that* $P \in \mathscr{A}(G)$, $G(X)$ *be a connected component of* $G(V - P)$ *and* q *be a vertex in* X. *If* $\mathrm{Adj}(X) = \mathrm{Adj}(q) \cap P$, *then* (i) $P \cup \{q\} \in \mathscr{A}(G)$ *and* (ii) $(P, \{q\}) \in \mathscr{B}(G)$.

Proof. The section graph $G(V - P - \{q\})$ consists of connected components of $G(V - P)$, exclusive of $G(X)$, and those of $G(X - \{q\})$. With respect to a connected component $G(X')$ of $G(V - P - \{q\})$ of the former type, it is clear that $q \notin \mathrm{Adj}(X')$ since $G(X)$ is a connected component of $G(V - P)$. Thus the assumption that $P \in \mathscr{A}_{X'}(G)$ implies that $P \cup \{q\} \in \mathscr{A}_{X'}(G)$. Let $G(Y)$ be a connected component of $G(V - P - \{q\})$ of the latter type. Then $\mathrm{Adj}(Y) \subset \{q\} \cup \mathrm{Adj}(X)$ and, furthermore, the assumption that $\mathrm{Adj}(X) = \mathrm{Adj}(q) \cap P$ implies that any vertex $p \in \mathrm{Adj}(X)$ is connected to q by an edge, i.e., by the simplest p, q chain. Therefore $P \cup \{q\} \in \mathscr{A}_Y(G)$ as long as $P \in \mathscr{A}_X(G)$. Hence part (i) has been proved.

Part (ii) is obvious since the set $\{q\}$ consists of a single vertex.

THEOREM 3. *For a graph* $G = (V, E)$, *let* P *and* $G(X)$ *be as in Theorem* 2. *Furthermore, let* Q *be a proper subset of* X, S *be a nonempty subset of* $\mathrm{Adj}(X)$ *and* $G(Y)$ *be a connected component of* $G(X - Q)$. *If*

(a) $(q, s) \in E, \forall q \in Q$ *and* $\forall s \in S$,

(b) $Q \subset \mathrm{Adj}(Y)$,

(c) $\mathrm{Adj}(Y) \cap S = \varnothing$ *and*

(d) $\mathrm{Adj}(Y) \cap P \cup S = \mathrm{Adj}(X)$,

then (i) $P \cup Q \in \mathscr{A}(G)$ *and* (ii) $(P, Q) \in \mathscr{B}(G)$.

Proof. All the vertices in $\mathrm{Adj}(Y)$ are either in P or Q. Then, as long as $P \in \mathscr{A}_X(G)$, it follows from (a) and (c) that $P \cup Q \in \mathscr{A}_Y(G)$. Let $G(Y')$, if any, be another connected component of $G(X - Q)$, and consider a pair of distinct vertices u, $v \in \mathrm{Adj}(Y')$. The existence of a u, v chain in $G(V - Y' - \mathrm{Adj}(Y') \cup \{u, v\})$ can be confirmed in the same way as with respect to $G(Y)$ whenever $u, v \in P$ or whenever at least one of $\{u, v\}$ is in S. Thus we can assume that one of $\{u, v\}$, say u, belongs to Q and the other vertex v belongs to either P or Q. Since Y' is a subset of X, (b) and (d) imply that $u, v \in \mathrm{Adj}(Y)$, while the connectivity of $G(Y)$ guarantees the existence of a desired u, v chain in $G(Y \cup \{u, v\})$. Hence $P \cup Q \in \mathscr{A}_{Y'}(G)$. Let $G(X')$, if any, be a connected component of $G(V - P)$ exclusive of $G(X)$. It is clear that $Q \cap \mathrm{Adj}(X') = \varnothing$. Thus the assumption that $P \in \mathscr{A}_{X'}(G)$ automatically implies that $P \cup Q \in \mathscr{A}_{X'}(G)$. Now part (i) has been proved.

Let $\hat{E}$ be the set of edges of the elimination graph $G\langle P \cup Q\rangle$ and W be the set of vertices adjacent to Q in $G\langle P \cup Q\rangle$. Then, as long as $Q \subset X$, any vertex in W is adjacent to X in the original graph G. Also it follows from (d) that W

$= \mathrm{Adj}(Y) \cap P \cup S$. Let (u, v) be a pair of distinct vertices such that $u \in Q$ and $v \in Q \cup W$. When $v \in S$, (a) implies that $(u, v) \in E$, i.e., $(u, v) \in \hat{E}$. When $v \in Q \cup W - S$, (b) and connectivity of $G(Y)$ imply that $(u, v) \in \hat{E}$. Hence $(P, Q) \in \mathscr{B}(G)$, i.e., part (ii) has also been proved.

LEMMA 1. *For a graph $G = (V, E)$, let P, $G(X)$, Q, S and $G(Y)$ be as in Theorem 3. Suppose* (a)–(c) *of Theorem* 3 *as well as*

(e) $\mathrm{Adj}(Y) \cap P \cup S \subset \subset \mathrm{Adj}(X)$

are satisfied. Then there exists a nonempty subset Q^* of $X - Y$, i.e., proper subset of X, and a nonempty subset S^* of $\mathrm{Adj}(X)$ *such that*

(a)* $(q, s) \in E$, $\forall q \in Q^*$ and $\forall s \in S^*$,

(b)* $Q \subset \mathrm{Adj}(Y^*)$,

(c)* $\mathrm{Adj}(Y^*) \cap S^* = \varnothing$ and

(e)* $\mathrm{Adj}(Y) \cap P \cup S \subset \subset \mathrm{Adj}(Y^*) \cap P \cup S^*$,

where $G(Y^*)$ is the connected component of $G(X - Q^*)$ including the vertices of $G(Y)$.

Proof. The algorithm which will be described in § 4 provides a constructive way of finding Q^*, S^* and Y^*. This is essentially a proof of the lemma.

THEOREM 4. *For a graph $G = (V, E)$, let P be a proper subset of V and $G(X)$ be a connected component of $G(V - P)$. Then there exists a proper subset Q of X which, for some nonempty subset S of* $\mathrm{Adj}(X)$ *and for some connected component $G(Y)$ of $G(X - Q)$, satisfies* (a)–(d) *of Theorem* 3, *if there exists a vertex $y \in X$ such that* $\mathrm{Adj}(y) \cap P \subset \subset \mathrm{Adj}(X)$.

Proof. Let p be a vertex in $\mathrm{Adj}(X) - \mathrm{Adj}(y) \cap P$. We set $S = \{p\}$ and

$$Q = \{x \in \mathrm{Adj}(p) \cap X \mid \text{there exists an } x, y \text{ chain in } G(X - \mathrm{Adj}(p) \cup \{x\})\}.$$

Since $G(X)$ is connected, the set Q must be nonempty. Now let $G(Y)$ be the connected component of $G(X - Q)$ which contains y. Then it is clear that (a)–(c) of Theorem 3 are satisfied. We assume that (d) of Theorem 3 is not satisfied, which implies that (e)* of Lemma 1 is satisfied. Then we can take another trio of a proper subset Q^* of X, a nonempty subset S^* of $\mathrm{Adj}(X)$ and a connected component $G(Y^*)$ of $G(X - Q^*)$ so that (a)*–(e)* of Lemma 1 are satisfied. Since the given graph G is finite, we can obtain, by repeated use of Lemma 1, a desired trio of Q, S and $G(Y)$ which satisfies (d) of Theorem 3.

Remark. Theorems 3 and 4 and Lemma 1 combined suggest a recursive procedure for finding a desired set Q of vertices.

4. A search technique. Suppose we have obtained a proper subset $P \subset V$ of vertices of a given graph $G = (V, E)$ such that $P \in \mathscr{A}(G)$, and we are to obtain another subset $Q \subset V - P$ such that (i) $P \cup Q \in \mathscr{A}(G)$ and (ii) $(P, Q) \in \mathscr{B}(G)$. In this section we shall present a specific search technique for obtaining such a set Q.

Initially all the vertices of G are considered to be "unreached". We start a search from some vertex, say $v_0 \in V$. The unit step, say kth step, of a search is to choose an edge $e_k = (v_i, v_k) \in E$, $i < k$, which connects a vertex v_i already "reached" with a new vertex v_k. Once a new vertex has been found, we mark it "reached". Such a process is called a *search* of G [5].

For our purpose, a search of G is started from some vertex, say v_0, in $\mathrm{Adj}(V - P)$, and then an edge $e_1 = (v_0, v_1)$ such that $v_1 \in V - P$ is selected. It

should be noted that v_1 identifies a connected component, say $G(X)$, of $G(V - P)$. The search is continued subject to the following edge selection rule.

Rule 1. Let $e_k = (v_i, v_k)$, $i < k$, be an edge to be selected. Then the vertex v_i already "reached" must be taken from those in X, while the new vertex v_k must be taken from those in P whenever it is possible. An edge $e_k = (v_i, v_k)$ with $v_k \in X$ may be selected only when all the vertices in $\mathrm{Adj}(v_i) \cap P$ are "reached". This rule is imposed until a new vertex in P is "reached". Note that this procedure generates a chain $\{v_1, v_2, \cdots, v_L = p\}$ joining v_1 and some "unreached" vertex $p \in P$ such that (i) $v_l \in X$, $l = 1, 2, \cdots, L - 1$, and (ii) for any "unreached" vertex $q \in P$, $(q, v_l) \notin E$, $l = 1, 2, \cdots, L - 2$.

Once a new vertex $p \in P$ is "reached," the set of vertices in $\mathrm{Adj}(p)$ is identified and the search is continued subject to the following edge selection rule instead of Rule 1.

Rule 2. Let $e_k = (v_i, v_k)$, $i < k$, be an edge to be selected. Then the vertex v_i already "reached" must be taken from those in $X - \mathrm{Adj}(p)$, while the new vertex v_k can be either in X or in P. This rule is imposed until no more edges can be selected. Let $G(Y)$; $v_1 \in Y$ be the connected component of $G(X - \mathrm{Adj}(p))$. Then this procedure finds all the "unreached" vertices in $Y \cup \mathrm{Adj}(Y)$.

Let T be the set of vertices which have been "reached" by the time when the restriction of Rule 2 is expired. Then the set $Y = T - P - \mathrm{Adj}(p)$ has the key property that $G(Y)$ is a connected component of $G(X - \mathrm{Adj}(p))$. Once Rule 2 is expired, the search is continued applying Rule 1 again. In general, our search is processed by alternating use of Rule 1 and Rule 2. The whole process is terminated when the search subject to Rule 1 finds no new vertex in P, i.e., when all the vertices in $X \cup \mathrm{Adj}(X)$ have been "reached".

Now we shall show how a desired set Q is obtained by means of the specific search described above. Let $e_1 = (v_0, v_1)$, with $v_0 \in P$ and $v_1 \in V - P$, be the first edge selected in the search. If v_1 is adjacent to another vertex, say v_2, in P, the edge $e_2 = (v_1, v_2)$ is selected next due to Rule 1. Then Rule 1 is replaced by Rule 2. But it yields no edge to be selected, so Rule 1 is again applied. In this way, the vertices in $\mathrm{Adj}(v_1) \cap P$ are found first. When all these vertices are found, the search is continued by applying Rule 1. If no new vertex in P can be "reached," then v_1 is adjacent to all the vertices in $\mathrm{Adj}(X)$. Thus $Q = \{v_1\}$ is a desired set as Theorem 2 says.

When a new vertex $p \in P$ is "reached," the vertices in $\mathrm{Adj}(p)$ are identified, and the search is further continued subject to Rule 2. Consider the stage when no new edge can be selected in this phase of the search. Let T be the set of vertices which have been "reached" by that time, Y be the set defined by $Y = T - P - \mathrm{Adj}(p)$, and let Q be the set defined by $Q = T \cap \mathrm{Adj}(p) - P$. Then it follows from the key property of our search that $G(Y)$ is a connected component of $G(X - Q)$ such that $Q \subset \mathrm{Adj}(Y)$. And it is clear that $p \notin \mathrm{Adj}(Y)$. Hence the set Q satisfies conditions (a)–(c) of Theorem 3 for $S = \{p\}$.

Now condition (d) of Theorem 3 is also satisfied if $T \cap P$ contains all the vertices in $\mathrm{Adj}(X)$. Otherwise a new vertex in P can be "reached" by continuing the search subject to Rule 1. This phase of the search starts from some vertex in Q, since no vertex in Y is adjacent to new vertices, and finally finds a new vertex, say s, in P. Then the vertices in $\mathrm{Adj}(s)$ are identified instead of those in $\mathrm{Adj}(p)$, and the search is continued subject to Rule 2 until no new edge can be selected. Let

T^* be the set of vertices which have been "reached" by that time. There are two cases to be considered.

Case 1. $\mathrm{Adj}(s) \cap Q = Q$. In this case, it is clear that $T^* = T \cup \{s\}$. Hence conditions (a)*–(e)* of Lemma 1 are satisfied if $S^* = S \cup \{s\}$, $Q^* = Q$ and $Y^* = Y$.

Case 2. $\mathrm{Adj}(s) \cap Q \subset\subset Q$. Let $Y^* = T^* - P - \mathrm{Adj}(s)$ and $Q^* = T^* \cap \mathrm{Adj}(s) - P$. Then $G(Y^*)$ is the connected component of $G(X - Q^*)$ such that $Q^* \subset \mathrm{Adj}(Y^*)$. It is also clear that $s \notin \mathrm{Adj}(Y^*)$. Thus the set Q^* satisfies conditions (a)*–(c)* of Lemma 1 for $S^* = \{s\}$. Furthermore, it follows from the assumption that $S \subset \mathrm{Adj}(Y^*)$, which implies that $\mathrm{Adj}(Y) \cap P \cup S \subset \mathrm{Adj}(Y^*) \cap P$. Hence condition (e)* is also satisfied.

As described above, the search enables us to obtain a proper subset $Q^* \subset\subset X$, which satisfies, for some nonempty subset $S^* \subset \mathrm{Adj}(X)$, all the conditions of Lemma 1 whenever $\mathrm{Adj}(Y) \cap P \cup S \subset\subset \mathrm{Adj}(X)$. Repeating this way, we can finally obtain a desired set Q which satisfies all the conditions of Theorem 3.

Example 2. As shown in Fig. 2, consider a proper subset $P \subset V$ of vertices of a graph $G = (V, E)$, where $a, c, f, j, h \in P$,

Figure 2(a): Initially we choose an edge (a, b); then b identifies a connected component $G(X)$ of $G(V - P)$, where $X = \{b, d, e, g, i, k, l\}$. By means of the early part of the search, we find all the vertices in $\mathrm{Adj}(b) \cap P$, which in this case is $\{a, c\}$.

Figure 2(b): The search applying Rule 1 finds a new vertex $f \in P$.

Figure 2(c): The vertices $e, g \in \mathrm{Adj}(f)$ are identified and the search applying Rule 2 determines the set $Q = \{e, g\}$, which satisfies conditions (a)–(c) of Theorem 3 for $S = \{f\}$ and $G(Y)$, $Y = \{b, d\}$.

Figure 2(d): The search applying Rule 1 finds a new vertex $h \in P$.

Figure 2(e): The vertices $g, k, l \in \mathrm{Adj}(h)$ are identified, and the search applying Rule 2 determines the set $Q^* = \{g, k\}$, which satisfies conditions (a)*–(c)* of Lemma 1 for $S^* = \{h\}$ and $G(Y^*)$; $Y^* = \{b, d, e, i\}$.

Figure 2(f): The search applying Rule 1 finds no new vertex in P. Therefore it is seen that Q^* satisfies all the conditions of Theorem 3 for S^* and $G(Y^*)$.

5. Computer algorithm. In this section we shall give an algorithm based on the search technique described in the previous section.

A given graph $G = (V, E)$ with $|V| = M$ and $|E| = N$ is represented as a set of lists of vertices. Each list consists of the set of vertices which are adjacent to a single vertex. The set of lists are packed in a one-dimensional array $E(\cdot)$ of length $2N$. A one-dimensional array $V(\cdot)$ of length M is used to locate, for each vertex, the corresponding lists in $E(\cdot)$.

Let P be a given proper subset of V. Then the vertices in P must be distinguished from others throughout the search. At each step of the search, it must be known whether each vertex is "reached" or not. Furthermore, the vertices in $\mathrm{Adj}(p)$ must be identified at each stage when the search applying Rule 1 finds a new vertex $p \in P$. In summary, three-bit storage is needed for each vertex in order to classify vertices.

As temporary storages, two one-dimensional arrays $VP(\cdot)$ and $VX(\cdot)$ consisting of N pointers are used to find "unreached" vertices in P and those in $(V - P)$, respectively, by scanning each adjacent vertex list from the top to the

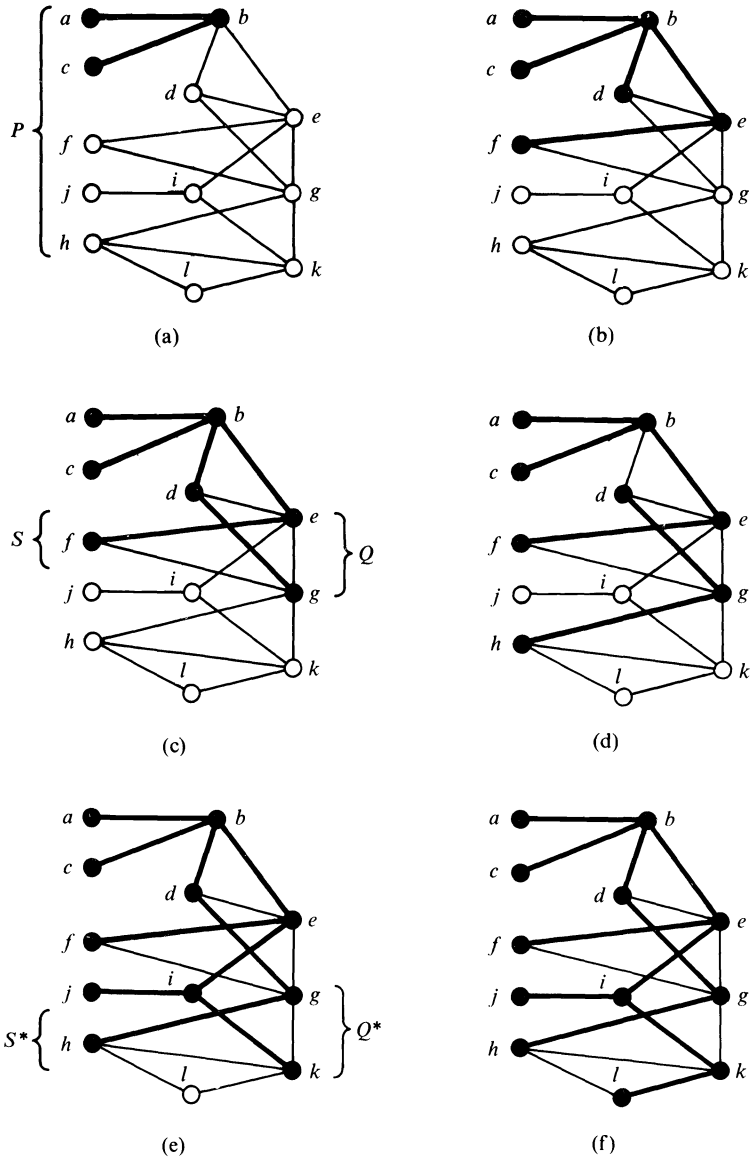


FIG. 2. *Search for obtaining a desired set Q*

bottom. For example, let $m \in \{1, 2, \cdots, M\}$ represents a vertex. Then the vertices adjacent to m are stored in $E(\cdot)$ from the $(V(m))$th position to the $(V(m+1)-1)$st position. $VP(m) = n$, $V(m) \leqq n < V(m+1)$, means that the search is at the stage where the vertices $E(V(m)), E(V(m)+1), \cdots, E(n)$ are known to be either in $V - P$ or to have already been "reached". $VX(\cdot)$ is used in a similar way. This facility enables us to avoid duplicate search of vertices.

We prepare two more temporary storages. A one-dimensional array of length M is used to accumulate vertices which constitute a candidate for the desired set $Q \subset V - P$. A stack of depth M is used to update the set of "reached" vertices whose adjacent vertices are necessarily tested. To be more exact, a vertex t is deleted from the stack once all the vertices in Adj (t) are known to have been "reached".

The data structure consists of several one-dimensional arrays, a stack and three bit-strings. Among them, $E(\cdot)$ is of length $2N$, while all other one-dimensional arrays including the stack and the bit-strings are of length M. In summary, the storage requirement is bounded by $k_1M + k_2N$ for some constants k_1 and k_2.

Now the following algorithm written in ALGOL-like notation obtains a desired set $Q \subset V - P$ for a given subset $P \subset V$ of a graph $G = (V, E)$.

```
    begin
    empty stack; comment initialization;
S0: pick a vertex v0 ∈ P and another vertex v1 ∈ V − P such that (v0, v1) ∈ E;
    mark v0 and v1 "reached";
    Q := {v1}; put v1 on stack;
    while stack is not empty do
      begin comment search subject to Rule 1;
      S1: delete the top vertex x from stack;
      S2: if there is an "unreached" vertex p ∈ P
            such that (p, x) ∈ E then
            begin comment Rule 1 is expired;
        S3: mark p "reached";
        S4: mark all the vertices in Adj (p) "p-adjacent";
        S5: Q := {x};
            while stack is not empty do
              begin comment search subject to Rule 2;
              S9: delete the top vertex y from stack;
                  if y is "p-adjacent" then
              S10: Q := Q ∪ {y};
                  else begin
              S11: while there is an "unreached" vertex r ∈ P such that (r, y) ∈ E
                     do mark r "reached";
              S12: while there is an "unreached" vertex v ∈ V − P such that
                     (v, y) ∈ E do mark v "reached" and put it on stack;
                   end;
              end comment Rule 2 is expired;
        S6: remove "p-adjacent" marks of the vertices in Adj (p);
        S7: put all the vertices in Q on stack;
            end;
      S8: else while there is an "unreached" vertex u ∈ V − P such that (u, x) ∈ E
            do mark u "reached" and put it on stack;
      end
    end;
```

To estimate running time, we shall examine the data structure and the flow of the algorithm. Statement S0 may require examination of all edges to find v_0

and v_1; time bound is $O(N)$. In statements S2 and S11, the vertices in Adj (x) and Adj (y), respectively, stored in $E(\cdot)$ are checked one by one. Since duplicated check of the same entry in $E(\cdot)$ can be avoided by the use of $VP(\cdot)$, the total number of required checks is bounded by $2N$. The time required for statements S8 and S12 is estimated in the same way. In statements S3, S8, S11 and S12, the number of operations of marking vertices "reached" is bounded by M ($\leqq N + 1$), i.e., by the number of vertices. In statement S4 or S6, the number of operations of attaching or removing "p-adjacent" marks of vertices is bounded by the total sum of the degrees of the vertices in P, which is less than $2N$. The total number of vertices put on Q is not greater than the number of the edges which connect between the vertices in P and those in $V - P$. Thus time bound for statements S5, S7 and S10 is $O(N)$. The running time for statements S1 and S9 is determined by the total number of entries put on the stack. Going through statement S8 or S12, at most M ($\leqq N + 1$) entries are put on the stack. Going through statement S7, at most N entries are put on the stack. In summary, a desired set Q can be obtained in $O(N)$ time, i.e., an optimal ordering can be obtained in $O(M \cdot N)$ time.

The algorithm presented here is much more efficient than the minimal triangulation (optimal ordering) algorithm in the previous paper [4]. The former requires only $O(M \cdot N)$ time, while the latter requires $O(M^2 \cdot N)$ time. In the previous algorithm, we chose a vertex, which satisfies the condition of Theorem C, at each stage of the vertex elimination. If the test of this condition is properly implemented, the running time is proportional to N. In the worst case, however, $M(M + 1)/2$ tests may be required to find an optimal ordering. Thus the total running time of the algorithm [4] is estimated by $O(M^2 \cdot N)$.

Remarks. 1. The running time $O(M \cdot N)$ of the proposed algorithm is no worse than the minimum degree algorithm [6], [7], which is widely used for the reason that relatively simple operations are involved to determine a good ordering. The minimum degree algorithm requires $O(M^3)$ time as far as a conservative upper bound is concerned.

2. Recently, another fast algorithm for obtaining an optimal ordering was presented [8]. This algorithm requires $O(M \cdot \bar{N})$ running time, rather than $O(M \cdot N)$, where $\bar{N}$ is the number of edges of the resultant triangulated graph. But it can obtain the corresponding minimal triangulation simultaneously.

3. More recent work [9] presents another $O(M \cdot N)$ algorithm to find an optimal ordering, together with a $O(M, \bar{N})$ algorithm to calculate the triangulation for a given ordering.

REFERENCES

[1] S. PARTER, *The use of linear graphs in Gauss elimination*, SIAM Rev., 3 (1961), pp. 119–130.

[2] C. BERGE, *Some classes of perfect graphs*, Graph Theory and Theoretical Physics, F. Harary, ed., Academic Press, New York, 1967, pp. 155–166.

[3] D. J. ROSE, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.

[4] T. OHTSUKI, L. K. CHEUNG AND T. FUJISAWA, *Minimal triangulation of a graph and optimal pivoting order in a sparse matrix*, Ibid., to appear.

[5] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[6] D. J. ROSE, *A graph-theoretic study of numerical solutions of sparse positive definite system of linear equations*, Graph Theory and Computing, R. Read, ed., Academic Press, New York, 1972, pp. 183–217.

[7] E. C. OGBUOBIRI, W. F. TINNEY AND J. W. WALKER, *Sparsity-directed decomposition for Gaussian elimination on matrices*, IEEE Trans. Power Apparatus and Systems, 89 (1970), pp. 141–150.

[8] T. FUJISAWA AND H. ORINO, *An efficient algorithm of finding a minimal triangulation of a graph*, Proc. 1974 IEEE Internat. Symp. on Circuits and Systems, San Francisco, 1974.

[9] D. J. ROSE AND R. TARJAN, *Algorithmic aspects of vertex elimination on graphs*, Memo. ERL-M483, Electronics Res. Lab., Univ. of Calif. at Berkeley, 1974.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

ON THE NUMBER OF ADDITIONS TO COMPUTE SPECIFIC POLYNOMIALS*

ALLAN BORODIN† AND STEPHEN COOK‡

Abstract. The number of addition-subtraction operations required to compute univariate polynomials is investigated. The existence of rational coefficient polynomials of degree n requiring $\sim(\sqrt{n})$ $\pm$ operations is established using an argument based on algebraic independence. A more analytic argument is used to relate $\pm$ complexity to the number of distinct real zeros possessed by a given real coefficient polynomial.

Key words. arithmetic complexity, algebraic complexity, additions, polynomial evaluation, algebraic independence, real zeros

1. Introduction. It is well known from the work of Motzkin [5], Belaga [2] and Pan [6], that "most" nth-degree polynomials $p \in R[x]$ require about $n/2$ $\times$, $\div$ operations and n $\pm$ operations and that these bounds can always be achieved within the framework of preconditioned evaluation.[1] More precisely, if p can be computed using less than $\lceil (n+1)/2 \rceil$ $\times$, $\div$ or less than n $\pm$ operations, then the coefficients of p are algebraically dependent.

However, it can be argued that only polynomials in $Q[x]$ are of any computational concern. Moreover, one would like "practical tests" to determine the complexity of a specific polynomial. With respect to nonscalar $*$ operations, Paterson and Stockmeyer [7] are able to show that approximately $\sqrt{n}$ such operations are required for "most" nth-degree polynomials in $Q[x]$. Also they show that every nth-degree polynomial can be computed in about $\sqrt{2n}$ nonscalar $*$ operations. In $Q[x]$, the scalar $*$ operations can be simulated by (an unbounded number of) $\pm$ operations. Strassen [9] uses a careful analysis of the Motzkin–Belaga argument (and also of the corresponding development in Paterson–Stockmeyer) to exhibit specific polynomials in $Z[x]$ whose required complexity is nearly that obtainable by general preconditioning methods. For example, any program for

$$p(x) = \sum_{i=0}^{n} 2^{2^{in^3}} x^i$$

* Received by the editors September 15, 1974, and in revised form April 28, 1975.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada. Now at Computer Science Department, Cornell University, Ithaca, New York 14853.

‡ Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A1.

[1] See Knuth [4] for a review. We use the following notation: R, Q, C for the field of reals, rationals, complex numbers, respectively, Z and N denote the integers and nonnegative integers, respectively; $F[y_1, \cdots, y_m]$ is the ring of polynomials, $F[[y_1, \cdots, y_m]]$ is the power series ring and $F(y_1, \cdots, y_m)$ the field of rational functions in $y_1, \cdots, y_m$ over F. We will also use $*$ operations to denote either a $\times$ or $\div$ operation.

requires

(i) either $n/2 - 4$ $*$ operations and $n - 4$ $\pm$ operations or at least $n^2/\log n$ total operations,
(ii) at least $\sqrt{n}$ nonscalar $*$ operations.

That is, if one chooses to trade off $\pm$ operations to reduce the $*$ complexity of $p(x)$, then it can be done but only with an exorbitant cost of at least $n^2/\log n$ $\pm$ operations.

The situation when counting $\pm$ operations with the potential of unlimited $*$ operations is not as clear. In fact, we are not aware of any previous results which show that not all $p \in Q[x]$ are computable in (say) 4 $\pm$ operations. A "usable" characterization of precisely which polynomials are computable in 4 $\pm$ operations is more than a tedious exercise. Does an analogue of Paterson–Stockmeyer hold? That is, can the output of a program (which is computing an nth-degree polynomial) using k $\pm$ operations *but an unbounded number of* $*$ *operations* be represented by $\sum_{i=0}^{n} q_i(\alpha_1, \cdots, \alpha_t)x^i$ for some fixed rational functions $\{q_i\}$, where the number t of parameters $\{\alpha_i\}$ is bounded by some function of k? We shall show in § 3 that this is the case with $t \sim k^2$. But unlike the situation in Paterson–Stockmeyer, we do not yet know if the use of unlimited $*$ operations can in general reduce the $\pm$ complexity of polynomials in $Q[x]$.

While the arguments based on algebraic dependence provide us with our best lower bounds thus far, a different approach of independent interest is taken in § 4. Namely, we are able to show that the number of $\pm$ operations required to compute any $p \in R[x]$ is bounded below by a function of the number of distinct real zeros of p. The potential (e.g., for producing nonlinear lower bounds) and limitations of this approach will be discussed.

2. The model and a review of results based on algebraic independence. We follow informally the notation of Winograd [11] and say that we are interested in computing $p \in F[x]$ over $G(x)$ given $G \cup \{x\}$, where $F \subseteq G$ and G is a field. That is, we think of a program P as a sequence of statements $\langle s_1, \cdots, s_m \rangle$; each s_i is of the form "p_i operation q_i", where operation $\in \{+, -, \times, \div\}$ and each operand p_i, q_i is either

(i) in $G \cup \{x\}$; i.e., a scalar constant or "x", or
(ii) a previously computed s_j $(j < i)$.

P computes $p \in F[x]$ if $p = s_m$ (as elements of $F[x] \subseteq G(x)$). When $\div$ is not allowed, we say that we are computing $p \in F[x]$ over $G[x]$ given $G \cup \{x\}$.

In this section and in § 3, the choice of F and G is not that essential, but for definiteness we can take $F = Q$ and $G = C$. Section 4 will depend essentially on the choice $G = R$.

DEFINITION 1. Let H be an extension field of $F = Q$. $u_1, \cdots, u_m \in H$ are *algebraically dependent* (over Q) if there exists a nontrivial $f \in Z[y_1, \cdots, y_t]$ such that $f(u_1, \cdots, u_t) = 0$.

LEMMA 1 (Van der Waerden [10]). *Let* $p_1, \cdots, p_m \in Q(\alpha_1, \cdots, \alpha_t)$. *If* $m > t$, *then* $p_1, \cdots, p_m$ *are algebraically dependent.*

For the sake of completeness and motivation, let's briefly sketch the lower bound of Paterson and Stockmeyer. Assuming no $\div$, we can construct a "canonical"

program using k nonscalar $\times$ operations; namely:

$$s_{-1} \leftarrow 1$$
$$s_0 \leftarrow x$$
$$s_i \leftarrow \left(\sum_{j<i} \alpha'_{j,i} s_j\right) \times \left(\sum_{j<i} \alpha''_{j,i} s_j\right)$$
$$s_{k+1} \leftarrow \sum_{j \leqq k} \alpha_{j,k+1} s_j.$$

Then $s_{k+1} = \sum_{j=0}^{r} p_j(\boldsymbol{\alpha}) x^j$, where $r \leqq 2^k$, and $\boldsymbol{\alpha} = \langle \alpha'_{-1,1}, \alpha'_{0,1}, \alpha''_{-1,1}, \cdots, \alpha_{k,k+1} \rangle = \langle \alpha_1, \cdots, \alpha_t \rangle$, where t is approximately k^2. With a little care, it can be shown that $t \leqq k^2 + 1$.

THEOREM 1 (Paterson and Stockmeyer). *If $n + 1 > k^2 + 1$, then there exists an n-th-degree polynomial in $Q[x]$ which is not computable over $C[x]$ in k nonscalar multiplications.*

Proof. Assume $p(x) = \sum_{j=0}^{n} a_j x^j$ is computable in k nonscalar multiplications. Then $p(x) = \sum_{j=0}^{n} p_j(\alpha_1, \cdots, \alpha_t) x^j$ for some choice of $\alpha_1, \cdots, \alpha_t$ with $t = k^2 + 1$.

But $p_0(\boldsymbol{\alpha}), \cdots, p_n(\boldsymbol{\alpha})$ are algebraically dependent if $n + 1 > t$, and hence there exists a nontrivial $f \in Z[y_1, \cdots, y_{n+1}]$ such that $f(p_0(\boldsymbol{\alpha}), \cdots, p_n(\boldsymbol{\alpha})) = 0$. If every $p \in Q[x]$ were computable in k nonscalar multiplications, then $f(q_0, \cdots, q_n) = 0$ for all $\langle q_0, \cdots, q_n \rangle$ in Q^{n+1}. Hence $f \equiv 0$ because Q^{n+1} is dense in R^{n+1} and f is continuous. This contradicts the assumption that f be nontrivial.

As Paterson and Stockmeyer observe (in extending the result to allow $\div$), if we can produce a finite number, say l, of canonical programs for some measure (rather than just one), then the same type of results will follow; for the "algebraic dependence of each program" is characterized by some $f_i \in Z[y_1, \cdots, y_{n+1}]$, and hence the coefficients of any nth-degree polynomial computable in k operations will be zero of $f = \prod_{i=1}^{l} f_i$.

From these observations, the following fact follows directly.

FACT 1. *Let $\psi : N \to N$ be any function.*

(a) *There are n-th-degree polynomials in $Q[x]$ which either require $\lceil (n+1)/2 \rceil$ $*$ operations or more than $\psi(n)$ $\pm$ operations.*

(b) *There are n-th-degree polynomials in $Q[x]$ which either require n $\pm$ operations or more than $\psi(n)$ $*$ operations.*

In either case, once $\psi(n)$ is given, there are only a finite number of canonical programs each having the appropriate number of parameters.

3. A lower bound for $\pm$ operations based on algebraic dependence. We shall now consider the situation when the number of $*$ operations is *not* bounded by any function of the degree. One might argue that this is a totally impractical hypothesis, but we believe that the questions arising out of the developments in §§ 3 and 4 are more than academic. The difficulty in trying to bound $\pm$ operations is suggested by the simplest example. Let $s \leftarrow (x + \alpha)^u$ represent the first $\pm$ step (say $u \in N$). If we treat u as a parameter, then $s = \alpha^u + u\alpha^{u-1}x + \binom{u}{2}\alpha^{u-2}x^2 + \cdots$.

We cannot immediately view s as $\sum_{j=0} p_j(\alpha) x^j$ with the p_j being polynomials. Nor can we treat each $\alpha, \alpha^2, \alpha^3, \cdots$ as a parameter, for then the number of parameters

will not be a bounded function of the number of $\pm$ operations. We might want to argue that u cannot be too large without introducing some inefficiency; but this is just the sort of question we cannot yet answer.

Let's first define a "canonical program" having k $\pm$ operations.

LEMMA 2 (Belaga). *Let f in $G(x)$ be computed by a program over $G(x)$ given $G \cup \{x\}$ using k $\pm$ operations. Then f is computed by the following "program" for some appropriate choice of $\gamma_1, \cdots, \gamma_{k+1}$ in G and $\{m_{j,i}\} \subseteq Z$:*

$$
\begin{aligned}
&T_0 \leftarrow 1\\
&S_0 \leftarrow x\\
&\vdots\\
&\left.\begin{aligned} T_i &\leftarrow \prod_{j<i} S_j^{m_{j,i}}\\ S_i &\leftarrow \gamma_i + T_i \end{aligned}\right\} 1 \leqq i \leqq k\\
&\vdots\\
&T_{k+1} \leftarrow \gamma_{k+1} \prod_{j \leqq k} S_j^{m_{j,k+1}}.
\end{aligned}
$$

A proof can be found in Borodin and Munro [1, § 3.2].

Allowing negative exponents accounts for $\div$ and also allows a simplification in the number of parameters introduced. On the other hand, we will have to view the computation as taking place over some power series ring $G[[x - \theta]]$ as in Strassen [9] in order to accommodate the negative exponents.

We want to express T_{k+1} as a polynomial in x whose coefficients are in some $H = Z(\alpha_1, \cdots, \alpha_t)$. Let's concern ourselves only with the computation of nth-degree polynomials. Suppose P computes p over $G[[x]]$. Then P correctly computes p over $G[[x]]$ (mod x^{n+1}), i.e., with all higher order terms dropped throughout the computation.

The example $s \leftarrow (x + \alpha)^u$ illustrates the approach to be taken. We can consider $n + 2$ cases: $u = 0, \cdots, u = n, u > n$. It is clear that for each $u = i$ ($i \leqq n$) we can represent s (mod x^{n+1}) as some $\sum_{j=0}^{n} p_j(\alpha)x^i$. For $u > n$, we have

$$
s = \alpha^u + u\alpha^{u-1}x + \cdots \binom{u}{n}\alpha^{u-n}x^n = \sum_{j=0}^{n} r_j(\alpha, \beta, u)x^j,
$$

where $\beta = \alpha^u$ and $r_j \in Z(\alpha, \beta, u)$; i.e.,

$$
r_0(\alpha, \beta, u) = \beta, \quad r_1(\alpha, \beta, u) = u\frac{\beta}{\alpha}, \quad \cdots, \quad r_n = \binom{u}{n}\frac{\beta}{\alpha^n}.
$$

For this simple example, the cases $u \leqq n$ could be subsumed by the case $u > n$, but in the proof of Theorem 2 this need not always be true.

If $u \in Z$ (rather than N) we would have $2n + 3$ cases: $u < -n, u = -n, \cdots, u = 0, \cdots, u = n, u > n$. Consider $u < 0$ and assume $\alpha \neq 0$. (If $\alpha = 0$, we would have to consider power series in $x - \theta$ rather than x, for some appropriate θ.)

Then

$$s \leftarrow 1/(\alpha + x)^{-u} = [1/(\alpha + x)]^{-u}$$
$$= \left[\frac{1}{\alpha} - \frac{1}{\alpha^2}x + \frac{1}{\alpha^3}x^2 - \frac{1}{\alpha^4}x^3 + \cdots\right]^{-u}.$$

Again, the cases $u = -i$ $(i \leqq n)$ could be handled separately. For $u < -n$, we have

$$s \,(\mathrm{mod}\, x^{n+1}) = \left[\frac{1}{\alpha} - \frac{1}{\alpha^2}x + \cdots (-1)^n \frac{1}{\alpha^{n+1}}x^n\right]^{-u}$$
$$= \sum_{j=0}^{n} r_j(\alpha, \beta, u)x^j \quad \text{with } \beta = \alpha^u.$$

THEOREM 2. *Consider computations over $C(x)$ given $C \cup \{x\}$, and let $n \geqq (k+2)^2$. Then there exists an n-th-degree polynomial $p(x)$ in $Z[x]$ which cannot be computed in k $\pm$ operations.*

The proof shows that in fact, "most" nth-degree polynomials in $Q[x]$ cannot be computed in k $\pm$ operations.

Proof. Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be an arbitrary nth-degree polynomial, and let P be a k $\pm$ step program which computes $p(x)$.

(A) Convert P to a program P' over $C(\theta, \tilde{x})$ given $C \cup \{\theta\} \cup \{\tilde{x}\}$ which computes $\tilde{p}(\tilde{x}) \triangleq p(x)$, where θ is a new indeterminate and $\tilde{x} = x - \theta$. Specifically, let

$$P' = \begin{cases} x \leftarrow \tilde{x} + \theta \\ P \end{cases}$$

and then

$$\tilde{p}(\tilde{x}) = \sum_{i=0}^{n} \tilde{a}_i \tilde{x}^i \quad \text{with} \quad \tilde{a}_j = \sum_{k=j}^{n} \binom{k}{k-j} a_k \theta^{k-j}.$$

Using Lemma 2, we have

$$P' = \begin{cases} S_0 = x \leftarrow \tilde{x} + \theta \\ T_1 \leftarrow S_0^{m_{0,1}} \\ S_1 \leftarrow \gamma_1 + T_1 \\ \vdots \\ T_{k+1} \leftarrow \gamma_{k+1} \prod_{j \leqq k} S_j^{m_{j,k+1}} \end{cases}$$

(B) P' introduces $v = (k+1)(k+2)/2$ exponents $\{m_{j,i}\}$, all of which can be treated as parameters. For every exponent, there are $2n + 3$ choices to be considered (namely, $m_{j,i} < -n, m_{j,i} = -n, \cdots, m_{j,i} = n, m_{j,i} > n$), or $(2n+3)^v$ *cases* in total. Each case will determine a canonical program. For each of these (*finitely many*) programs, we are able to algebraically characterize the program statements (subresults) and the target polynomial.

(C) *Claim.* Having chosen any one of the $(2n + 3)^v$ cases, there are rational functions $\{p_j^i\}$ over $Q(\theta)$ such that for $0 \leqq i \leqq k$,

$$S_i = \sum_{j=0}^{n} p_j^i(\alpha_1, \cdots, \alpha_{t(i)})\tilde{x}^j \qquad (\text{mod } \tilde{x}^{n+1})$$

for an appropriate choice of parameters $\{\alpha_l\} \subseteq C$ with $t(i) < (i + 1)^2$. Consequently, $\tilde{p}(x) = \sum_{j=0}^{n} p_j^{k+1}(\alpha_1, \cdots, \alpha_{t(k+1)})\tilde{x}^j$ for some appropriate choice of the parameters.

For each of the $(2n + 3)^v$ cases, we can prove the claim by induction on k. For the basis of the induction, $S_0 \leftarrow \tilde{x} + \theta$ and thus $t(0) = 0$. Now assume

$$S_i = \sum_{j=0}^{n} p_j^i(\alpha_1, \cdots, \alpha_{t(i)})\tilde{x}^j \qquad (\text{mod } \tilde{x}^{n+1}),$$

with $t(i) < (i + 1)^2$ for $0 \leqq i \leqq r$. We want to show that

$$S_{r+1} = \sum_{j=0}^{n} p_j^{r+1}(\alpha_1, \cdots, \alpha_{t(r+1)})\tilde{x}^j \qquad (\text{mod } \tilde{x}^{n+1})$$

and that $t(r + 1) \leqq t(r) + 2(r + 1) < (r + 2)^2$.

We have

$$T_{r+1} \leftarrow \prod_{j \leqq r} S_j^{m_{j,r+1}}$$

$$S_{r+1} \leftarrow \gamma_{r+1} + T_{r+1}.$$

Introduce new parameters (and rename by $\alpha_{t(r)+1}, \cdots, \alpha_{t(r+1)}$) to represent

$$\gamma_{r+1}, m_{0,r+1}, \cdots, m_{r,r+1}, [p_0^1(\alpha_1, \cdots, \alpha_{t(1)})]^{m_{1,r+1}}, \cdots, [p_0^r(\alpha_1, \cdots, \alpha_{t(r)})]^{m_{r,r+1}}.$$

We have thus introduced $2(r + 1)$ new parameters. Now it remains to show that $S_{r+1} = \sum p_j^{r+1}(\alpha_1, \cdots, \alpha_{t(r+1)})\tilde{x}^j (\text{mod } \tilde{x}^{n+1})$. $S_{r+1} = \prod_{j=0}^{r} S_j^{m_{j,r+1}} + \gamma_{r+1}$.

Look at any

$$S_i^{m_{i,r+1}} = \left[\sum_{j=0}^{n} p_j^i(\alpha_1, \cdots, \alpha_{t(i)})\tilde{x}^j\right]^{m_{i,r+1}} \qquad (\text{mod } \tilde{x}^{n+1}).$$

We must consider the $2n + 3$ cases $m_{i,r+1} < -n$, $m_{i,r+1} = -n, \cdots, m_{i,r+1} > n$. Let us just outline the case $m_{i,r+1} > n$.

The coefficient of $\tilde{x}^l$ $(l \leqq n < m_{i,r+1})$ is

$$\sum_{m_1 \cdot 1 + m_2 \cdot 2 + \cdots + m_n \cdot n = l} \binom{m_{i,r+1}}{m_1} \binom{m_{i,r+1} - m_1}{m_2} \cdots \binom{m_{i,r+1} - m_1 - \cdots - m_{n-1}}{m_n}$$
$$\cdot [p_0^i(\alpha_1, \cdots)]^{m_{i,r+1} - m_1 - \cdots - m_n} p_1^i(\cdot)^{m_1} \cdots p_n^i(\cdot)^{m_n}.$$

And as in the simple example $(x + \alpha)^u$, the expression can be written as a rational function $g(\alpha_1, \cdots, \alpha_{t(r)}, m_{i,r+1}, [p_0^i]^{m_{i,r+1}})$. So it follows that $\prod S_j^{m_{j,r+1}} + \gamma_{r+1}$ $(\text{mod } \tilde{x}^{n+1})$ can be represented as desired.

(D) So for each of the $(2n+3)^v$ cases, we are computing $\sum_{j=0}^{n} p_j^{k+1}(\alpha_1, \cdots, \alpha_{t(k+1)})\tilde{x}^j$ with rational functions p_j^{k+1} over $Q(\theta)$; alternatively, we have

$$\sum_{j=0}^{n} q_j^{k+1}(\theta, \alpha_1, \cdots, \alpha_{t(k+1)})\tilde{x}^j$$

with q_j^{k+1} over Q. Now recall the assumption that $n \geqq (k+2)^2 \geqq t(k+1)+1$. Using Lemma 1, we know that for each case c there is a nontrivial polynomial f_c over Z such that $f_c(q_0^{k+1}, \cdots, q_n^{k+1}) = 0$. Let $f \triangleq \prod_{c=1}^{(2n+3)^v} f_c$. By exhausting all the cases, we have shown that if $\tilde{p}(\tilde{x}) = \sum_{j=0}^{n} \tilde{a}_j \tilde{x}^j$, then $f(\tilde{a}_0, \cdots, \tilde{a}_n) = 0$.

Recalling that $\tilde{a}_j = \sum_{k=j}^{n} \binom{k}{j} a_k \theta^{k-j}$, we can view $f(\tilde{a}_0, \cdots, \tilde{a}_n)$ as a polynomial in θ which must be identically zero. In particular, the constant term of $f(\tilde{a}_0, \cdots, \tilde{a}_n)$, which is $f(a_0, \cdots, a_n)$, must be zero. □

(E) Summarizing, we have shown that there is one fixed nontrivial polynomial f over Z such that if $p(x) = \sum_{i=0}^{n} a_i x^i$ is computable in k $\pm$ operations and $n \geqq (k+2)^2$, then $f(a_0, \cdots, a_n) = 0$. Now we can complete the argument just as in Paterson and Stockmeyer [7]. The nontrivial polynomial f cannot be zero on any dense subset of Q^{n+1}, so that the $\pm$ complexity of "most" nth-degree polynomials in $Q[x]$ is at least $\sqrt{n} - 2$ (i.e., the exceptions $\langle a_0, \cdots, a_n \rangle$ are nowhere dense in Q^{n+1}). Since any polynomial $p(x)$ in $Q[x]$ can be viewed as $q \cdot z(x)$ with q in Q and $z(x)$ in $Z[x]$, it follows that there are polynomials in $Z(x)$ which are difficult to compute. □

COROLLARY 1. *There exist n-th-degree polynomials in $Q[x]$ which require $\sqrt{n} - 2$ $\pm$ operations even if we do the computation* mod x^{n+1}, *i.e., chop off high order terms without cost.*

We remark that by calculating upper bounds on the degree and weight of the polynomials $\{p_j^i(\alpha_1, \cdots)\}$, we can exhibit, as in Strassen [9], specific polynomials (with integer coefficients) which require $\sim\sqrt{n}$ $\pm$ operations.

Whether or not a $\sqrt{n}$ upper bound on $\pm$ operations can generally be obtained remains an open question. We suspect that while it may be possible to achieve a saving (in $\pm$ operations by the unlimited use of $*$ operations) when computing mod x^{n+1}, the additional requirements imposed by the cancellation of high order terms will preclude any such saving. That is, $\pm$ operations in computations over $Q(x)$ cannot in general be reduced by $*$ operations.

We state the following conjecture: there is a function $\gamma(k, n)$ satisfying the following property—if p is an nth-degree polynomial (say in $Q[x]$) and p is computable in k $\pm$ operations, then p is computable in k $\pm$ operations and no more than $\gamma(k, n)$ $*$ operations.

Finally, we can note that if a general saving in $\pm$ operations can be achieved for any fixed degree n_0 (say $\beta(n_0)$ $\pm$ operations), then a proportionate saving can be achieved for all $n \geqq n_0$ (i.e., only need about $\beta(n_0) \cdot n/n_0$ $\pm$ operations). More precisely, if $p(x)$ has degree $n = d(n_0+1) - 1$, it can be written as

$$(c_{0,1} + \cdots + c_{n_0,1}x^{n_0} + (c_{0,2} + \cdots + c_{n_0,2}x^{n_0})x^{n_0+1} + \cdots + (c_{0,d} + \cdots + c_{n_0,d}x^{n_0})x^{(d-1)(n_0+1)},$$

which could then be evaluated in $[d\beta(n_0) + (d-1)]$ $\pm$ operations.

4. A lower bound based on the number of real zeros. In Strassen [8], we see the first significant results concerning nonlinear lower bounds for arithmetic complexity. Algebraic geometry provides the proper notion of "degree" for a set of (rather than just one) polynomials in several variables. The geometric formulation of degree is "correct" from a complexity point of view, since Strassen is able to show that the degree can at most double after a $*$ operation and is unchanged after any $\pm$ operation. In this way, one can prove for example that any nth-degree polynomial evaluated at n arbitrary points requires $n \log n$ $*$ operations.

For $\pm$ operations, we do not yet have an appropriate concept or property (such as degree) which can be used to derive nonlinear lower bounds. For example: is polynomial multiplication nonlinear with respect to $\pm$ operations? Does there exist an nth-degree polynomial which requires $n \log n$ $\pm$ operations for computation at n arbitrary points? One type of property that may be relevant is to look at the zeros associated with the polynomials computed during a computation. Such a property could also provide a constructive means for analyzing the complexity of specific polynomials. If we look at all complex zeros, then we can obviously generate an nth-degree $p \in R[x]$ which has n distinct zeros in one $\pm$ operation (of course, these zeros have a nice structure).

The approach of this section is to show that the number of distinct *real* zeros in any polynomial p in $R[x]$ is bounded by a function of the number of $\pm$ operations required to compute p. Unfortunately (unlike degree with respect to $*$ operations), it is not true that if p_1 and p_2 have $\leqq r$ distinct real zeros, then $p_1 + p_2$ has $\leqq \varphi(r)$ distinct real zeros (for some function $\varphi : N \to N$).

We consider again the canonical program given in Lemma 2 with $G = R$. Throughout this section, S_i and T_i are defined as in Lemma 2, and are to be viewed as rational functions in $R(x)$. We want to bound the number of distinct real roots in T_n as a function of n. To do so, a more general induction hypothesis seems necessary.

Notation. If f is in $R(x)$, then f' denotes df/dx.

DEFINITION 2. Let f be in $R(x)$.

$$\#f = \text{the number of distinct real zeros} + \text{the number of distinct real poles in } f.$$

If f is in $R[x]$, then $\#f$ is just the number of distinct real zeros.

The main theorem of this section is motivated by the following lemma.

LEMMA 3. *If f in $R[x]$ has k nonzero terms, then $\#f \leqq 2k - 1$. Also, there exists an f in $R[x]$ having k nonzero terms and $\#f = 2k - 1$.*

Proof. The proof proceeds by induction on k.

$k = 1$. $f(x) = ax^r$ and $x = 0$ is the only possible zero.

Induction step. Let $f(x) = x^r \cdot g(x) = x^r(a_{i_0} + \cdots + a_{i_k}x^{i_k})$. $\#f \leqq 1 + \#g \leqq 1 + \#g' + 1$ (By Rolle's theorem). So $\#f \leqq 2 + (2k - 1) = 2(k + 1) - 1$. The polynomial $f(x) = \prod_{i=1}^{k-1} x(x^2 - i)$ achieves the bound. □

Since we will be dealing with rational functions, we need a generalization of Rolle's theorem.

LEMMA 4. *Let $f = p/q$ be in $R(x)$. Then $\#f \leqq 2\#f' + 1$.*

Proof. The proof is in two cases.

Case 1. $\#q \geqq \#p$. Since every pole of f is a pole of f', it follows that $\#f \leqq 2\#q \leqq 2\#f'$.

Case 2. $\#q < \#p$. We claim that the number of zeros of f' is at least $\#p - (\#q + 1)$, since f is a continuous function except at its poles. Hence $\#f' \geqq \#p - (\#q + 1) + \#q = \#p - 1$. So $2\#f' \geqq 2\#p - 2 > \#f - 2$. □

We need one more lemma before proceeding to the main theorem.

LEMMA 5. *Recall the meaning and notation for S_i from Lemma 2. Then*

$$\frac{S_n'}{S_n} = \frac{p_n(S_0, \cdots, S_n)}{S_0 S_1 \cdots S_n},$$

where $p_n \in R[y_0, \cdots, y_n]$, and $\deg(p_n) \leqq n$.

Proof. The proof is by induction on n. When $n = 0$, take $p_0 = 1$. When $n > 0$, we have

$$S_n' = T_n' = T_n\left[\sum_{i=0}^{n-1} m_{i,n}\frac{S_i'}{S_i}\right].$$

Hence one can check that

$$p_n(S_0, \cdots, S_n) = (S_n - \gamma_n)\sum_{i=0}^{n-1} m_{i,n} S_{i+1} \cdots S_{n-1} p_i(S_0, \cdots, S_i),$$

by dividing both sides by $S_0 \cdots S_n$ and applying the induction hypothesis. □

THEOREM 3. *Let*

$$f = \sum_{i=1}^{N} S_0^{r_{0,i}} \cdots S_n^{r_{n,i}} Q_i(S_0, \cdots, S_n),$$

with each Q_i in $R[y_0, \cdots, y_n]$ having $\deg \leqq M$, *and $r_{j,i} \in Z$. Then $\#f \leqq \varphi(n, N, M)$ for some function $\varphi : N^3 \to N$.*

Note. The function φ, which will be defined by the induction proof, is independent of the exponents $r_{j,i}$ in Z. Our interest lies in the bound $\varphi(n, 1, 0)$.

Proof. The proof is by induction on n, with the induction step being established by induction on N.

$n = 0$.

$$f = \sum_{i=1}^{N} S_0^{r_{0,i}} Q_i(S_0) = \sum_{i=1}^{N} x^{r_{0,i}} Q_i(x) = x^t \sum_{i=1}^{N} x^{t_i} Q_i(x),$$

where t is chosen so that each $t_i \geqq 0$. Since $\deg(Q_i) \leqq M$, the total number of terms in the polynomial represented by the sum is at most $N(M + 1)$. Hence, by Lemma 3, we have

$$\#f \leqq 1 + 2N(M + 1) - 1 = 2N(M + 1).$$

Thus $\varphi(0, N, M) = 2N(M + 1)$.

$n > 0$. $N = 1$.

$$f = \prod_{j=0}^{n} S_j^{r_{j,1}} Q_1(S_0, \cdots, S_n).$$

$$\#f \leqq \#\left(\prod_{j=0}^{n} S_j^{r_{j,1}}\right) + \#Q_1(S_0, \cdots, S_n).$$

(a)

$$\# \prod_{j=0}^{n} S_j^{r_{j,1}} \leqq \# \prod_{j=0}^{n} S_j$$

$$= \# \prod_{j=0}^{n-1} S_j \left(\prod_{k=0}^{n-1} S_k^{m_{k,n}} + \gamma_n \right)$$

$$= \# \left(\prod_{j=0}^{n-1} S_j^{m_{j,n}+1} + \prod_{j=0}^{n-1} S_j \gamma_n \right).$$

So

$$\# \textstyle\prod_{j=0}^{n} S_j^{r_{j,1}} \leqq \varphi(n-1, 2, 0).$$

(b) Let $Q_1 = \sum_{k=0}^{M} q_k(S_0, \cdots, S_{n-1}) S_n^k$ with $\deg(q_k) \leqq M - k$, and let $y = S_n - \gamma_n = \prod_{j=0}^{n-1} S_j^{m_{j,n}}$. Then

$$Q_1 = \sum_{k=0}^{M} q_k(S_0, \cdots, S_{n-1})(y + \gamma_n)^k$$

$$= \sum_{k=0}^{M} \tilde{q}_k(S_0, \cdots, S_{n-1}) y^k = \sum_{k=0}^{M} \prod_{j=0}^{n-1} S_j^{k \cdot m_{j,n}} \tilde{q}_k(S_0, \cdots, S_{n-1}),$$

with $\deg(\tilde{q}_k) \leqq M$. We have thus established that $\# Q_1 \leqq \varphi(n-1, M+1, M)$. Then $\varphi(n, 1, M) = \varphi(n-1, 2, 0) + \varphi(n-1, M+1, M)$. End $N = 1$.

$n > 0$, $N > 1$.

$$f = \sum_{i=1}^{N} \prod_{j=0}^{n} S_j^{r_{j,i}} Q_i(S_0, \cdots, S_n).$$

Write

$$f = \prod_{j=0}^{n} S_j^{r_{j,1}} Q_1(S_0, \cdots, S_n)[g],$$

where

$$g = 1 + \sum_{i=2}^{N} \prod_{j=0}^{n} S_j^{\tilde{r}_{j,i}} \tilde{Q}_i(S_0, \cdots, S_n) \quad \text{and} \quad \tilde{Q}_i = Q_i / Q_1.$$

Thus $\# f \leqq \varphi(n, 1, M) + \# g$. Using Lemma 4, $\# g \leqq 2 \# g' + 1$. It remains to analyze g'.

$$g' = \sum_{i=2}^{N} \prod_{j=0}^{n} S_j^{\tilde{r}_{j,i}} \left[\sum_{k=0}^{n} \tilde{r}_{k,i} \frac{S_k'}{S_k} \tilde{Q}_i + \tilde{Q}_i' \right]$$

$$= \sum_{i=2}^{N} \prod_{j=0}^{n} S_j^{\tilde{r}_{j,i}} \left[\sum_{k=0}^{n} \tilde{r}_{k,i} \frac{S_k'}{S_k} \frac{Q_i}{Q_1} + \frac{Q_1 Q_i' - Q_1' Q_i}{Q_1^2} \right]$$

$$= \frac{1}{Q_1^2} \left\{ \sum_{i=2}^{N} \prod_{j=0}^{n} S_j^{\tilde{r}_{j,i}} \left[\sum_{k=0}^{n} \tilde{r}_{k,i} \frac{S_k'}{S_k} Q_1 Q_i + Q_1 Q_i' - Q_1' Q_i \right] \right\}$$

$$= \frac{1}{Q_1^2} \left\{ \sum_{i=2}^{N} \prod_{j=0}^{n} S_j^{\tilde{r}_{j,i}} [\hat{Q}_i] \right\}.$$

Using Lemma 5 and observing that

$$Q_i' = \sum_{k=0}^{n} \frac{\partial Q_i}{\partial S_k} S_k',$$

we see that

$$\hat{Q}_i = \frac{\hat{p}_i}{\prod_{k=0}^{n} S_k} \quad \text{with } \deg(\hat{p}_i) \leqq n + 2M.$$

Therefore

$$\#g' \leqq \#Q_1 + \#\left\{ \sum_{i=2}^{N} \prod_{j=0}^{n} S_j^{\hat{r}_{j,i}-1} \hat{p}_i \right\}$$

$$\leqq \varphi(n, 1, M) + \varphi(n, N-1, n+2M).$$

So $\#f \leqq \varphi(n, N, M)$, where $\varphi(n, N, M) = \varphi(n, 1, M) + 2[\varphi(n, 1, M) + \varphi(n, N-1, n+2M)] + 1$. □

Unfortunately, the bound obtainable from this induction is not very useful. Let $\rho(k) = \max\{\#p | p \text{ is a polynomial computable in } k \pm \text{operations}\}$. Then $\rho(k) \leqq \varphi(k, 1, 0)$.

Because $\varphi(n, N, M)$ is defined in terms of $\varphi(n, N-1, n+2M)$, we are exponentiating the third argument M in order to reduce the second argument N to 1. Therefore, since $\varphi(n, 1, M)$ is defined in terms of $\varphi(n-1, M+1, M)$, we exponentiate the third argument M in order to decrease the first argument by one. That is, our induction will result in a bound for $\varphi(n, 1, 0)$ which grows (very roughly) like

$$n \left(\begin{array}{c} 2^n \\ \cdot \\ \cdot \\ 2 \end{array} \right.$$

If $\varphi(n, N, M)$ were defined in terms of $\varphi(n, N-1, n+M)$ rather than $\varphi(n, N-1, n+2M)$, the result would be a bound for $\varphi(n, 1, 0)$ which grows (very roughly) like n^{2^n}. We are hoping that $\rho(n) \leqq c^n$ for some constant c, but we do not believe that our induction can be used to yield such a bound.

FACT 2. $\rho(k) \geqq 3^k$.

Proof. By induction on k, we can exhibit a suitable polynomial p_k having 3^k distinct simple real roots.

$k = 0$. $p_0(x) = x$.

Induction step. Choose ε sufficiently small that

$$p_{k+1}(x) = ([p_k(x)]^2 - \varepsilon^2) \cdot p_k(x)$$

is as desired. □

If indeed $\rho(k)$ were bounded by c^k for some constant c, we would feel more justified in thinking of ρ as a "$\pm$ analogue for degree". For consider $u(k) = \max\{\deg(p) | p \text{ is computable in } k * \text{ operations}\}$.

FACT 3. $u(k) = 2^k$.

Proof. $u(k) \leqq 2^k$ follows from Kung [3] or Strassen [8] and $p_k(x) = x^{2^k}$ trivially establishes $u(k) \geqq 2^k$. □

The scenario we would like to see developed is that a simple exponential bound on $\rho(k)$ is derived and then extended to (sets of) multivariate polynomials. The goal is to establish a nonlinear lower bound. We conclude with some suggestions for further research along these lines.

1. Can we redefine the bound by changing the definition of $\#p$? For example, consider $\#p$ = number of *integer* zeros in p and $\rho(k) = \max\{\#p | p$ computable in $k \pm$ operations$\}$.

2. Can we derive bounds for computations over $C(x)$ rather than just $R(x)$? Theorem 2 does not apply because Rolle's theorem (or Lemma 4) does not apply. For p in $C[x]$, we might consider $\#p = \min\{\deg r | r \neq 0$ in $R[x, y]$ and $z = x + yi$ is a zero of p implies $r(x, y) = 0\}$. This definition is motivated by looking at those polynomials computable in one $\pm$ operation.

3. Can we extend the development to multivariate polynomials? For example, consider $\#p$ = number of isolated zeros in p. We say $\langle x_1, \cdots, x_m\rangle$ is an *isolated* zero if there is a sufficiently small neighborhood around $\langle x_1, \cdots, x_n\rangle$ containing no other zeros. Here we are motivated by the fact that if $q(x)$ has n distinct zeros, then $p(x_1, \cdots, x_m) = q(x_1)^2 + \cdots + q(x_m)^2$ has n^m isolated zeros.

Acknowledgment. We would like to thank Zvi Kedem and Ludmilla Revah for many helpful discussions.

REFERENCES

[1] A. Borodin and I. Munro, *Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[2] E. C. Belaga, *Some problems in the computation of polynomials*, Dokl. Akad. Nauk. SSSR, 123 (1958), pp. 775–777.

[3] H. J. Kung, *The computational complexity of algebraic numbers*, Proc. of 5th Ann. ACM Symp. on Theory of Computing, May 1973, pp. 152–159.

[4] D. E. Knuth, *Seminumerical Algorithms*, The Art of Computer Programming, vol. II, Addison-Wesley, Reading, Mass., 1969.

[5] T. S. Motzkin, *Evaluation of polynomials and evaluation of rational functions*, Bull. Amer. Math. Soc., 61 (1955), p. 163.

[6] V. Y. Pan, *Methods of computing values of polynomials*, Russian Math. Surveys, 21 (1966), no. 1.

[7] M. Paterson and L. Stockmeyer, *On the number of nonscalar multiplications necessary to evaluate polynomials*, this Journal, 2 (1973), pp. 60–66.

[8] V. Strassen, *Die Berechnungskomplexitat von elementarsymmetrischen Funktunen und von Interpolationskoeffizienten*, Numer. Math., 20 (1973), pp. 238–251.

[9] ———, *Polynomials with rational coefficients which are hard to compute*, this Journal, 3 (1974), pp. 128–149.

[10] B. L. Van der Waerden, *Modern Algebra*, vol. 1, Frederick Ungar, New York, 1964.

[11] S. Winograd, *On the number of multiplications to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

SIAM J. COMPUT.
Vol. 5, No. 1, March 1976

A COMPARISON OF TWO ALGORITHMS FOR GLOBAL DATA FLOW ANALYSIS*

KEN KENNEDY†

Abstract. The problem of determining the points in a program at which variables are "live" (will be used again) is introduced and discussed. Two solutions, one which uses a simple iterative algorithm and one which uses an algorithm based on "Cocke–Allen interval" analysis, are presented and analyzed. These algorithms are compared on "self-replicating" families of reducible program flow graphs. The results are inconclusive in that the interval method requires fewer bit-vector steps on some graphs and more on others. If n is the number of nodes in a program flow graph and the number of edges is linearly proportional to n, then both algorithms require $O(n^2)$ steps in the worst case.

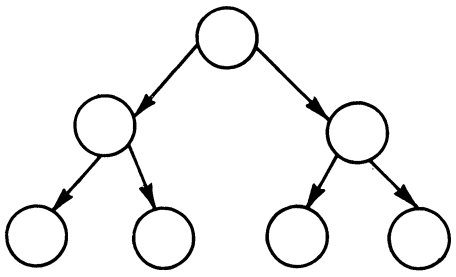
Key words. optimization of compiled code, compiler, flow graph reducibility, interval analysis, live variables, algorithmic complexity

1. Introduction. When analyzing computer programs for the purpose of code optimization, there is a class of problems which require the construction of data flow information from the control flow graph. One such problem is that of determining which variables are "live"—i.e., which variables will be used again—at any given point in the program.

An algorithm that uses "Cocke–Allen interval" analysis [1]–[5] to solve this problem was presented by the author in [6]. An alternate iterative method has been described by several authors [7], [8], [9]. This paper attempts to compare these two methods for complexity. The results, although inconclusive, provide some insights into the general nature of these methods and of the problem itself.

2. Background. The flow analysis of a program usually begins with the program expressed in some intermediate text which is scanned and subdivided into *basic blocks*, sequences of instructions which are always executed in order. If the first instruction of a basic block is executed, every instruction in that block will be executed. After the last instruction in a block, control may transfer to any one of a number of basic blocks called the *successors* of the block just executed.

For the purposes of the analysis in this paper, it is convenient to consider extended basic blocks as blocks. An *extended basic block* is just a tree of blocks:



The use of such blocks will usually reduce substantially the size of the problem to be dealt with by global analysis algorithms.

* Received by the editors August 9, 1974, and in revised form April 21, 1975.

† Department of Mathematical Sciences, Rice University, Houston, Texas 77001. This work was supported by the National Science Foundation under Grant NSF-OCA-GJ-40585.

We may represent the program in question by its *control flow graph* (or *flow graph*) in which each node stands for an extended basic block and each edge represents a possible branch from such a block to one of its successors; that is, if y is a successor of x, then there is a directed edge from x to y in the flow graph. A flow graph is therefore a triple $G = (N, E, n_0)$, where

(i) N is a finite set of nodes,
(ii) E is a finite set of edges (a subset of $N \times N$), and
(iii) n_0 is the *program entry node*—the unique node which is the successor of no other node in N.

The inverse of the successor relation in a flow graph may also be defined. Node x is a *predecessor* of y if and only if y is a successor of x, i.e., if and only if there is an edge (x, y) in E. We define the following two notations:

$$S(x) = \{y \in N | (x, y) \in E\} \quad \text{and} \quad P(x) = \{y \in N | (y, x) \in E\}.$$

Thus $S(x)$ is the set of successors of x and $P(x)$ is the set of its predecessors. If N_1 is a set of nodes, $N_1 \subset N$, then

$$S[N_1] = \bigcup_{x \in N_1} P(x).$$

That is, $S[N_1]$ is the set of successors of nodes in N_1. Similarly,

$$P[N_1] = \bigcup_{x \in N_1} P(x)$$

A *path* from x_1 to x_k is a sequence of nodes $(x_1, \cdots, x_k)$ such that (x_i, x_{i+1}) is in E for $1 \leqq i < k$. The *path length* of $(x_1, \cdots, x_k)$ is $k - 1$. We say that the path above is a *cycle* if $x_1 = x_k, k > 1$. A *simple path* is a path which contains no cycles.

2.1. Intervals. Let G be a flow graph and h a node of G. The (maximum) *interval with header h*, denoted by $I(h)$, is constructed by the following algorithm due to Cocke and Allen [3].

ALGORITHM A (Maximum interval construction).
Input. Flow graph G and designated node h.
Output. The set of nodes $I(h)$.
Method.[1]

```
begin
    I(h) := {h};
        # Iteratively add nodes that have all their predecessors in
          I(h) #
    while ∃x ∈ S[I(h)] − I(h)
          such that P(x) ⊂ I(h)
    do
          I(h) := I(h) ∪ {x}
    od
end  □
```

We note here that the order in which nodes are added to $I(h)$ will be important

[1] Pound signs (#) are used to delimit comments.

later and is called *interval order.* Interval order is by no means unique (in fact, it depends on the order in which successors of $I(h)$ are considered); however, it imposes a total order on the nodes of $I(h) - \{h\}$ which preserves the partial order defined by the successor relation. The result is that if we process the nodes of $I - \{h\}$ in interval order, we will process a node only after we have processed each of its predecessors in I, and if we process in reverse interval order, we will process a node only after we have processed each of its successors in $I - \{h\}$.

From Algorithm A we can proceed directly to a second algorithm, due also to Cocke and Allen [3], which partitions the entire flow graph into a set of disjoint intervals.

ALGORITHM B (Interval partition).
Input. Flow graph $G = (N, E, n_0)$.
Output. The set of intervals INTS(G).
Method. We use a set H of interval headers.

```
begin # The program entry node is a header #
    H := {n0};
    INTS(G) := ∅;
    while H ≠ ∅ # there are more headers #
        do
            x := an arbitrary node in H;
            H := H − {x};
            Find I(x) using Algorithm A;
            INTS(G) := INTS(G) ∪ {I(x)}
                # Successors of I(x) which cannot be added are
                  headers #
            H := H ∪ (S[I(x)] − I(x))
        od
end □
```

As an example of this process, consider the flow graph in Fig. 1. The interval heads chosen by Algorithm B will be 1, 2, 5 and the intervals are $\{1\}, \{2, 3, 4\}$, and $\{5, 6, 7\}$.

If G is a flow graph, then the derived flow graph of G, denoted by $I(G)$, is defined as follows:

(a) The nodes of $I(G)$ are the intervals in INTS(G).
(b) If J, K are two intervals, there is an edge from J to K in $I(G)$ if and only if there exist nodes $n_J \in J$ and $n_K \in K$ such that n_K is a successor of n_J in G. Note that n_K must be the header of K.
(c) The initial node of $I(G)$ is $I(n_0)$.

We call G the *underlying flow graph* of $I(G)$.

The sequence $(G_0, G_1, \cdots, G_m)$ is called the *derived sequence* for G if $G = G_0$, $G_{i+1} = I(G_i)$, $G_{m-1} \neq G_m$, and $I(G_m) = G_m$. G_i is called the *derived graph of order i* and G_m is the *limit flow graph* of G. A flow graph is said to be *reducible* if and only if its limit flow graph is the *trivial flow graph,* a single node with no edge; otherwise, the flow graph is *nonreducible.*

Let $I(h)$ be an interval of flow graph G. We classify the edges which branch out of nodes of $I(h)$ three ways.

1. Those edges which branch from nodes in $I(h)$ to h are called *latches* of $I(h)$.

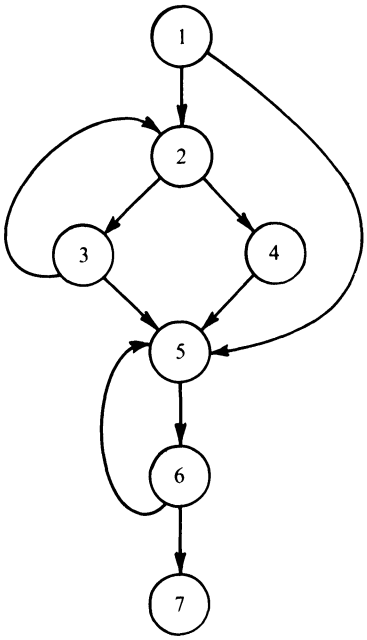


FIG. 1. *A flow graph*

2. Those edges which branch from nodes in $I(h)$ to nodes outside the interval are called *exit edges* of $I(h)$.
3. All other edges are called *forward edges* of $I(h)$. All forward edges branch from nodes of $I(h)$ to nodes of $I(h) - \{h\}$.

2.2. Live-dead analysis. A path in a flow graph G is said to be *definition-clear* with respect to a variable X, or *X-clear*, if there is no assignment to X in any node on that path. A variable X is *live* at a point p in the flow graph if there exists an X-clear path from p to a use of X. If there is no such path, X is said to be *dead* at p. Thus X is live at p if its value at p may be used before it is redefined.

Our aim is to compute, for each node b in the control flow graph, the set $L(b)$ of all variables which are live on entry to b. The global sets $L(b)$ can be defined in terms of two sets which contain strictly local information. Given an extended basic block b in the flow graph:

1. IN(b) is the set of variables X such that there is an X-clear path from the entry of b to a use of X within b; i.e., a use of X appears in b before any redefinition of b.
2. $T(b, y)$, defined for every successor y of b, is the set of variables X for which there is an X-clear path through b; i.e., the set of variables which are not defined in b on the path from b to y.

THEOREM 1. *Let $G = (N, E, n_0)$ be a flow graph. For any $b \in N$,*

$$L(b) = \mathrm{IN}(b) \cup \bigcup_{y \in S(b)} (T(b, y) \cap L(y)). \tag{1}$$

Proof. There is an X-clear path from the entry of b to a use of X if and only if
(a) there is such a path to a use within b, i.e., $X \in \text{IN}(b)$, or
(b) there is an X-clear path through b to some successor $y \in S(b)$ *and* there is an X-clear path from the entry of y to a use of X. This can be expressed $X \in \bigcup_{y \in S(b)} (T(b, y) \cap L(y))$.

The "or" of conditions (a) and (b) is expressed by union. Thus

$$X \in L(b) \Leftrightarrow X \in (\text{IN}(b) \cup \bigcup_{y \in S(b)} (T(b, y) \cap L(y))). \quad \square$$

2.3. A simple, iterative bit propagation algorithm [7], [8], [9], [10]. We now present a simple algorithm for computing the sets $L(b)$. This algorithm uses bit-vectors to represent the requisite sets—where each bit-vector has one bit position for each possible variable X. The bit for X is on if X is in the set and off otherwise.

Let LVIN(b) be the bit-vector for the set $L(b)$, INSIDE(b) the bit-vector for IN(b), and THRU(b, y) the bit-vector for $T(b, y)$. The following relations hold:

1. By analogy with Theorem 1,

$$\text{LVIN}(b) = \text{INSIDE}(b) \vee \bigvee_{y \in S(b)} (\text{THRU}(b, y) \wedge \text{LVIN}(y)).$$

2. For exit blocks of the program (blocks with no successors), the equation reduces to

$$\text{LVIN}(b) = \text{INSIDE}(b).$$

The following algorithm, adapted from [7], computes these bit vectors by iterating until a steady state condition is detected.

ALGORITHM C. (Compute LVIN vectors).

Input. 1. Flow graph $G = (N, E, n_0)$, $|N| = n$. The nodes are numbered from 1 to n in some strategic way. Each node is referred to by its number.
2. Bit-vectors INSIDE(j), THRU(j, k), $1 \leqq j \leqq n$, k ranging over successors of j.

Output. Bit-vectors LVIN(j), $1 \leqq j \leqq n$.

Method.

```
begin # Initialize LVIN vectors #
    for j from 1 to n
        do LVIN(j) := INSIDE(j) od;
     # Iteratively apply the equation for LVIN(j) at each node
       until no more changes are detected #
    change := true;
C1: while change
        do change := false;
           for j from 1 to n do
              oldLVIN := LVIN(j);
              LVIN(j) := LVIN(j) ∨ ∨_{k∈S(j)} (THRU(j, k) ∧ LVIN(k));
              if LVIN(j) ≠ oldLVIN then
                    change := true fi
           od
        od
end □
```

Termination and correctness of Algorithm C have been shown in [10]. A *backward branch* of a reducible flow graph G is an edge which becomes a latch in some interval of the derived sequence for G. Let d be the maximum number of backward branches in any simple path within G. The following theorem is from Hecht and Ullman [7].

THEOREM 2. *If the numbering of nodes in G (where G is reducible) is chosen suitably (by a method defined in* [7]*), the body of the while-loop (labeled* C1*) in Algorithm* C *will be executed at most* $d + 2$ *times.*

COROLLARY 2.1. *If* $e = |E|$ *is the number of edges in G (where G is reducible), then Algorithm* C *requires at most* $2e(d + 2)$ *bit-vector operations.*

Proof. Each time the body of the loop is executed, two bit-vector operations,

$$\text{LVIN}(j) \vee (\text{THRU}(j, k) \wedge \text{LVIN}(k)),$$

are executed for each edge (j, k) in the program, for a total of $2e$ such operations. But the loop is iterated at most $d + 2$ times. □

The upper bound presented in Corollary 2.1 is different from that derived by Hecht and Ullman [7] because we assume that we are dealing with extended basic blocks to facilitate the comparisons of § 4. If simple basic blocks are used, then the THRU vectors have only one exit, and the computation can be performed in $(d + 2)(e' + n')$, where e' and n' are the number of edges and nodes, respectively, in the flow graph G' in which nodes represent simple rather than extended basic blocks. Clearly $e' \geqq e$ and $n' \geqq n$, so the Hecht–Ullman bound is not necessarily less than the Corollary 2.1 bound; however, this assumption may favor the interval algorithm to be presented in the next section.

Note also that neither of these upper bounds is "tight" unless we assume that truly global data flow is present in the graph. By this we mean that the estimate is exact if we assume that any simple path in the graph is the shortest X-clear path from a node where S is live to a use of X for some variable X. We use this assumption implicitly in the comparisons of § 4.

3. The interval method. An alternate method which uses interval analysis on reducible flow graphs was presented by the author in [6]. The method consists of two parts. First, the sets IN and T are computed for intervals of the derived graph, then for intervals of the next derived graph, and so on until these sets are available for the trivial limit graph. Then (1) (from Theorem 1) is applied first to the highest order derived graph, then to the nodes of its graph, and so on until $L(b)$ has been computed for each node in the original flow graph. Thus there are two passes through the nodes and edges of the derived sequence. We treat these passes separately.

Pass 1. The purpose of pass 1 is to compute, for each interval in the sequence of derived graphs, the sets IN and T. The following bit-vector algorithm computes these sets for an interval, given their values for the nodes of that interval. Note that it passes through the interval in interval order.

ALGORITHM D (INSIDE and THRU for an interval).

Input. 1. The nodes of an interval I numbered from 1 to n_I in interval order.

2. Successor and predecessor information for each node in the interval.

3. The set of successor intervals J of I (successors in the derived graph) along with their headers h_J.
4. INSIDE(j), $1 \leqq j \leqq n_I$.
5. THRU(j, k), $1 \leqq j \leqq n_I$, k ranging over all successors of j.

Auxiliary variables.

1. PATHOUT(j, k), $1 \leqq j \leqq n_I$, k ranging over all successors of j which are not the header of I. PATHOUT will be a bit vector representing the set of variables X such that there is an X-clear path from interval entry through node j to the exit from j leading to k.
2. PATH(j), $2 \leqq j \leqq n_I$. PATH(j) will be a bit vector representing the set of variables X such that there is an X-clear path from interval entry to the entry of node j. PATH(1) is not needed because it would consist simply of the vector with all 1's.

Output. INSIDE(I) and THRU(I, J) for each successor J of I in the derived graph.

Method.

```
begin # Initialize INSIDE(I) and PATHOUT #
      INSIDE(I) := INSIDE(1);
      for all k ∈ S(1)
         do
            D1: PATHOUT(1, k) := THRU(1, k)
         od;
            # Iterate through I − {h} in interval order #
      for j from 2 to n_I
         do
            D2: PATH(j) := ∨_{k∈P(j)} PATHOUT(k, j);
            D3: INSIDE(I) := INSIDE(I) ∨ (PATH(j) ∧ INSIDE(j));
            D4: for each m ∈ S(j) − {1}
                   do
                      PATHOUT(j, m) := PATH(j) ∧ THRU(j, m)
                   od
         od;
            # Compute THRU vectors for I #
      for all J ∈ S(I) with header h_J
         do
            D5: THRU(I, J) := ∨_{k∈P(h_J)∩I} PATHOUT(k, h_J)
         od
end □
```

THEOREM 3. *Algorithm* D *terminates and is correct.*

Proof. Termination is trivial since the loops can only be executed a finite number of times.

Correctness is shown by three lemmas. Note that we write "$X \in$ BITVECTOR" when we mean "the position for X in BITVECTOR is 1".

LEMMA 3.1. *For every* j, $2 \leqq j \leqq n_I$, $X \in$ PATH(j) *if and only if there is an* X*-clear path to* j *from interval entry.*

Proof of Lemma 3.1. The proof proceeds by induction on j.

(a) *Basis.* $j = 2$. The only predecessor of node 2 must be the interval header, node 1. Therefore, PATH(2) = PATHOUT(1, 2) = THRU(1, 2) because of steps D1 and D2. But $X \in$ THRU(1, 2) if and only if there is an X-clear path from the header's entry, which is identical to the interval entry, to node 2.

(b) *Induction step.* Assume that the lemma is true for $1 \leqq j \leqq p - 1$ where $p \leqq n_I$. By step D2, $X \in$ PATH(p) if and only if $X \in$ PATHOUT(k, p) for some predecessor k of p. But this is true (step D4) if and only if $X \in$ PATH(k) and $X \in$ THRU(k, p). Since we are processing in interval order, k must have been processed on an earlier step so $k < p$. Therefore $X \in$ PATH(p) if and only if, for some predecessor k of p, there is an X-clear path from interval entry to k (by the induction hypothesis) and there is an X-clear path through k to p. But this can be true if and only if there is an X-clear path from interval entry to p. □

LEMMA 3.2. *After execution of Algorithm* D, $X \in$ INSIDE(I) *if and only if there is an X-clear path from interval entry to a use of X within I.*

Proof of Lemma 3.2. $X \in$ INSIDE(I) if and only if $X \in$ INSIDE(1) (initial statement) or for some j, $2 \leqq j \leqq n_I$, $X \in$ INSIDE(j) and $X \in$ PATH(j) (step D3).

If $X \in$ INSIDE(1), then there is an X-clear path from interval entry to a use of X within I—the path from the entry of node 1 to the use within 1. Assume $X \notin$ INSIDE(1); then for some j, $2 \leqq j \leqq n_I$, $X \in$ INSIDE(j) and $X \in$ PATH(j). These are simultaneously true if and only if there is an X-clear path to the entry of j (by Lemma 3.1) and an X-clear path to a use within j, but this is equivalent to the existence of an X-clear path to a use within I.

LEMMA 3.3. *After Algorithm* D *is executed,* $X \in$ THRU(I, J), *where J is a successor of I, if and only if there is an X-clear path through I to J.*

Proof of Lemma 3.3. Let h_J be the header of J. By step D5, $X \in$ THRU(I, J) if and only if h_J has a predecessor $k \in I$ such that $X \in$ PATHOUT(k, h_J). But by step D3, $X \in$ PATHOUT(k, h_J) if and only of $X \in$ PATH(k) and $X \in$ THRU(k, h_J), which is true if and only if there is an X-clear path from the entry of I to k (by Lemma 3.1) and there is such a path through k to h_J. This last is equivalent to the existence of an X-clear path through I to h_J. □

Lemmas 3.2 and 3.3 establish the theorem. □

Recall that n is the number of nodes in the interval I. Let e_I be the number of edges leaving nodes of that interval, e_I^f be the number of forward edges of I, e_I^o be the number of exit edges of I, and e_I^b be the number of latches of I. Note that $e_I = e_I^f + e_I^o + e_I^b$. Let N_I^J be the number of successors of I in the derived graph.

THEOREM 4 (Complexity of Algorithm D). *Algorithm* D *requires at most*

$$2(e_I - e_I^b) + n_I - N_I^J - 2$$

bit-vector operations.

Proof. One operation to compute PATHOUT is required for each edge leaving a node of I except those edges which leave or branch back to the header. Except in the trivial case that the header is an exit node, at least one edge must leave the header, so at most $e_I - e_I^b - 1$ operations are required for this computation. At each node other than the header, the computation of PATH requires one less bit-vector operation than the number of edges entering that node. The number of edges which enter nodes other than the header is e_I^f, so $e_I^f - n_I + 1$

operations are required to compute PATH. At each node other than the header, two bit-vector operations are required to compute INSIDE(I) for a total of $2n_I - 2$. For each successor interval J, the computation of THRU(I, J) requires one less operation than the number of edges entering J from I or a total of $e_I^o - N_I^J$ operations. Summing these, we get

$$(e_I - e_I^b - 1) + (e_I^f - n_I + 1) + (2n_I - 2) + (e_I^o - N_I^J)$$
$$= (e_I - e_I^b) + (e_I^f + e_I^o) + n_I - N_I^J - 2.$$

Using the identity $e_I^f + e_I^o = e_I - e_I^b$ yields

$$2(e_I - e_I^b) + n_I - N_I^J - 2. \qquad \square$$

Pass 1 of the live analysis merely consists of applying Algorithm D to the intervals of each derived graph starting with the first order derived graph and ending with the limit flow graph. This will be incorporated into Algorithm F below.

Pass 2. Pass 2 will use the THRU and INSIDE vectors computed in pass 1 to compute the vectors LVIN for each block in the program using a bit-vector analogue of (1) in Theorem 1. If $G_m = (N_m, E_m, n_0^m)$ is the limit flow graph of the reduction sequence, consisting of the single node n_0^m with no edges, clearly LVIN(n_0^m) = INSIDE(n_0^m). Thus we know the input LVIN vector for the interval represented by n_0^m.

We now present an algorithm which, given the LVIN information for an interval I and all its successor intervals J, computes the LVIN vectors for each node within that interval.

ALGORITHM E (LVIN Computation using intervals).

Input. 1. An interval I with nodes numbered from 1 to n_I in interval order.
2. LVIN(I).
3. All successors J of I with their headers h_J,
4. LVIN(J) for all successors J of I.
5. INSIDE(j), $1 \leqq j \leqq n_I$.
6. THRU(j, k), $1 \leqq j \leqq n_I$, k ranging over all successors of j.

Output. LVIN(j), $1 \leqq j \leqq n_I$.

Method.

```
begin # LVIN of the header of an interval = LVIN of the interval #
E1:  LVIN(1) := LVIN(I);
     for each J ∈ S(I) with header h_J
        do
           E2: LVIN(h_J) := LVIN(J)
        od;
           # Iterate through I in reverse interval order applying the
             LVIN equation #
     for j from n_I to 2 by -1
        do
           E3: LVIN(j) := INSIDE(j) ∨ ∨_{k∈S(j)} (THRU(j, k)
                                  ∧ LVIN(k))
        od
end  □
```

THEOREM 5 (Termination and correctness of Algorithm E). *Algorithm* E *terminates and is correct.*

Proof. Termination follows from the finiteness of I.

LEMMA 5.1. *Whenever step* E3 *is executed,* LVIN(k) *has already been computed for each successor* k *of* j.

Proof of Lemma 5.1. A successor k of j must be one of three possibilities:

1. The header h_J of a successor interval J, in which case LVIN(k) was computed in step E2.
2. The header of the interval I, in which case LVIN(k) was computed in step E1.
3. Another node in $I - \{1\}$, in which case LVIN(k) was computed on a previous execution of step E3. This follows because step E3 is applied to the nodes of $I - \{1\}$ in reverse interval order which assures us that E3 will be applied to successors of j before it is applied to j. □

Lemma 5.1 and Theorem 1 are sufficient to prove Theorem 5. □

THEOREM 6 (Complexity of Algorithm E). *Let* e_I *be the number of edges leaving nodes of* I. *Algorithm* E *requires at most* $2(e_I - 1)$ *bit-vector operations.*

Proof. Step E3 is applied once to each node except the header. For each edge leaving such a node, there is one bit-vector operation to compute LVIN(k) $\wedge$ THRU(j, k) and one to "or" the result with INSIDE(j). Thus 2 bit-vector operations are required for each edge leaving a node of the interval except those leaving the header. There must be at least one edge leaving the header, so $2(e_I - 1)$ steps are required. □

3.1. The complete algorithm. We now present the complete algorithm to perform live analysis using the interval method.

ALGORITHM F (Live analysis).

Input. 1. A reducible flow graph $G = (N, E, n_0)$.
2. The derived sequence $(G_0, G_1, \cdots, G_m)$ for G, where $G = G_0$ and G_m is the trivial flow graph. $G_i = (N_i, E_i, n_0^i)$.
3. INSIDE(b) for every $b \in N$.
4. THRU(b, k) for every $(b, k) \in E$.
5. An interval order listing of the contents of each interval in the derived sequence.

Output. LVIN(b) for every $b \in N$.

Method.

```
begin # Iterate through the derived sequence in inner-to-outer order
        applying Algorithm D #
    for i from 1 to m
        do
            F1: for each node x in G_i # x is an interval #
                do
                    Apply Algorithm D to compute INSIDE and THRU
                    vectors for x
                od

        od;
        # Compute LVIN for the whole program #
```

```
        LVIN(n_0^m) := INSIDE(n_0^m);
            # Iterate through the derived sequence in outer-to-inner order
              applying Algorithm F #
        for i from m to 1 by -1
            do
                F2: for each node x in G_i # x is an interval #
                    do
                        Apply Algorithm E to compute LVIN vectors for the
                        nodes contained in x
                    od
            od
    end □
```

THEOREM 7. *Algorithm* F *terminates and is correct.*

Proof. Termination follows from the finiteness of m, which follows from the reducibility of G_0.

Correctness follows from Theorems 3 and 5 and the following observations:

1. In step F1, whenever Algorithm D is applied to an interval, the THRU and INSIDE vectors for nodes of that interval have been computed, because F1 is applied to the derived graphs in increasing order.

2. In step F2, whenever Algorithm E is applied to an interval LVIN vectors have been computed for entry to that interval and its successor intervals because F2 is applied to the derived graphs in decreasing order (i.e., the limit graph first, then its underlying graph, and so on). □

Consider a derived sequence $(G_0, G_1, \cdots, G_m)$ for flow graph $G = (N, E, n_0)$. Let $e_i = |E_i|$ and $n_i = |N_i|$, $1 \leqq i \leqq m$. Let e be the number of edges in the original flow graph and n be the number of its nodes. Finally, let l_i be the number of latches in G_i—i.e., the number of edges in E_i which branch to headers when Algorithm B is applied to G_i.

THEOREM 8 (Complexity of Algorithm F). *Algorithm* F *requires at most*

$$4e + n - 4 + 3\sum_{i=1}^{m-1}(e_i - n_i) - 2\sum_{i=0}^{m-1} l_i$$

bit-vector operations.

Proof. Let us consider the derived graph $G_i = (N_i, E_i, n_0^i)$ where $i > 1$. Each node in this graph represents an interval I. The nodes contained in I are nodes of the graph G_{i-1}. Let e_I be the number of edges in E_{i-1} which leave nodes in I. Then

$$\sum_{I \in N_i} e_I = e_{i-1}$$

because the nodes contained in all $I \in N_i$ comprise all nodes in N_{i-1}. For the same reason, if n_I is the number of nodes in I then

$$\sum_{I \in N_i} n_I = n_{i-1},$$

which is the number of nodes in the underlying graph. Let e_I^b be the number of

latches within $I \in N_i$. Then

$$\sum_{I \in N_i} e_I^b = l_{i-1},$$

the number of latches in the underlying graph. If N_I^J is the number of $J \in N_i$ which are successors of I, then

$$\sum_{I \in N_i} N_I^J = e_i,$$

because this sum merely counts the predecessor-successor pairs in G_i.

Trivially,

$$\sum_{I \in N_i} 1 = n_i.$$

These summation identities will be used in our evaluation.

Recall that by Theorem 4, each execution of Algorithm D requires

$$2e_I - 2e_I^b + n_I - N_I^J - 2$$

operations. Step F1 calls Algorithm D once for each $I \in N_i$, so each execution of step F1 requires

$$\begin{aligned} &2 \sum_{I \in N_i} e_I - 2 \sum_{I \in N_i} e_I^b + \sum_{I \in N_i} n_I - \sum_{I \in N_i} N_I^J - 2 \sum_{I \in N_i} 1 \\ &\quad = 2e_{i-1} - 2l_{i-1} + n_{i-1} - e_i - 2n_i \end{aligned}$$

bit-vector operations. Step F1 is executed once for each i, $1 \leqq i \leqq n$, so the execution of the first pass requires

$$\sum_{i=1}^{m} (2e_{i-1} - 2l_{i-1} + n_{i-1} - e_i - 2n_i)$$

bit-vector operations.

By Theorem 6, each execution of Algorithm E in step F2 requires $2e_I - 2$ bit-vector operations. Since Algorithm E is called once for each $I \in N_i$ in step F2, that step requires

$$2 \sum_{I \in N_i} e_I - 2 \sum_{I \in N_i} 1 = 2e_{i-1} - 2n_i$$

bit-vector operations. Step F2 is repeated once for each i, $m \geqq i \geqq 1$, so the second pass requires $\sum_{i=1}^{m} (2e_{i-1} - 2n_i)$ bit-vector operations.

Adding these two sums, we get

$$\sum_{i=1}^{m} (4e_{i-1} - 4n_i + n_{i-1} - e_i - 2l_{i-1}).$$

If we note that $4e_0 + n_0 = 4e + n$, the bound becomes

$$4e + n + \sum_{i=2}^{m} (4e_{i-1} + n_{i-1}) - \sum_{i=1}^{m} (4n_i + e_i) - 2 \sum_{i=1}^{m} l_{i-1}.$$

Renumbering the first and third summations, combining terms and using $n_m = 1$

and $e_m = 0$, we get

$$4e + n - 4 + \sum_{i=1}^{m-1} (3e_i - 3n_i) - 2\sum_{i=0}^{m-1} l_i,$$

which is the desired result. □

We note here that we have used a subtle restriction on graphs in proving Theorems 4, 6 and 8. The restriction is merely this: no exit node may be self-looping; that is, no node in the graph may have itself as its only successor. This is not a serious restriction since self-looping nodes will not appear in any derived graph and a self-loop on an exit node will have no effect on the live analysis—in fact, such self-loops can be ignored with no change in the results.

4. Comparison of algorithms. We wish to compare the number of bit-vector steps required by Algorithm C and Algorithm F. However, this is difficult since the upper bounds are expressed in ways that are not strictly comparable. To accomplish the comparison we introduce the concept of *form*. A countable class of graphs $(G_0, G_1, G_2, \cdots,)$ have *self-replicating form* if

(i) $I(G_p) = G_{p-1}$;

(ii) each interval in G_p is either G_0 or G_1 (of the same class);

(iii) G_0 is the trivial flow graph.

We call G_p the *level p graph* of the given form. We shall see that it is usually possible to construct functions f_n, f_e, and f_l such that if $(G_0, G_1, G_2, \cdots)$ is a sequence of self-replicating graphs, $f_n(p)$ is the number of nodes in G_p, $f_e(p)$ is the number of edges, and $f_l(p)$ is the number of latches.

LEMMA 9. *Let $G = G_p$ be a level p self-replicating graph. The length m of the derived sequence for G_p is just p, and the derived graph of order i is just G_{p-i}.*

Proof. By property (i) above of self-replicating graphs, the derived sequence for G_p is just $(G_p, G_{p-1}, \cdots, G_0)$. Both conclusions follow trivially from this observation. □

LEMMA 10. *Let e_i be the number of edges in the i-th derived graph of G_p, and let n_i and l_i be the number of nodes and latches, respectively, in that derived graph. Let f_e, f_n and f_l be the edge, node and latch functions for graphs of the same form as G_p. Then if m is the length of the derived sequence for G_p,*

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{j=1}^{p-1} (f_e(j) - f_n(j))$$

and

$$\sum_{i=0}^{m-1} l_i = \sum_{j=1}^{p} f_l(j).$$

Proof. Note that e_i is just the number of edges in the ith derived graph, which is G_{p-i} by Lemma 9. Thus $e_i = f_e(p - i)$. Similarly, $n_i = f_n(p - i)$ and $l_i = f_l(p - i)$.

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{i=1}^{p-1} (f_e(p - i) - f_n(p - i)),$$

which, if $j = p - i$, becomes

$$\sum_{j=1}^{p-1} (f_e(j) - f_n(j)).$$

Similarly,

$$\sum_{i=0}^{m-1} l_i = \sum_{i=1}^{p-1} f_l(p - i).$$

Letting $j = p - i$, we get

$$\sum_{j=1}^{p} f_l(j).$$ □

In the comparisons below, we will informally define graphs of several different forms (through examples). As a notational convention we use $T(F)$ to represent the number of operations required by Algorithm F and $T(C)$ to represent the number required by Algorithm C. The ratio r_p is defined as

$$r_p = T(F)/T(C)$$

for a level p graph of the given form.

4.1. Seashell graphs. The "seashell graph" arises from nested computation loops with no loop exits other than the standard one. Its form is shown by the example in Fig. 2. It is clear from this example that the seashell graph has $p + 1$

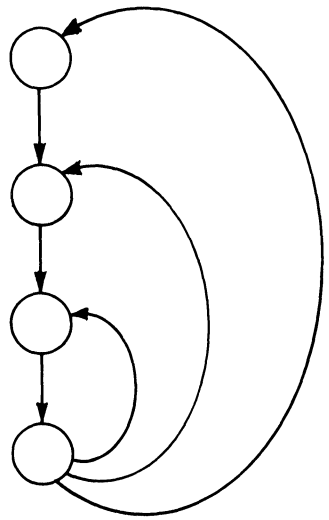


FIG. 2. *Seashell graph of level* $p = 3$

nodes, $2p$ edges, and 1 latch. The longest simple path with a backward branch has at most one such branch for any p, so $d = 1$. Thus Algorithm C will require

$$T(C) = 2e(d + 2) = 12p$$

bit-vector operations.

To evaluate Algorithm F, we need to evaluate the two summations

$$\sum_{i=0}^{m-1} l_i = \sum_{j=1}^{p} 1 = p \qquad \text{(by Lemma 10)},$$

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{j=1}^{p-1} (2j - j - 1)$$

$$= \sum_{j=1}^{p-1} (j - 1) = \sum_{j=1}^{p-2} j = \frac{(p-2)(p-1)}{2}.$$

The estimate for Algorithm F is

$$T(\mathrm{F}) = 4(2p) + (p + 1) - 4 + \frac{3(p-2)(p-1)}{2} - 2p$$

$$= 7p - 3 + \frac{3p^2 - 9p + 6}{2}$$

$$= \frac{3p^2 + 5p}{2} = \frac{3p + 5}{2} p.$$

The ratio r_p of $T(\mathrm{F})$ to $T(\mathrm{C})$ for a graph of level p is

$$r_p = \frac{3p + 5}{24}.$$

Thus, in this case, $r_p = O(p)$, and for large values of p, Algorithm C will be much better than Algorithm F. However, let us consider small values of p:

$$r_1 = \tfrac{1}{3}, \quad r_2 = \tfrac{11}{24}, \quad r_3 = \tfrac{7}{12}.$$

In fact, a simple solution of $(3p + 5)/24 < 1$ shows us that whenever $p < 7$, Algorithm F will require fewer bit-vector steps than Algorithm C.

It is interesting to note here that in tests on 50 randomly selected FORTRAN programs [11], Knuth found that none had more than 6 derived graphs (the average was less than 3). A seashell graph of level 7 or more has at least 7 derived graphs $(G_6, G_5, \cdots, G_0)$, so such graphs are probably rare.

4.2. Spiral graphs. While the seashell graph arises from deeply nested loops which involve only computation, nested loops with tests and loop exits will often result in "conditional seashell" or "spiral" graphs, whose form is shown in Fig. 3. The spiral graph has $2p + 1$ nodes, $4p$ edges, one latch and p backward branches. Since there is a simple path from the core to the exit which includes all backward branches, $d = p$. Thus

$$T(\mathrm{C}) = 2e(d + 2) = 8p(p + 2) = p(8p + 16).$$

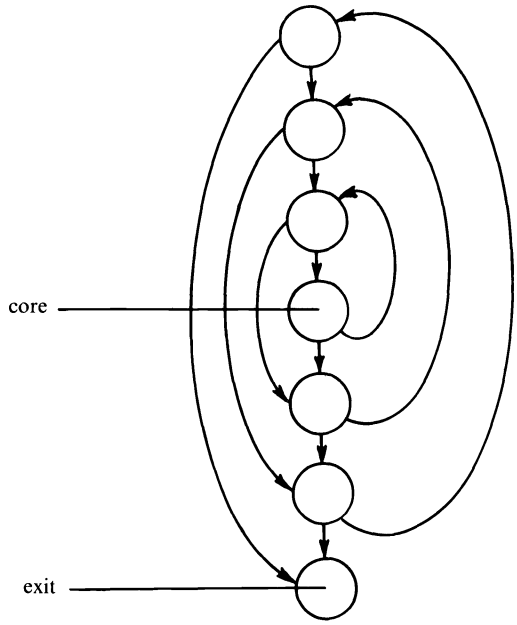


FIG. 3. *Spiral graph of level* $p = 3$

To evaluate Algorithm F (using Lemma 10),

$$\sum_{i=0}^{m-1} l_i = \sum_{j=1}^{p} 1 = p,$$

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{j=1}^{p-1} (4j - 2j - 1) = 2\sum_{j=1}^{p-1} j - (p-1)$$

$$= p(p-1) - (p-1) = p^2 - 2p + 1,$$

$$T(\mathrm{F}) = 4(4p) + 2p + 1 - 4 + 3p^2 - 6p + 3 - 2p$$

$$= 3p^2 + 10p = p(3p + 10).$$

The ratio $r_p = T(\mathrm{F})/T(\mathrm{C})$ is

$$r_p = \frac{3p + 10}{8p + 16} = O(1).$$

Note that $r_1 = \frac{13}{24}$, $r_2 = \frac{1}{2}$, and r_p is asymptotic to $\frac{3}{8}$ for large p. Thus Algorithm C requires approximately twice as many operations on this type of graph.

4.3. Hecht–Ullman binary graph. This example is taken from Hecht and Ullman [7]; hence the name. Its form is shown in Fig. 4. This graph has 2^p nodes, $2^{p+1} - 2$ edges, and 2^{p-1} latches for $p > 0$. Hecht and Ullman have shown that

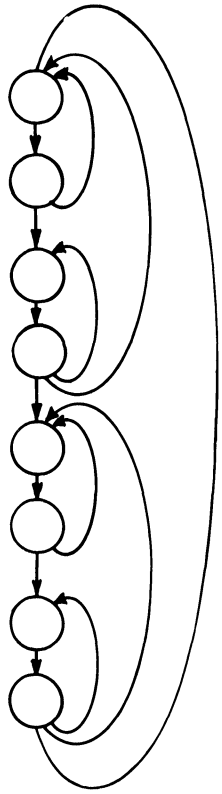


FIG. 4. *Hecht–Ullman binary graph of level* $p = 3$

$d = 1$, so

$$T(\mathrm{C}) = 6(2^{p+1} - 2) = 12(2^p - 1).$$

For Algorithm F (using Lemma 10),

$$\sum_{i=1}^{m-1} l_i = \sum_{j=1}^{p} 2^{j-1} = \sum_{j=0}^{p-1} 2^j = 2^p - 1,$$

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{j=1}^{p-1} (2^{j+1} - 2 - 2^j)$$

$$= \sum_{j=1}^{p-1} (2^j - 2) = 2 \sum_{j=1}^{p-1} (2^{j-1} - 1)$$

$$= 2 \sum_{j=0}^{p-2} (2^j - 1) = 2(2^{p-1} - 1) - 2(p - 1) = 2^p - 2p,$$

$$T(\mathrm{F}) = 4(2^{p+1} - 2) + 2^p - 4 + 3(2^p - 2p) - 2(2^p - 1)$$

$$= 10 \cdot 2^p - 10 - 6p.$$

The ratio is

$$r_p = \frac{10(2^p - 1) - 6p}{12(2^p - 1)} = O(1).$$

Note that $r_1 = \frac{1}{3}, r_2 = \frac{1}{2}, r_3 = \frac{13}{21}$, and the ratio is asymptotic to $\frac{5}{6}$ for large p.

4.4. Double-chain graph. The double-chain graph can be produced by a series of nested while-loops. Its form is shown in Fig. 5. This graph has $p + 1$ nodes,

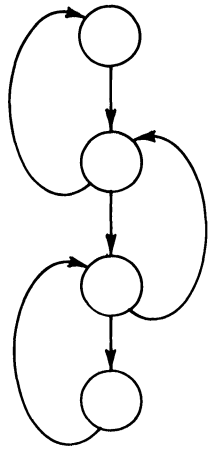


FIG. 5. *Double-chain graph of level* $p = 3$

$2p$ edges, and p latches. Clearly, $d = p$, so

$$T(\mathrm{C}) = 8p + 4p^2 = p(4p + 8).$$

For Algorithm F (using Lemma 10),

$$\sum_{i=0}^{m-1} l_i = \sum_{j=1}^{p} 1 = p,$$

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{j=1}^{p-1} 2_j - j - 1 = \sum_{j=0}^{p-2} j$$

$$= \frac{(p-1)(p-2)}{2} = \frac{p^2 - 3p + 2}{2},$$

$$T(\mathrm{F}) = 4(2p) + (p + 1) - 4 + \frac{3p^2 - 9p + 6}{2} - 2p = \frac{(3p + 5)p}{2}.$$

The ratio is

$$r_p = \frac{3p + 5}{8p + 16} < \frac{3(p + 2)}{8(p + 2)} = \frac{3}{8}.$$

Thus the ratio is always less than $\frac{1}{2}$. Algorithm C always requires at least twice as many operations on this graph. Note that essentially the same effect is achieved by the spiral graph for which r_p is also asymptotic to $\frac{3}{8}$.

4.5. Diamond graph. A diamond graph, whose form is shown in Fig. 6, arises

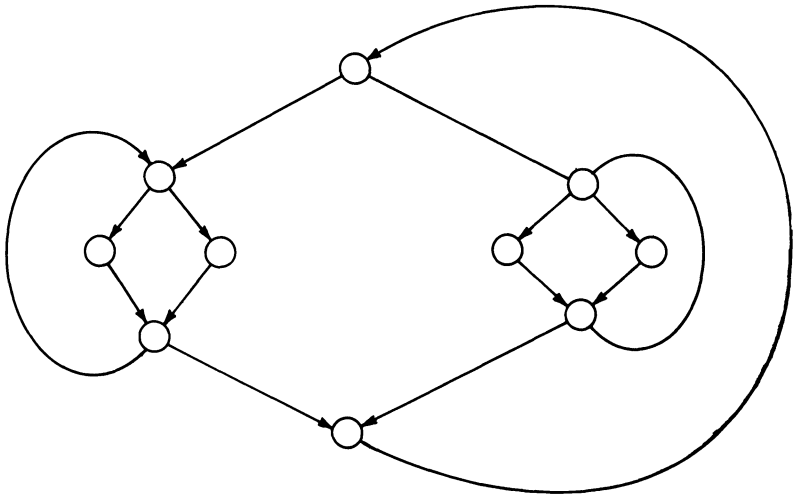


FIG. 6. *Diamond graph of level* $p = 2$

from if-then-else clauses within loops which are themselves in if-then-else clauses, etc. The reader can verify that a diamond graph has $3 \cdot 2^p - 2$ nodes, $5 \cdot 2^p - 5$ edges and 2^{p-1} latches for $p > 0$. Clearly, $d = 1$, so

$$T(\mathrm{C}) = 6(5 \cdot 2^p - 5) = 30(2^p - 1).$$

For Algorithm F (using Lemma 10),

$$\sum_{i=0}^{m-1} l_i = \sum_{j=1}^{p} 2^{j-1} = \sum_{j=0}^{p-1} 2^j = 2^p - 1,$$

$$\begin{aligned}\sum_{i=1}^{m-1} (e_i - n_i) &= \sum_{j=1}^{p-1} (5 \cdot 2^j - 5) - (3 \cdot 2^j - 2) \\ &= \sum_{j=1}^{p-1} 2^{j+1} - 3(p-1) = 2\sum_{j=1}^{p-1} 2^j - 3(p-1) \\ &= 4\sum_{j=0}^{p-2} 2^j - 3(p-1) = 4(2^{p-1} - 1) - 3(p-1) \\ &= 2^{p+1} - 1 - 3p,\end{aligned}$$

$$\begin{aligned}T(\mathrm{F}) &= 20(2^p - 1) + (3 \cdot 2^p - 2) - 4 + 3(2^{p+1} - 1 - 3p) - 2(2^p - 1) \\ &= 27 \cdot 2^p - 27 - 9p,\end{aligned}$$

$$r_p = \frac{27(2^p - 1) - 9p}{30(2^p - 1)} = O(1).$$

Note that $r_1 = \frac{3}{5}$, and r_p is asymptotic to $\frac{9}{10}$ for large p.

If we make one small change to the diamond graph, we get the "skewed diamond" graph in Fig. 7. The analysis of Algorithm F for this graph is *exactly the*

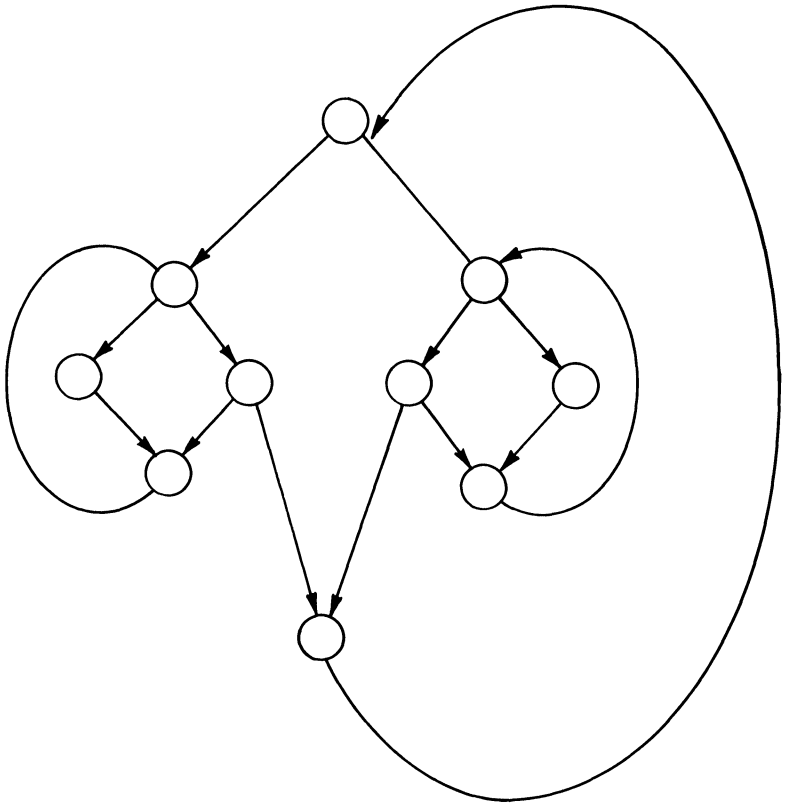


FIG. 7. *Skewed diamond graph, level* $p = 2$

same, but now $d = p$. Thus

$$T(\mathrm{C}) = 2(p + 2)(5 \cdot 2^p - 5) = (10p + 20)(2^p - 1).$$

The ratio becomes

$$r_p = \frac{27(2^p - 1) - 9p}{(10p + 20)(2^p - 1)},$$

which is $O(1/p)$! Thus Algorithm C may blow up badly while Algorithm F remains stable. But there is a major difference between this case and the seashell graph, where r_p was $O(p)$. For diamond graphs with $p \geqq 1$, r_p will always be less than 1. In fact $r_1 = \frac{3}{5}$, $r_2 = \frac{52}{120}$, and r_p is monotone decreasing.

4.6. Fan graph. The fan graph (see Fig. 8) is a form which can be produced only by unstructured programming using goto's, so it should be considered rare. The fan graph has $2^{p+1} - 1$ nodes, $2^{p+2} - 4$ edges, and 2^p latches for $p > 0$.

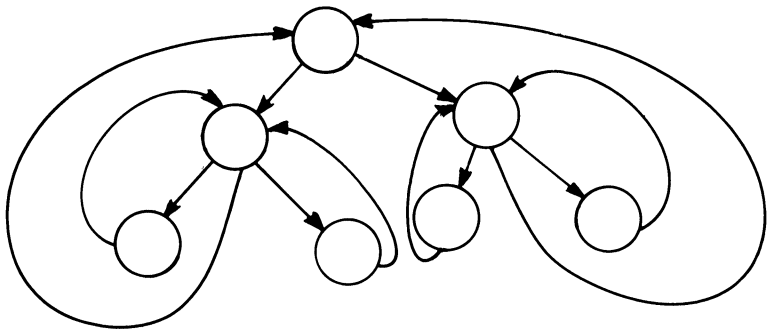


FIG. 8. *Fan graph of level* $p = 2$

Clearly, $d = p$, so

$$T(\mathrm{C}) = 2(p + 2)(2^{p+2} - 4) = (8p + 16)(2^p - 1).$$

For Algorithm F (using Lemma 10),

$$\sum_{i=0}^{m-1} l_i = \sum_{j=1}^{p} 2^j = 2\sum_{j=0}^{p-1} 2^j = 2(2^p - 1),$$

$$\sum_{i=1}^{m-1} (e_i - n_i) = \sum_{j=1}^{p-1} ((2^{j+2} - 4) - (2^{j+1} - 1))$$

$$= \sum_{j=1}^{p-1} (2^{j+1} - 3) = 4\sum_{j=0}^{p-2} 2^j - 3(p - 1)$$

$$= 4(2^{p-1} - 1) - 3p + 3 = 2^{p+1} - 1 - 3p,$$

$$T(\mathrm{F}) = 4(2^{p+2} - 4) + (2^{p+1} - 1) - 4$$

$$+ 3 \cdot (2^{p+1} - 1 - 3p) - 2(2^{p+1} - 2)$$

$$= 20 \cdot 2^p - 20 - 9p = 20(2^p - 1) - 9p.$$

The ratio is

$$r_p = \frac{20(2^p - 1) - 9p}{(8p + 16)(2^p - 1)} < \frac{5}{6} \quad \text{for } p \geqq 1.$$

Once again the ratio is $O(1/p)$ and always less than 1.

5. Summary and conclusions. We have derived an upper bound for the number of bit-vector operations required by the interval method (Algorithm F) for live analysis and compared this with the upper bound for the simple iterative method (Algorithm C) on several self-replicating graphs. The results (summarized in Table 1) are inconclusive since they show that r_p, the ratio of the number of steps required by Algorithm F to the number required by Algorithm C on a level p graph can be $O(p)$, $O(1)$, or $O(1/p)$ for graphs of different forms. However, in all examples, Algorithm F required fewer steps for graphs of level $p < 7$.

TABLE 1
Summary of comparisons on self-replicating families

Class of G_p	No. of nodes n	No. of edges e	Total no. of latches $\sum_{i=1}^{m-1} l_i$	d	$T(C)$	$T(F)$	$r_p = \frac{T(F)}{T(C)}$	Asymptotic behavior of r_p
Seashell	$p+1$	$2p$	p	1	$12p$	$\frac{3p+5}{2}p$	$\frac{3p+5}{24}$	$r_p = O(p)$
Spiral	$2p+1$	$4p$	p	p	$p(8p+16)$	$p(3p+10)$	$\frac{3p+10}{8p+16}$	$r_p \sim \frac{3}{8}$
Hecht–Ullman binary	2^p	$2^{p+1}-2$	2^p-1	1	$12(2^p-1)$	$10\cdot 2^p - 10 - 6p$	$\frac{10(2^p-1)-6p}{12(2^p-1)}$	$r_p \sim \frac{5}{6}$
Double-chain	$p+1$	$2p$	p	p	$p(4p+8)$	$\frac{3p+5}{2}p$	$\frac{3p+5}{8p+16}$	$r_p \sim \frac{3}{8}$
Diamond	$3\cdot 2^p-2$	$5\cdot 2^p-5$	2^p-1	1	$30(2^p-1)$	$27\cdot 2^p - 27 - 9p$	$\frac{27(2^p-1)-9p}{30(2^p-1)}$	$r_p \sim \frac{9}{10}$
Skewed diamond	$3\cdot 2^p-2$	$5\cdot 2^p-5$	2^p-1	p	$(10p+20)(2^p-1)$	$27\cdot 2^p - 27 - 9p$	$\frac{27(2^p-1)-9p}{(10p+20)(2^p-1)}$	$r_p = O\left(\frac{1}{p}\right)$
Fan	$2^{p+1}-1$	$2^{p+2}-4$	$2(2^p-1)$	p	$(8p+16)(2^p-1)$	$20(2^p-1)-9p$	$\frac{20(2^p-1)-9p}{(8p+16)(2^p-1)}$	$r_p = O\left(\frac{1}{p}\right)$

The reader is cautioned on three points. First, the estimates obtained herein are based only on bit-vector operations; other costs have not been considered. For example, the costs of interval bookkeeping in Algorithm F and bit-vector testing in Algorithm C have been ignored. These costs could substantially affect performance. Second, the estimate for Algorithm C is exact *only if* there is global data flow. If all the data flow is confined to local regions, Algorithm C will converge more rapidly. Third, Algorithm C works on all graphs while Algorithm F works only for reducible flow graphs.

Acknowledgments. John Cocke and Frances Allen of IBM contributed advice, suggestions, and encouragement during the development of Algorithm F. Barry Rosen of IBM Research read the manuscript originally and provided corrections and criticisms. Both referees made suggestions for significant improvements that have been incorporated into the final version of this paper. The author thanks all of these people for their time and effort.

REFERENCES

[1] F. E. ALLEN, *Control flow analysis*, SIGPLAN Notices, 5 (1970), pp. 1–19.

[2] ———, *A basis for program optimization*, Proc. IFIP Conf. 71, North-Holland, Amsterdam, 1971.

[3] F. E. ALLEN AND J. COCKE, *Graph theoretic constructs for program control flow analysis*, IBM Res. Rep. RC3923, T. J. Watson Res. Center, Yorktown Heights, N.Y., July 1972.

[4] J. COCKE AND J. T. SCHWARTZ, *Programming Languages and their Compilers*, New York University, New York, 1969.

[5] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation, and Compiling. Vol. II: Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[6] K. KENNEDY, *A global flow analysis algorithm*, Internat. J. Comput. Math., 3 (1971), pp. 5–15.

[7] M. S. HECHT AND J. D. ULLMAN, *Analysis of a simple algorithm for global data flow problems*, Proc. ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages, Boston, Mass., Oct. 1973.

[8] M. SCHAEFER, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[9] G. A. KILDALL, *A unified approach to global program optimization*, Proc. ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages, Boston, Mass., October 1973.

[10] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, Acta Informatica, 2 (1973), pp. 191–213.

[11] D. E. KNUTH, *An empirical study of FORTRAN programs*, Software—Practice and Experience, 1 (1971), pp. 105–134.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

MULTIDIMENSIONAL SEARCHING PROBLEMS*

DAVID DOBKIN AND RICHARD J. LIPTON†

Abstract. Classic binary search is extended to multidimensional search problems. This extension yields efficient algorithms for a number of tasks such as a secondary searching problem of Knuth, region location in planar graphs, and speech recognition.

Key words. binary search, secondary search, efficient algorithms, planar graphs, finite element methods

1. Introduction. One of the most basic operations performed on a computer is searching. A search is used to decide whether or not a given word is in a given collection of words. Since many searches are usually performed on a given collection, it is generally worthwhile to organize the collection into a more desirable form so that searching is efficient. The organization of the collection—called *preprocessing*—can be assumed to be done at no cost relative to the cost of the numerous searches.

One of the basic searching methods is the *binary searching method* (Knuth [1]). For the purposes of this paper, we can view binary search as follows:

Data: A collection of m points on a line.

Query: Given a point, does it equal any of the m points?

Binary search, since it organizes the points into a balanced binary tree, can answer this query in $\lfloor \log m \rfloor + 1$ "steps", where a step is a single comparison.[1] Note the preprocessing needed to form the balanced binary tree is a sort which requires $O(m \log m)$ steps. For the algorithms under consideration here, we will define a step in an algorithm as a comparison of two scalars or the determination of whether a point in 2-dimensional Euclidean space lies on, above or below a given line. For notational simplicity, we will define $g(m)$ as the number of steps necessary to perform a search through a set of m ordered objects: Thus, $g(m) = \lfloor \log m \rfloor + 1$.

This paper generalizes binary search to higher dimensions. Throughout, it is assumed that data can be organized in any manner desired at no cost. Thus our cost criterion for evaluating the relative efficiencies of searching algorithms will be the number of steps required to make a single query into the reorganized data.

The search problems considered are specified by a collection of data and a class of queries. These problems include:

1. *Data*: A set of m lines in the plane.
   Query: Given a point, does it lie on any line?
2. *Data*: A set of m regions in the plane.
   Query: Given a point, in which region does it lie?
3. *Data*: A set of m points in the plane.
   Query: Given a new point, to which of the original points is it closest?
4. *Data*: A set of m lines in n-dimensional space.
   Query: Given a point, does it lie on any line?

---

* Received by the editors October 4, 1974, and in revised form June 14, 1975.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. The work of the first author was supported in part by the National Science Foundation under Grant GJ-43157.

[1] Throughout this paper, all logarithms are taken to base 2.

5. *Data*: A set of m linear varieties of dimension[2] k in n-dimensional space.
   Query: Given a point, does it lie on any of the objects?
6. *Data*: A set of m hyperplanes (linear varieties of dimension $n-1$) in n-dimensional space.
   Query: Given a point, does it lie on any hyperplane?

These example problems form the basis for some important problems in diverse areas of computer science. Problems 1, 2 and 3 are fundamental to certain operations in computer graphics [2] and secondary searching. In particular, problem 3 is a reformulation of an important problem discussed by Knuth [1] concerning information retrieval. Problems 4, 5 and 6 are generalizations of the widely studied knapsack problem.

The main results of this paper are that fast algorithms exist for problems 1–6. In particular: problems 1–3 have $O(\log m)$ algorithms; problems 4–6 have $O(f(n)\log m)$ algorithms, where $f(n)$ is some function of the dimension of the space ($f(n)$ is determined more exactly later). The existence of these fast algorithms is somewhat surprising. For instance, lines in the plane (problem 1) are not ordered in any obvious way; hence, it is not at all clear how one can use binary search to obtain fast searches.

2. Basic algorithm in E^2. All of our fast algorithms are extensions of a fast algorithm for computing the predicate:

$$\exists 1 \leqq i \leqq m\ [(x, y) \text{ is on } L_i],$$

where $L_1, \cdots, L_m$ are lines and (x, y) is a point in a 2-dimensional Euclidean space (E^2). This predicate merely consists of querying whether a point in the plane lies on any of a given set of lines. Therefore, we begin with a proof that this predicate can be computed in $O(\log m)$ steps.

THEOREM 1. *For any set of lines $L_1, \cdots, L_m$ in the plane, there is an algorithm that computes $\exists 1 \leqq i \leqq m\ [(x, y)$ is on $L_i]$ in $3g(m)$ steps, given that the lines have been preconditioned.*

Proof. Let the intersections of the lines be given by the points $z_1, \cdots, z_n$ ($n \leqq m(m-1)/2$) and let the projections of these points onto the x-axis be given by $p_1, \cdots, p_n$. These points define a set of intervals $I_1, \cdots, I_{n+1}$ on the x-axis such that $I_1 = (-\infty, p_1)$, $I_i = (p_{i-1}, p_i)$, $i = 2, \cdots, n$, $I_{n+1} = (p_n, \infty)$, and within the slice of the plane defined by each of these intervals, no two of the original lines intersect. Thus we can define the relation $<_i$ $(1 \leqq i \leqq n+1)$ as follows:

$$L_j <_i L_k \quad \text{iff} \quad \forall x \in E^1[\text{if } p_i \leqq x \leqq p_{i+1}, \text{ then } L_j(x) \leqq L_k(x)].$$

(Note that $L(x)$ is equal to the value of y such that $(x, y) \in L$, and we set $p_0 = -\infty$, $p_{n+1} = \infty$.) By a simple continuity argument, it follows that each $<_i$ is a linear ordering on the lines $L_1, \cdots, L_m$. We can thus define for each i $(1 \leqq i \leqq n+1)$ a permutation $\pi(i, 1), \cdots, \pi(i, m)$ of $1, \cdots, m$ such that

$$L_{\pi(i,1)} <_i L_{\pi(i,2)} <_i \cdots <_i L_{\pi(i,m)}$$

[2] A linear variety of dimension k is the intersection of $n-k$ distinct hyperplanes having a nonempty intersection.

for $i = 1, \cdots, n+1$. An algorithm consisting of a binary search into a set of at most $m(m-1)/2+2$ objects (the points $\{p_i\}$) and a binary search into a set of m objects (the lines $L_{\pi(i,1)}, \cdots, L_{\pi(i,m)}$ for the proper choice of i) requires at most $g(m)+g(m(m-1)/2+2)$ steps, and since g is a monotonically increasing function with $g(m^2) \leqq 2g(m)$, this quantity is at most $3g(m)$. Degeneracies which may be introduced into the above algorithm by lines perpendicular to the x-axis may be removed by a rotation of the axes to a situation where no line is perpendicular to the new x-axis. □

Before studying applications of this algorithm to the problems mentioned above, it is worthwhile to examine its structure in more detail. What we have done is to find a method of applying an ordering to a set of lines in the plane. For a set of lines in the plane, no natural ordering exists, and thus it is reasonable to assume that any search algorithm which is "global" (i.e., considers the entire plane at once) must use a number of steps which grows linearly with the number of lines. The algorithm presented in Theorem 1 defines a set of regions of the plane in which the lines are ordered. In this sense, the algorithm is "local", and the two steps consist of finding the region in which to search and then doing a local search on an ordered set. The orderings are found during preprocessing of the data. The projections of intersection points (i.e., $\{p_i\}$) define the local regions into which the plane can be subdivided, and the permutations (i.e., $\{\pi(i, \cdot)\}$) define the orderings within each of the subdivisions of the plane. Moreover, it is clear that the algorithm not only determines whether the point lies on any line but also between which lines the point lies, if it does not lie on any line. Using this information, we can determine in which region of the plane determined by the given lines the point lies. Thus we have the following corollary.

COROLLARY. *Given a set of regions formed by m lines in the plane, we can determine in* $3g(m)$ *steps in which region a given point lies, given that the lines have been preconditioned.*

This algorithm forms the basis for what follows. We proceed by studying extensions of this algorithm to higher dimensions and applications of our basic algorithm and its extensions to some interesting problems of computer science.

3. Extensions to E^n. We have seen that searching in a set of m 0-dimensional objects in 1-dimensional space can be done in $g(m)$ steps and that searching in a set of m 1-dimensional objects in 2-dimensional space can be done in $3g(m)$ steps given that the original objects were preprocessed before any searches are undertaken. In the present section, we extend the search question to seek methods of searching in a set of m linear varieties of dimension k in n-dimensional space. In order to provide a clearer exposition, a series of lemmas will be presented, each of which can be viewed as a generalization of Theorem 1.

LEMMA 1. *For any set of lines* $L_1, \cdots, L_m$ *in n-dimensional Euclidean space* $(n \geqq 2)$, *there is an algorithm which computes* $\exists 1 \leqq i \leqq m\ [x \text{ is on } L_i]$ *for x a point in* E^n *in* $(n+1)g(m)$ *steps, given that the lines have been preconditioned.*

Proof. The proof is by induction on n and follows from Theorem 1 for $n = 2$. Now, suppose that $n > 2$. It is possible to find a hyperplane H such that none of the lines is perpendicular to H and such that no two of the lines project onto the same line in H. Projecting the lines onto H yields a set of lines $L'_1, \cdots, L'_m$ on H, and

projecting x onto H yields a point x' on H. Furthermore, if x lies on L_i, then x' lies on L_i'. By the induction hypothesis, we can determine on which lines of the set $\{L_1', \cdots, L_m'\}$ x' lies, in $ng(m)$ steps. If x' doesn't lie on any L_i', then x doesn't lie on any L_i. And if x' lies on $\{L_{i_1}', \cdots, L_{i_k}'\}$, the lines $L_{i_1}, \cdots, L_{i_k}$ are linearly ordered at x' with respect to the projected coordinate, and with a logarithmic search we can determine if x lies on any of $\{L_{i_1}, \cdots, L_{i_k}\}$. Since $i_k \leqq m$, this search requires at most $g(m)$ steps, and m lines in E^n can be searched in $(n+1)g(m)$ steps. □

LEMMA 2. *For any set of hyperplanes $H_1, \cdots, H_m$ in E^n $(n \geqq 2)$, there is an algorithm which determines, for any point x, whether x is on any hyperplane or which hyperplanes it is between in at most $(3 \cdot 2^{n-2} + (n-2))g(m)$ steps, given that the hyperplanes have been preconditioned.*

Proof. Let $h(m, n)$ be the time required to do the search. From Theorem 1, we know that $h(m, 2) \leqq 3g(m)$, and we will show here that $h(m, n) \leqq h(m^2, n-1) + g(m)$. Let K be a hyperplane which is not identical to any of the original hyperplanes. Then we proceed by forming the set of $(n-2)$-dimensional objects $J_1, \cdots, J_k$ formed as intersections of pairs of the hyperplanes we considered. Thus, for example, $J_1 = H_1 \cap H_2, \cdots, J_k = H_{m-1} \cap H_m$ and $k \leqq m(m-1)/2 < m^2$. From these hyperplanes, we form their projections $J_1', \cdots, J_k'$ onto K. If the point x projects onto x', we can by the induction hypothesis determine in less than $h(m^2, n-1)$ steps in which region of $(n-1)$-dimensional space x' lies. With respect to each of these regions, the hyperplanes $H_1, \cdots, H_m$ are ordered and can be searched in $g(m)$ steps. Thus if x' doesn't lie on any J_k', the lemma holds. And if x' lies on a hyperplane J_i', a search requiring less than $g(m)$ steps will determine in which region of E^n the point x lies. This proves that $h(m, n) \leqq h(m^2, n-1) + g(m)$. Solving this recursion yields $h(m, n) \leqq h(m^{2^k}, n-k) + kg(m)$ or

$$h(m, n) \leqq h(m^{2^{n-2}}, 2) + (n-2)g(m) = (3 \cdot 2^{n-2} + (n-2))g(m). \qquad \square$$

Combining the results and proof techniques of Lemmas 1 and 2 yields the following general theorem on searching k-dimensional objects in E^n.

THEOREM 2. *For any set $\theta_1, \cdots, \theta_m$ of linear varieties of dimension k in E^n, there is an algorithm that computes $\exists 1 \leqq i \leqq m[x \text{ is on } \theta_i]$ in $(3 \cdot 2^{k-1} + (n-2))g(m)$ steps for any point x in E^n, given that the θ_i have been preconditioned.*

Proof. Let $f(m, k, n)$ be the number of steps required by the search. Then if $k < n-1$, we can, by an argument similar to that used in the proof of Lemma 1, project the varieties onto a hyperplane in E^n, and proceeding as there, we find it is clear that $f(m, k, n) \leqq f(m, k, n-1) + g(m)$ if $k < n-1$. Continuing this induction yields $f(m, k, n) \leqq f(m, k, k+1) + (n-k-1)g(m)$. Combining this result with the result of Lemma 2 that $f(m, k, k+1) \leqq (3 \cdot 2^{k-1} + (k-1))g(m)$ yields the result $f(m, k, n) \leqq (3 \cdot 2^{k-1} + (n-2))g(m)$, as in the statement of the theorem, where we make use of the identity $h(m, n) = f(m, n-1, n)$.

4. Applications in E^2. Before presenting applications, the basic algorithm must be (i) examined with respect to preprocessing, (ii) examined with respect to storage requirements, (iii) also extended to a slightly more general case.

Instead of m lines, assume that we are given m lines or line segments. The problem is then to search the regions formed by these generalized lines. It is easy

to see that the basic algorithm can be adapted here and that it operates in time $3g(m)$. Let N be the number of intersection points formed by the m lines. The preprocessing is:

1. Find the N intersection points formed by the m lines.
2. Store these intersection points after they are projected onto the x-axis.
3. For each two adjacent intersection points t_1 and t_2, find the permutation of the m lines in the region $t_1 \leqq x \leqq t_2$.

Step 1 takes $O(m^2)$ since finding the intersection of two lines consists merely of finding the solution of a simple linear system of equations. Step 2 is a sort of N objects; hence, it takes $O(N \log N)$ time. Finally, step 3 takes at most $O(m \log m)$ for each of the N regions: to determine the order of the m lines takes at most a sort of m objects. In summary, preprocessing takes $O(mN \log m) + O(N \log N)$, which is $O(m^3 \log m)$ since N is $O(m^2)$ in the worst case. The storage requirements are easily seen to be: $O(N)$ from step 1 and $O(mN)$ from step 3. Thus the total storage needed is seen to be $O(Nm)$.

We will now study two applications of the basic algorithm.

4.1. Planar graph search. Suppose that we are given a planar graph G with m edges. How fast can we determine which region of G a new point is in? For example, this location problem is central to the finite element method [3].

THEOREM 3. *There is an algorithm which can in time $O(\log m)$ and space $O(m^2)$ determine in which region of a planar graph with m edges a point lies, if the graph has been preconditioned.*

Proof. By an application of Euler's relation [4] the m line segments of G can only intersect in $O(m)$ points. Therefore, the basic algorithm—as modified—shows that planar graphs can be searched in $O(\log m)$ time. The preprocessing required is $O(m^2 \log m)$; the storage required is $O(m^2)$. □

4.2. Post office problem. The post office problem is a search problem for which Knuth [1] states there is no known efficient solution. Suppose that we are given m cities or "post offices". How fast can we determine which post office is nearest to a new point? This is the post office problem. We will now show how to

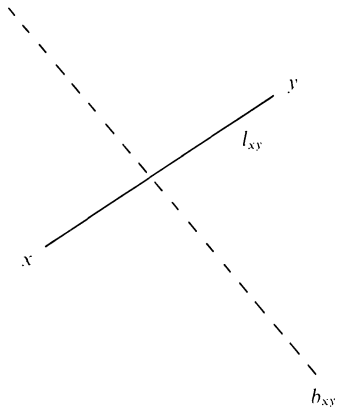


FIG. 1. *b_{xy} is the perpendicular bisector of l_{xy}. Therefore points below b_{xy} are closer to x than y, and points above b_{xy} are closer to y than x.*

reduce it to the planar search problem of 4.1: between each pair of post offices x and y construct the line segment l_{xy}. Then construct the perpendicular bisector of l_{xy}; call it b_{xy} (see Fig. 1). The line b_{xy} divides the plane into two regions. The points in the half-plane containing x are nearer to x than y; the points in the other half-plane are all nearer to y than x.

In order to solve the post office problem it is sufficient to determine, for a given point, which region of the regions formed by the $\binom{m}{2}$ lines b_{xy} it lies in. By the basic algorithm, this can be done in $3g(\binom{m}{2}) = O(\log m)$ time with $O(m^4)$ storage.

These applications of our basic algorithm are clearly optimal with respect to time to within a constant. (This follows since it takes at least $g(m)$ time to search m objects.) They, however, also demonstrate that our algorithms tend to use a large amount of storage. An interesting open question is therefore: can one search a planar graph's m regions in time $O(\log m)$ with storage $O(m)$? Or even storage $O(m \log m)$?

5. Applications in E^n. Most of the applications of the above algorithms in E^n are straightforward extensions of the applications given in the previous section. However, because of the exponential term in the operation count of Theorem 2, these extensions are only of interest if k, the dimension of the objects to be searched, is small relative to m, the number of objects to be searched. Typically, we would require that m is larger than 2^k and hopefully as large as 2^{2^k}. In cases where k and m do satisfy these criteria, speed-ups do occur by applying the algorithms of § 3.

Consider the problem of finding the closest points in spaces of small dimension. An example of such a problem occurs in the area of speech recognition. Sounds can be classified according to a set of less than 8 characteristics [5], and thus we may consider a data base for a speech recognition system to consist of a set of points in E^8. When such a system is used to understand a speaker, the method used is to find for each sound uttered the closest sound in the data base. In order to develop a real time speech recognition system, it is reasonable to allow large quantities of preprocessing and storage to be arranged in advance as a trade-off so that each sound uttered by the speaker can be identified as rapidly as possible. Thus for a set of m sounds to be in the data base, the speed up of $O(m/g(m))$ afforded by an extension of the closest-point algorithm of the previous section to E^8 is very useful. Further studies of this extension are necessary to yield improvements in the storage requirements.

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[2] W. NEWMAN AND R. SPROULL, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1973.

[3] J. A. GEORGE, *A computer implementation of the finite element method*, Ph.D. thesis, Stanford Univ., Stanford, Calif., 1971.

[4] C. L. LIU, *Introduction to Combinatorial Mathematics*, Addison-Wesley, Reading, Mass., 1968.

[5] S. LEVINSON, Private communication.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

ON THE ADDITIVE COMPLEXITY OF MATRIX MULTIPLICATION*

ROBERT L. PROBERT†

Abstract. A graph-theoretic model is introduced for bilinear algorithms. This facilitates in particular the investigation of the additive complexity of matrix multiplication. The number of additions/subtractions required for each of the problems defined by symmetric permutations on the dimensions of the matrices are shown to differ conversely as the size of each product matrix. It is noted that this result holds for any system of dual problems, not only dual matrix multiplication problems. This additive symmetry is employed to obtain various results, including the fact that 15 additive operations are necessary and sufficient to multiply two 2×2 matrices by a bilinear algorithm using at most 7 multiplication operations.

Key words. matrix multiplication, additive complexity, bilinear algorithms, symmetric computations, analysis of algorithms, additions, computational complexity, duality

1. Introduction. Recently, considerable attention has been focused upon the need for obtaining lower bounds on the arithmetic complexity of computations of certain simple, frequently encountered functions. Investigations into the essential or theoretical complexity of matrix computations such as matrix multiplication have been carried out by Dobkin [1], Fiduccia [2], [4], Hopcroft and Kerr [6], Hopcroft and Musinski [7], Strassen [16], [17], Winograd [18], [19], this author [12] and others. In each case, however, the measure of complexity which is used is the number of multiplications required by any algorithm which performs the computation. The minimum number of additions/subtractions required, or the theoretical additive complexity of matrix multiplication and related problems has been investigated to some extent in [4], [8], [10], [13] and [20], though certainly not as extensively as multiplicative complexity.

The purpose of this paper is mainly to present a technique for investigating the additive complexity of matrix multiplication problems. We feel that research into additive complexity may be a useful approach for characterizing the overall arithmetic complexity of matrix computations. As well, we attempt to provide a general framework for studying other classes of dual problems.

In [12], we proved formally that to investigate the multiplicative complexity of algorithms which do not exploit commutativity of multiplication, it sufficed to analyze the multiplicative complexity of bilinear chains. Whether a similar (probably linear) reduction is possible for additive complexity is an open problem.[1] In this paper we will consider only bilinear algorithms for computing sets of bilinear forms, i.e., algorithms whose nonscalar multiplication steps may be performed in only one order. This class of algorithms is properly contained in the

---

* Received by the editors October 4, 1974, and in revised form June 26, 1975.

† Computational Science Department, University of Saskatchewan, Saskatoon, Saskatchewan, Canada S7N OWO. This work was supported by the National Research Council of Canada under Grant A8982.

[1] Since this paper was first submitted, the author has shown [14] that total arithmetic complexity is increased at most by a linear factor in considering computations only by algorithms which are bilinear chains rather than computations by the commutativity-exploiting algorithms in class *W*. A. Borodin and I. Munro will include an almost equivalent result in a forthcoming book.

class NC [12] of algorithms which do not exploit commutativity of multiplication. In turn, NC is a proper subclass of the class W of algorithms defined by Winograd [19] which may exploit multiplicative commutativity. If arithmetic complexities over these classes are related by a linear factor, then lower bounds on the arithmetic complexity of a function over the class of bilinear chains will be linear multiples of lower bounds on the complexity over the class W of algorithms.[2]

We begin by reviewing several basic definitions, then introducing a straightforward graph model for bilinear computations, the addition flow representation. Next, we show that addition flow representations are natural models for studying the additive complexity of matrix multiplication problems. After a brief review of the multiplicative symmetry theorem [1], [7], [12], we present an analogous theorem on additive duality of matrix multiplication problems. As an application of this additive symmetry, we show that fast bilinear algorithms for multiplying 2×2 matrices (algorithms which use fewer than 8 multiplication steps) must use no fewer than 15 additions/subtractions. For the sake of comparison, Strassen's algorithm uses 18 such operations. To show that this lower bound is achievable, we include a fast algorithm due to S. Winograd, which does not assume commutativity, and which meets the lower bound of 15 additive operations.

The additive complexity of inner product and matrix-vector product are immediate corollaries of additive symmetry. Finally, we conclude with a generalization of additive duality to systems of dual problems, and with suggestions for further applications of addition flow representations.

2. Basic definitions. We refer to the computation of the product of two matrices $A_{m\times n}B_{n\times p} = Y_{m\times p}$ with elements belonging to a ring (with unit) R as the computation of an (m, n, p) product, where the left-hand matrix is $(m \times n)$, and the right-hand matrix is an $(n \times p)$ matrix. Then the *multiplicative complexity*, $\mathcal{M}(m, n, p)$ of computing (m, n, p) products is the smallest number of multiplications of the form $\mathcal{L}_K\{a_{ij}\} \cdot \mathcal{L}_K\{b_{ij}\}$ which can be used to compute $AB = Y$. A, B, Y and matrix variables, not particular matrices. $\mathcal{L}_K\{x_{ij}\}$ denotes a K-linear form in the elements x_{ij}, where K is a subring of the center of R. Usually $K = \{0, 1, -1\}$.

We denote the not necessarily optimal set of t multiplications used by an algorithm for (m, n, p) products by $M = \{M_1, \cdots, M_t\}$. Then the *additive complexity* (minimum possible number of addition/subtraction operations) of computing (m, n, p) products by means of the fixed set M of t multiplications is denoted $\mathcal{A}(m, n, p, M)$. The smallest possible number of addition/subtraction steps which can form and combine any set of exactly t multiplications to compute (m, n, p) products is denoted $\mathcal{A}(m, n, p, t)$.

For example, (m, n, p) products are computed by the classical algorithm in mnp multiplication steps and $mp(n-1)$ additions/subtractions. Thus $\mathcal{M}(m, n, p) \leqq mnp$ and $\mathcal{A}(m, n, p, t) \leqq mp(n-1)$, where $t = mnp$.

A K-bilinear algorithm (see also [12]) for computing the entries of the product of matrices A and B (denoted $E(AB)$) from $E(A) \cup E(B)$ is a finite sequence $f_1, f_2, \cdots$ such that

[2] By the result mentioned in the previous footnote, this is the case.

(i) for each k, $f_k \in E(A) \cup E(B)$,
or $f_k = f_i$ "op" f_j for $i, j < k$ and "op" $\in \{'+', '-', '\times'\}$,
or $f_k = r \times f_i$, where $r \in K$,

(ii) whenever $f_k = f_i \times f_j$, f_i and f_j must be K-linear forms in elements of A and B, respectively, and

(iii) for each element $z \in E(AB)$, there is at least one k such that $f_k = z$.

In other words, we can think of a computation of $A \cdot B$ according to a bilinear algorithm as consisting of four disjoint stages. Suppose a bilinear algorithm used t multiplication steps $M_1, M_2, \cdots, M_t$. The corresponding computation would proceed as follows:

1. form left-hand factors (linear forms in A) for the t multiplications,
2. form right-hand factors (linear forms in B) for the t multiplications,
3. compute the t values $\{M_1, M_2, \cdots, M_t\}$,
4. form the product elements from the M_i values.

This definition can be generalized in a natural way to n-linear algorithms for computing n-linear forms. Unless otherwise specified, the term algorithm will mean bilinear algorithm in this paper. We remind the reader that this type of algorithm cannot exploit the commutative law for multiplication.

In order to facilitate analysis, we can define a natural correspondence between matrix products and matrix-vector products as in [3] and [13].

The *tensor product* $A_{m\times n} \otimes B_{r\times s}$ is $C_{mr\times ns}$, where $c_{i+k,j+p} = a_{ij}b_{k+1,p+1}$, $0 \leqq k \leqq r-1$, $0 \leqq p \leqq s-1$, for all i and j. If $B_{r\times s}$ has the s columns $b_1, \cdots, b_s$ define $\kappa(B) = [b_1^T, b_2^T, \cdots, b_s^T]^T$, i.e., an rs-column vector. Then, any (m, n, p) product AB can be encoded as a matrix-vector product Xy where $X = I_p \times A$, $y = \kappa(B)$, i.e., $E(AB) = E(Xy)$. Again, this method can be iteratively applied to translate the product of n matrices into a sequence of $n-1$ matrix-vector products. Now we can employ the following decomposition theorem.

THEOREM 1 [3]. *One can compute Xy (matrix-vector products) with t multiplications by a K-bilinear algorithm if and only if $X - D = CUB$, where B, C, D are fixed matrices with elements in K and U is a $t \times t$ diagonal matrix with entries in $\mathscr{L}_K\{x_{ij}\}$.*

Thus searching for faster bilinear algorithms for (m, n, p) products is equivalent to encoding the problem as a matrix-vector product Xy and then decomposing X into a product of 3 simple matrices C, U, B, where the dimension of U is minimized. This approach was suggested in [3] and employed in [12] for studying the multiplicative complexity of matrix multiplication.

We present a natural graph-theoretic model of computation which lends itself to the investigation of lower bounds on additive complexity as well. Essentially, the model consists of an ordered triple of three graphs $\langle G_1, G_2, G_3\rangle$ such that G_1 encodes the addition/subtraction steps used to form left-hand factors of the fixed set of multiplication steps, G_2 encodes the addition/subtraction steps used to form the corresponding right-hand factors, and G_3 represents exactly how to combine the results of the multiplications to form the elements in the product matrix. In other words, given a (bilinear) algorithm α which computes (m, n, p) products using the t multiplications $M = \{M_1, M_2, \cdots, M_t\}$, an *addition flow representation* of α, denoted F_α, is an ordered triple $\langle G_1, G_2, G_3\rangle$, where each G_i is an acyclic, multisource, multisink flow graph with vertex and edge sets denoted

VG_i, ΓG_i, respectively. As well, if the source and sink set of each G_i is denoted R_i, S_i, respectively, then

1. $|R_1| = mn$, $|R_2| = np$, $|S_3| = mp$,
2. $|S_1| = |S_2| = |R_3| = t$,
3. there exists a labeling morphism ℓ which maps each intermediate result in the bilinear algorithm α into a possibly, but not necessarily, unique vertex in exactly one of VG_1, VG_2, VG_3. (A complete, formal definition is given in [13].)

As an example, consider a representation for α_S, Strassen's algorithm for computing $A_{2\times 2}B_{2\times 2} = Y_{2\times 2}$ [15]. The 7 multiplications $M_S = \{M_1, \cdots, M_7\}$ used by α_S are:

$$M_1(a_{11}+a_{22})(b_{11}+b_{22}), \qquad M_4 a_{22}(-b_{11}+b_{21}),$$
$$M_2(a_{21}+a_{22})b_{11}, \qquad M_5(a_{11}+a_{12})b_{22},$$
$$M_3 a_{11}(b_{12}-b_{22}), \qquad M_6(-a_{11}+a_{21})(b_{11}+b_{12}),$$
$$M_7(a_{12}-a_{22})(b_{21}+b_{22}).$$

Then α_S will have flow representation $F_S = \langle G_1, G_2, G_3\rangle$, where (depending on the addition/subtraction steps in α_S) G_1, G_2 and G_3 might be as shown in Figs. 1, 2 and 3, respectively.

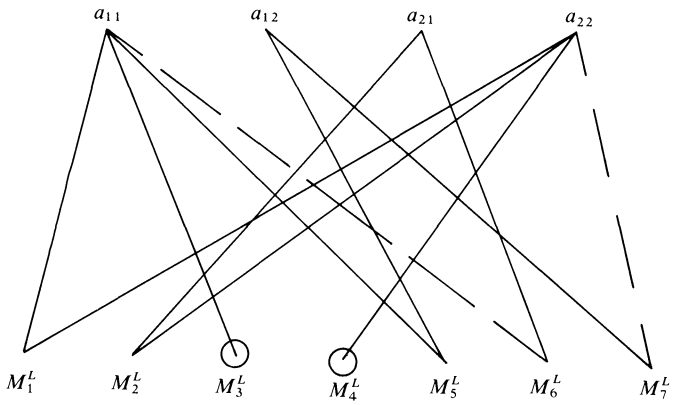


FIG. 1. *Graph G_1 of flow representation F_S*

To simplify diagrams, we adopt the conventions that all edges are directed downwards, solid edges have weight 1, dashed edges have weight -1. For example, M_1^R, $(b_{11}+b_{22})$, according to G_2 is computed as $f_1 = b_{11}+b_{12}$,

$$f_2 = b_{12}-b_{22}, \qquad M_1^R = f_3 = f_1 - f_2.$$

The circled sink vertices represent "placeholders" in a computation of α_S, representing a step of the form $f_i = 1 \times f_j$. Such steps are clearly not counted in evaluating the arithmetic complexity of the computation, but the representing

vertices provide us a uniform encoding for all final values for each stage of the computation.

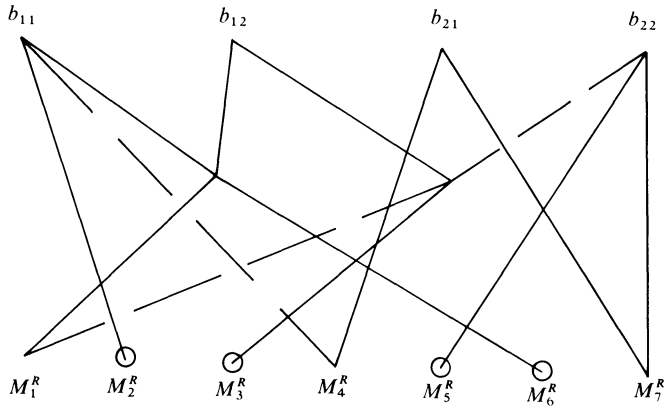


FIG. 2 *Graph G_2 of F_S*

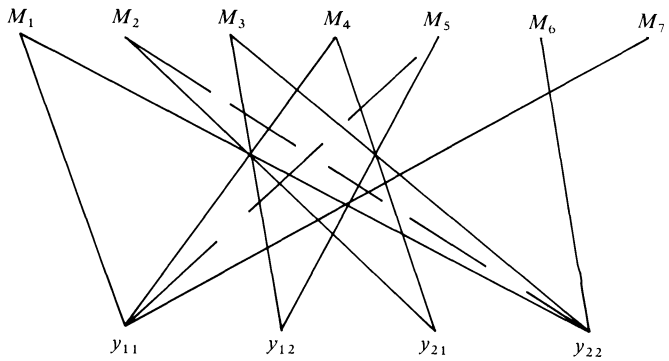


FIG. 3 *Graph G_3 of F_S*

The reader should verify that G_3 in Fig. 3 specifies the computation of the product elements from the results of the seven multiplication steps.

3. Properties of graph representations. In [13], we show the following.

LEMMA 2. *For any (bilinear) α which computes (m, n, p) products in t multiplications and* **a** *additions/subtractions, there is a (bilinear) algorithm α' which performs the same computation in t multiplications and no more than* **a** *addition/subtraction steps, and α' has an addition flow representation.*

Thus this model adequately represents the class of bilinear algorithms with respect to arithmetic complexity; hence we will consider only algorithms which have flow representations.

We can show as well, that any computation with the final results of each stage specified has a unique (up to graph isomorphism) representation. However, the same representation can represent many algorithms depending on the method of

traversing, since each computation involves a path from source vertices. We define the *standard algorithm with representation* $F=\langle G_1, G_2, G_3\rangle$ to be the algorithm which follows the four natural stages of a bilinear computation in order. For example, the multiplications represented by R_3 are calculated before G_3 is processed.

Thus, instead of studying the class of (bilinear) algorithms which compute (m, n, p) products, we investigate without loss of generality the class of flow representations.

The number of additions/subtractions represented at a particular nonsource vertex v in some G_i is clearly indegree$(v)-1$. Thus we have the next lemma.

LEMMA 3. *The number of additions/subtractions used by algorithm* α *with addition flow representation* $F_\alpha=\langle G_1, G_2, G_3\rangle$, *denoted* $C^+(F_\alpha)$, *is* $\sum_{i=1}^{3}[|\Gamma G_i|-|VG_i|+|R_i|]$.

For example, the number of addition/subtraction steps represented by $F_S=\langle G_1, G_2, G_3\rangle$ is $(12-11+4)+(14-13+4)+(12-11+7)=18$, as expected.

The sizes of the source and sink sets in a flow representation exactly specify the parameters of the matrix multiplication problem. For example, given any addition flow representation $F=\langle G_1, G_2, G_3\rangle$, the standard algorithm with the representation computes (m, n, p) products using t multiplication steps, where

$$t=|R_3|, \quad n=(|R_1|\cdot|R_2|/|S_3|)^{1/2}, \quad m=|R_1|/n, \quad p=|R_2|/n.$$

Thus it is not surprising to find that operations on addition flow representations produce representations for matrix multiplication with modified parameters.

We now define two such operations, rotation and reflection (or dual).

Given an addition flow representation $F=\langle G_1, G_2, G_3\rangle$, the *rotation* of F, denoted F^R, is an ordered triple of graphs $\langle G_3^{DT}, G_1, G_2^{DT}\rangle$. For any G_i, G_i^D is the directional dual (edge reversal) of G_i with relabeling of the sink and source sets appropriate to the position of G_i^D within the new ordered triple (we will later show that rotation and reflection yield new addition flow representations). G_i^T is the transposition of graph G_i, where, for example, each occurrence of a_{ij}, b_{ij} or y_{ij} (whichever is appropriate to the position of G_i^T in the new triple) in a vertex name is replaced by a_{ji}, b_{ji}, or y_{ji}, respectively. G_i^{DT} denotes the graph produced by taking the transposition of the directional dual of G_i. Clearly, $G_i^{DT}=G_i^{TD}$.

The *reflection* (*dual*) of an addition flow representation $F=\langle G_1, G_2, G_3\rangle$, denoted F^D, is defined as the ordered triple of graphs $\langle G_1^T, G_3^D, G_2^D\rangle$.

At this point, we should verify that the matrix multiplication parameters corresponding to F^R and F^D are permutations formed by a rotation and reflection, respectively, of the parameters associated with F.

LEMMA 4. *Let* $F=\langle G_1, G_2, G_3\rangle$ *be an addition flow representation for computing* (m, n, p) *products in* t (*nonscalar*) *multiplication steps. Then* F^R *represents a method of computing* (p, m, n) *products in* t *multiplications, i.e., a problem with parameters which are a rotation of the original problem's parameters. Similarly,* F^D *is an addition flow representation for computing* (n, m, p) *products, a reflection of the original* (m, n, p) *product problem.*

Proof. By definition, if $F^R=\langle \hat{G}_1, \hat{G}_2, \hat{G}_3\rangle$ represents a matrix multiplication algorithm (which it does, as shown later), then the algorithm computes (u, v, w)

products, where

$$v = \left(\frac{|\hat{R}_1| \cdot |\hat{R}_2|}{|\hat{S}_3|}\right)^{1/2} \quad \text{(where } \hat{R}_i, \hat{S}_i \text{ are the source, and sink set, respectively, for graph } \hat{G}_i\text{)}$$

$$= \left(\frac{|S_3| \cdot |R_1|}{|R_2|}\right)^{1/2} = \left(\frac{mp \cdot mn}{np}\right)^{1/2} = m.$$

Then, $u = |\hat{R}_1|/m = mp/m = p$, and $w = |\hat{R}_2|/m = mn/m = n$.

Thus $(u, v, w) = (p, m, n)$, which is a rotation permutation of the original parameter triple (m, n, p).

Similarly, let $F^D = \langle \tilde{G}_1, \tilde{G}_2, \tilde{G}_3 \rangle$ represent an algorithm to compute (x, y, z) products. Again, by the definition of addition flow representations,

$$y = \left(\frac{|\tilde{R}_1| \cdot |\tilde{R}_2|}{|\tilde{S}_3|}\right)^{1/2} = \left(\frac{|R_1| \cdot |S_3|}{|R_2|}\right)^{1/2} = \left(\frac{mn \cdot mp}{np}\right)^{1/2} = m.$$

Then $x = |\tilde{R}_1|/m = mn/m = n$, and $z = |\tilde{R}_2|/m = mp/m = p$.

Thus $(x, y, z) = (n, m, p)$, a reflection permutation of the original parameter triple (m, n, p). The lemma follows.

Geometrically, we could represent these transformations R and D as rotations and reflections on triangles (see Fig. 4). In other words, these operators on addition flow representations form a dihedral group of transformations on an ordered list of problem parameters.

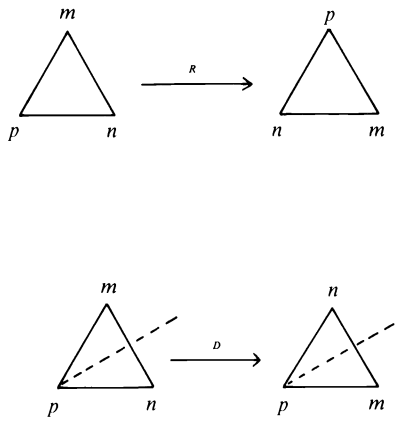


FIG. 4. *Geometric representation of transformations R and D*

To show the standard algorithms corresponding to F^R and F^D are the appropriate matrix multiplication algorithms, we first note that, as expected, matrices representing the contribution of source vertices to the terms represented by sink vertices in F_α provide a characterization of the algorithm α.

Given an addition flow representation $F_\alpha = \langle G_1, G_2, G_3 \rangle$, for each component graph G_i of F, we define a *connection matrix* C^i with dimension $|S_i| \times |R_i|$ as follows:

$$c^i_{jk} \quad \text{has value} \quad \sum_{p \in P} \prod_{\delta \in p} w_\delta,$$

where P is the set of all paths p from $r_k \in R_i$ to s_j in S_i, p is the set of edges in a particular path, and w_δ is the scalar weight assigned to edge δ (if α is K-bilinear, then $w_\delta \in K$). If P is empty, c^i_{jk} is set to zero. Thus, c^i_{jk} indicates the contribution of the value represented by r_k in R_i to the linear form represented by s_j in S_i.

Now in [13], we showed the next lemma.

LEMMA 5. *Suppose α is a bilinear algorithm for computing (m, n, p) products. Let $F_\alpha = \langle G_1, G_2, G_3\rangle$ be the addition flow representation for α. Then*

$$E(A_{m\times n}B_{n\times p}) = E(Xy), \quad \text{where } y = \kappa(B) \text{ and } X = I_p \otimes A = C^3 V C^2,$$

where the source and sink vertices representing matrix elements follow the κ(column-major) ordering, and V is a $t \times t$ diagonal matrix such that

$$v_{ii} = M_i = (c^1_{i1}, c^1_{i2}, \cdots, c^1_{i,mn})\kappa(A)$$

and

$$v_{ij} = 0 \quad \text{for } i \neq j.$$

Proof. The result follows from the definition of connection matrices and the bilinearity of α.

THEOREM 6. *If α computes (m, n, p) products and has representation $F = \langle G_1, G_2, G_3\rangle$, then the standard algorithm $\hat{\alpha}$ with representation $\hat{F} = F^D = \langle G_1^T, G_3^D, G_2^D\rangle$ computes (n, m, p) products.*

Proof. The source set $\hat{R}_2$ in the second graph of $\hat{F}$ has the property that $|\hat{R}_2| = |S_3| = mp$. Assume without loss of generality that the mp elements represented by $\hat{R}_2$ belong to an $m \times p$ matrix U, i.e., that $\hat{R}_2$ represents $\kappa(U)$. $\hat{R}_1$ may be thought of as representing A^T or an $n \times m$ matrix W (since A, B are matrix variables, not particular matrices).

Therefore $\hat{\alpha}$ computes $(C^2)^T \hat{V} (C^3)^T \kappa(U)$, where $\hat{V}$ is V with each occurrence of a_{ij} in V corresponding exactly to an occurrence of a_{ji} (or w_{ji}) in $\hat{V}$. Thus $\hat{v}_{ij} = 0$ for $i \neq j$ and $\hat{v}_{ii} = (c^1_{i1}, c^1_{i2}, \cdots, c^1_{i,mn})\kappa(A^T)$ or $(c^1_{i1}, \cdots, c^1_{i,mn})\kappa(W)$. But $(C^2)^T \hat{V} (C^3)^T \kappa(U)$ is $X^T z$, where $z = \kappa(U)$. Since $E(Xy) = E(A_{m\times n}B_{n\times p})$ is computed by $\alpha,\hat{\alpha}$ computes $E(X^T z) = E(W_{n\times m}U_{m\times p})$ or (n, m, p) products as required.

COROLLARY 7. *If α with addition flow representation F computes (m, n, p) products using no more than t multiplication steps and $\mathbf{a}$ additive steps, then α', the standard algorithm for F^D, computes (n, m, p) products using t multiplication steps and $\mathbf{a} + (m - n)p$ additive operations (scalar multiplications are not counted). More concisely, $C^+(F^D) - C^+(F) = (m - n)p$.*

The simplest proof of the analogous result for F^R seems to depend on the following observation from [13].

LEMMA 8. *If α with addition flow representation $F = \langle G_1, G_2, G_3\rangle$ computes (m, n, p) products, then the standard algorithm with flow representation $\langle G_2^T, G_1^T, G_3^T\rangle$ computes (p, n, m) products using no more multiplications or additive operations than α.*

By successive applications of Theorem 6 and Lemma 8, we obtain Theorem 9 below.

THEOREM 9. *If α with addition flow representation $F = \langle G_1, G_2, G_3\rangle$ computes (m, n, p) products, then α', the standard algorithm with representation $F^R = \langle G_3^{DT}, G_1, G_2^{DT}\rangle$, computes (p, m, n) products.*

Proof. By Theorem 6, the standard algorithm with representation

$$F^D = \langle G_1^T, G_3^D, G_2^D \rangle = \langle H_1, H_2, H_3 \rangle,$$

say, computes (n, m, p) products.

By Lemma 8, the standard algorithm with representation $\langle H_2^T, H_1^T, H_3^T \rangle$ computes (p, m, n) products.

But

$$\begin{aligned} \langle H_2^T, H_1^T, H_3^T \rangle &= \langle (G_3^D)^T, (G_1^T)^T, (G_2^D)^T \rangle \\ &= \langle G_3^{DT}, G_1, G_2^{DT} \rangle = F^R, \end{aligned}$$

as required. This theorem directly yields an analogous result to Corollary 7.

COROLLARY 10. *If α has addition flow representation F and computes (m, n, p) products in t multiplication steps and $\mathbf{a}$ additive operations then α', the standard algorithm with addition flow representation F^R, computes (p, m, n) products in t multiplication steps and $\mathbf{a}+(m-n)p$ additive operations. Using our notation, $C^+(F^R) - C^+(F) = (m-n)p$.*

As a minor note, we observe that by alternately applying the operations of reflection and rotation to the addition flow representation of a given algorithm for computing (m, n, p) products, we obtain algorithms of no greater multiplicative complexity for computing the five symmetric matrix products, i.e., (u, v, w) products where (u, v, w) is a dihedral permutation of (m, n, p). This result, which we call the multiplicative symmetry theorem, was discovered and proved independently by Dobkin and Brockett [1], Hopcroft and Musinski [7], this author [12], and Strassen [17].

THEOREM 11 (multiplicative symmetry). *$\mathcal{M}(m, n, p) = \mathcal{M}(\sigma(m, n, p))$, where $\sigma(m, n, p)$ yields any symmetric permutation of (m, n, p). Also, given an algorithm which computes (m, n, p) products using t multiplications, we can derive an algorithm to compute any symmetric product using t multiplication steps.*

By these properties of addition flow representations, we have the main additive symmetry result. The number of additive operations is not invariant over symmetric algorithms, but we have an easily specified relationship between given and derived algorithms.

THEOREM 12 (additive symmetry). *Let α be an algorithm with additive flow representation F_α for computing (m, n, p) products using t multiplication steps. Let α' be the standard algorithm corresponding to the addition flow representation F' derived from F_α by applying rotation and/or reflection operations. Then α' is additively optimal over the class of algorithms using t multiplication steps iff α uses $\mathcal{A}(m, n, p, t)$ additive operations, i.e., iff α is additively optimal.*

Proof. Assume α uses only $\mathcal{A}(m, n, p, t)$ additive operations and α' with representation F' is an algorithm for computing $\sigma(m, n, p)$ products in t multiplications, where σ is a symmetric permutation on three elements.

We note that by Corollaries 7 and 10 there is a fixed difference between the number of additive operations used by F and F^D, or by F and F^R. Also, note that $F^{D^2} = F$, that is, $D^{-1} = D$, where D is an operation on addition flow representations. In the same way, $R^{-1} = R^2$. In other words, $\{R, D\}$ generate $D3$, the dihedral group on three elements. Also, $(X \cdot Y)^{-1} = Y^{-1} \cdot X^{-1}$, where $X, Y \in D3$.

Thus, for any sequence $\mathcal{S}$ of D and R operations on F_α to yield F', there exists an inverse sequence of D and R operations, $\mathcal{S}^{-1}$, such that given any algorithm β with addition flow representation F_β using t multiplications to compute $\sigma(m, n, p)$ products, the standard algorithm with addition flow representation $(F_\beta)^{\mathcal{S}^{-1}}$ will compute (m, n, p) products using t multiplication steps.

By induction on the number of rotation and reflection operations in $\mathcal{S}$, we can show that

$$C^+(F_\alpha)^{\mathcal{S}} - C^+(F_\alpha) = C^+(F_\beta) - C^+(F_\beta)^{\mathcal{S}^{-1}}.$$

Thus, if α is additively optimal over t-multiplication algorithms, $C^+(F_\alpha) \leqq C^+(F_\beta)^{\mathcal{S}^{-1}}$. Therefore $C^+(F_\alpha)^{\mathcal{S}} \leqq C^+(F_\beta)$, where β was arbitrarily chosen. Then α' is an additively optimal t-multiplication algorithm for $\sigma(m, n, p)$ products. The proof of the converse is analogous. Thus the theorem holds.

In view of Corollaries 7 and 10, we have the following result.

COROLLARY 13. *Given any algorithm α with addition flow representation F_α which computes (m, n, p) products using* **a** *additive operations, there exist algorithms to compute*

(p, m, n)		$\mathbf{a}+(m-n)p$	
(n, p, m)		$\mathbf{a}+(p-n)m$	
(n, m, p)	*products using*	$\mathbf{a}+(m-n)p$	*additive operations.*
(m, p, n)		$\mathbf{a}+(p-n)m$	
(p, n, m)		$\mathbf{a}$	

Moreover, if we know of a bilinear t-multiplication algorithm which computes any of the above five products in fewer additive operations than stated above, we can derive a t-multiplication algorithm from it which uses fewer additive operations than α to compute (m, n, p) products.

A unifying observation is that the increase in additive complexity from an (m, n, p) product bilinear computation to a dual computation is exactly equal to the decrease in product matrix size. For example,

$$\mathcal{A}(n, p, m, t) - \mathcal{A}(m, n, p, t) = mp - mn.$$

A different formulation of this principle and proof of additive duality were first obtained by C. Fiduccia [4].

4. Ramifications of additive symmetry. In this section, we apply Theorem 12 to yield results about the essential additive complexity of matrix multiplication.

For example, we might ask which of the six related matrix products (listed in Corollary 13) has the highest or lowest essential additive complexity.

LEMMA 14. *Suppose it is possible to compute (m, n, p) products by an algorithm using t multiplication steps. Then $\mathcal{A}(u, v, w, t)$ is greatest (least) for all permutations (u, v, w) of (m, n, p) when $v = \max(\min)\{m, n, p\}$.*

Proof. Suppose we have an additively optimal algorithm α which uses t multiplications to compute (m, n, p) products. In other words, $C^+(F_\alpha) = \mathcal{A}(m, n, p, t)$. Then, by Theorem 11, $C^+(F_\alpha^*) = \mathcal{A}(u, v, w, t)$, where the standard algorithm with representation F^* ($*$ is some combination of R and D operations) computes

(u, v, w) products. If $v = \max(\min)\{m, n, p\}$, then $u - v$ and $w - v$ are each $\leqq 0$ $(\geqq 0)$. Thus, by Corollary 13, $\mathcal{A}(u, v, w, t) \geqq (\leqq) \mathcal{A}(\sigma(u, v, w), t)$.

This supports the intuitive feeling that for any fixed number of multiplications, the size of the inner product in any of six symmetric matrix multiplication problems directly influences additive complexity. In other words, for this class of matrix products, a set of a few product elements each with many terms has a higher additive complexity than a larger set of product elements each of which is a sum of fewer terms.

There are some matrix product computations which can be carried out at no additive cost. Algorithms which achieve this trivial additive lower bound of zero are therefore optimal. C. Fiduccia suggested to the author that by then employing Corollary 13, we can obtain directly the additive complexities of symmetric problems.

For example, $\mathcal{A}(n, 1, 1, n) = \mathcal{A}(m, 1, n, mn) = 0$.

Thus we immediately have the additive complexity of inner product and the product of a matrix by a vector from additive symmetry. Previous proofs of these results employed independence arguments.

LEMMA 15. *$n-1$ additive operations are necessary and sufficient to compute the inner product of two n-vectors.*

Proof. From [19], $\mathcal{M}(1, n, 1) = n$. By Corollary 13, $\mathcal{A}(1, n, 1, n) = \mathcal{A}(n, 1, 1, n) + (n-1) \cdot 1 = n - 1$.

LEMMA 16. *$mn - m$ additive operations are necessary and sufficient to compute the product of an $m \times n$ or $n \times m$ matrix with an n-vector.*

Proof. $\mathcal{M}(m, 1, n) = mn$ [19]. By Corollary 13, $\mathcal{A}(m, n, 1, mn) = \mathcal{A}(m, 1, n, mn) + (n-1)m = mn - m$. By Theorem 12, $\mathcal{A}(1, n, m, mn) = mn - m$ as well.

Now we derive the additive complexity of multiplying 2×2 matrices by 7-multiplication algorithms.

Since Strassen's algorithm α_S introduced earlier uses 7 multiplications and 18 additive operations, $\mathcal{A}(2, 2, 2, 7) \leqq 18$. Winograd communicated to the author a variant of the following 15-addition/subtraction algorithm which he discovered.

LEMMA 17. $\mathcal{A}(2, 2, 2, 7) \leqq 15$.

Proof. Compute $A_{2\times 2} B_{2\times 2} = Y_{2\times 2}$ as follows (algorithm α_W): form

$$
\begin{aligned}
s_1 &\leftarrow a_{21} + a_{22}, & s_5 &\leftarrow b_{12} - b_{11},\\
s_2 &\leftarrow s_1 - a_{11}, & s_6 &\leftarrow b_{22} - s_5\\
s_3 &\leftarrow a_{11} - a_{21}, & s_7 &\leftarrow b_{22} - b_{12},\\
s_4 &\leftarrow a_{12} - s_2, & s_8 &\leftarrow s_6 - b_{21},
\end{aligned}
$$

using 8 additions and subtractions.

Then calculate the following 7 multiplications:

$$
\begin{aligned}
M_1 &\leftarrow s_2 \cdot s_6, & M_4 &\leftarrow s_3 \cdot s_7,\\
M_2 &\leftarrow -a_{11} \cdot b_{11}, & M_5 &\leftarrow s_1 \cdot s_5,\\
M_3 &\leftarrow a_{12} \cdot b_{21}, & M_6 &\leftarrow s_4 \cdot b_{22},\\
& & M_7 &\leftarrow a_{22} \cdot s_8.
\end{aligned}
$$

Finally, using only 7 more additions/subtractions, form

$$s_{10} \leftarrow M_3 - M_2, \quad s_{13} \leftarrow s_{11} + M_5,$$
$$s_{11} \leftarrow M_1 - M_2, \quad s_{14} \leftarrow s_{13} + M_6,$$
$$s_{12} \leftarrow s_{11} + M_4, \quad s_{15} \leftarrow s_{12} - M_7,$$
$$s_{16} \leftarrow s_{12} + M_5.$$

Then

$$y_{11} = s_{10}, \quad y_{12} = s_{14},$$
$$y_{21} = s_{15}, \quad y_{22} = s_{16}.$$

Although upper bounds are not really the main subject of this paper, we should note that this algorithm can be used recursively according to various implementation strategies to multiply square matrices.

THEOREM 18 [5], [13]. *Using a dynamic implementation strategy (a strategy tailored to the numerical properties of n), matrices of order n can be multiplied using fewer than* $4.54\, n^{\log 7}$ *total arithmetic operations. In the best case, when* $n = 2^p$ *for arbitrary positive p, only* $3.73\, n^{\log 7}$ *total arithmetic operations need be used asymptotically.*

The remainder of this section is devoted to proving a lower bound on $\mathcal{A}(2, 2, 2, 7)$.

Suppose $F = \langle G_1, G_2, G_3 \rangle$ is an addition flow representation. $C^+(G_i)$, where $i \in \{1, 2, 3\}$ will denote the number of additive operations represented by G_i.

Thus, $C^+(G_i) = |\Gamma G_i| - |VG_i| + |R_i|$.

Now, if we examine all algorithms (with addition flow representations) which compute (2, 2, 2, 7) products, we will find that every algorithm uses at least 4 additive operations to form the left-hand sides of the seven multiplications that algorithm uses. In other words, for any addition flow representation F for a 7-multiplication algorithm which computes (2, 2, 2) products, we can show that $C^+(G_i) \geqq 4$, where $F = \langle G_1, G_2, G_3 \rangle$. Initially, we consider only computations over $GF(2)$.

LEMMA 19. *If the standard algorithm with addition flow representation* $F = \langle G_1, G_2, G_3 \rangle$ *computes* (2, 2, 2) *products using 7 multiplications, then* $C^+(G_1) \geqq 4$, *where the computation is over* $GF(2)$.

Proof. This result is proved in [13]; we present a sketch of the proof here.

Hopcroft and Kerr [6] prove that 7 multiplications are required to multiply two general matrices A, B of order 2. Working over $GF(2)$, they divide the set of possible left-hand sides of the seven multiplications into two disjoint subsets: S_1, the set of left-hand sides with "diagonal" character [6], [7], [13], and S_2, the remaining left-hand sides. For example, $(a_{11} + a_{12} + a_{21})$ is an element of S_1, whereas $(a_{11} + a_{12} + a_{21} + a_{22}) \in S_2$.

Then, they prove the following.

LEMMA 20. *If an algorithm for* $A \times B$ *has k left-hand sides from* a_{11}, $(a_{12} + a_{21})$, $(a_{11} + a_{12} + a_{21})$, *then the algorithm requires* $6 + k$ *multiplications.*

COROLLARY 21 [6]. *Similar theorems hold for left-hand sides from*

(a) $a_{21}, (a_{11}+a_{22}), (a_{11}+a_{21}+a_{22})$,
(b) $a_{12}, (a_{11}+a_{22}), (a_{11}+a_{12}+a_{22})$,
(c) $a_{22}, (a_{12}+a_{21}), (a_{12}+a_{21}+a_{22})$.

Using a straightforward argument from linear algebra, they show the following result.

LEMMA 22. *Any algorithm for $A \times B$ whose multiplications have k left-hand sides from*

(a) $a_{11}, a_{12}, a_{11}+a_{12}$ *or*
(b) $a_{21}, a_{22}, a_{21}+a_{22}$

requires $\lceil 6+k/2 \rceil$ *multiplications.*

Now suppose that there exists a flow representation $F_\alpha = \langle G_1, G_2, G_3 \rangle$, where $C^+(G_1) \leqq 3$, and α computes (2, 2, 2) products using 7 multiplications.

By Lemma 20, each of $a_{11}, a_{12}, a_{21}, a_{22}$ can appear as a left-hand side at most once; otherwise α would require 8 multiplications. In fact, we can use the original statement of Lemma 20 [6] to demonstrate that no left-hand side of multiplications in α is repeated. Therefore, α has $a_{11}, a_{12}, a_{21}, a_{22}$ as four left-hand sides, and the remaining three left-hand sides are computable using 3 additive operations. This means that at least one left-hand side is the sum of exactly two elements of A.

There are exactly six such possible sums. By Lemma 20, Corollary 21 and Lemma 22, if any such sum were present, α would require 8 multiplications, a contradiction. For example, $(a_{11}+a_{12})$ could not be a left-hand side of a multiplication in α; this would imply α contains 3 left-hand sides from group (a) of Lemma 22 and would therefore require 8 multiplications.

Therefore, no left-hand side can be the sum of exactly two elements of A. Therefore no such standard algorithm, and hence no such addition flow representation, exists, proving Lemma 19.

COROLLARY 23. *If the standard algorithm with addition flow representation $F = \langle G_1, G_2, G_3 \rangle$ computes* (2, 2, 2) *matrix products in* 7 *multiplications, then* $C^+(G_2) \geqq 4$.

Proof. Assume $C^+(G_2) \leqq 3$. By Lemma 8, the standard algorithm with addition flow representation $\langle G_1', G_2', G_3' \rangle = \langle G_2^T, G_1^T, G_3^T \rangle$ computes (2, 2, 2) products in 7 multiplication steps. Then $C^+(G_1') = C^+(G_2^T) = C^+(G_2) \leqq 3$. Thus we have derived a 7-multiplication algorithm which computes (2, 2, 2) products using only three additive operations to form the left-hand sides of its multiplications. But this contradicts Lemma 19, proving the corollary.

Thus, 8 additive operations are necessary and sufficient to form the left-hand and right-hand sides of the multiplications in a 7-multiplication (2, 2, 2) algorithm. Again using additive symmetry, we can show the following.

LEMMA 24. *If the standard algorithm with addition flow representation $F = \langle G_1, G_2, G_3 \rangle$ computes* (2, 2, 2) *matrix products using* 7 *multiplications, then* $C^+(G_3) \geqq 7$.

Proof. The standard algorithm with representation $F^D = \langle G_1', G_2', G_3' \rangle$ computes (2, 2, 2) products in 7 multiplications by Theorem 6. But

$C^+(G_2') = C^+(G_3^D) \geqq 4$ by Corollary 23. Therefore

$$|\Gamma G_3^D| - |VG_3^D| + |R_3^D| \geqq 4,$$
$$(|\Gamma G_3| - |VG_3| + |R_3|) + |S_3| \geqq 4 + |R_3|.$$

The first expression is $C^+(G_3)$, $|R_3| = 7$ and $|S_3| = 4$. Thus we have

$$C^+(G_3) + 4 \geqq 4 + 7 \quad \text{or} \quad C^+(G_3) \geqq 7,$$

as required.

This completes the proof of a lower bound of 15 additive operations on the additive complexity of computing (2, 2, 2) products with fast bilinear algorithms over $GF(2)$. This result, then, will hold for any algebraic structure from which there is a homomorphism (over all operations) to $GF(2)$. For example, since bilinear computations over the ring of integers are homomorphic to bilinear computations in $GF(2)$ (with subtraction over Z mapped to addition over $GF(2)$), this lower bound holds for bilinear computations over the ring of integers involving addition, subtraction and multiplication. Thus, for such structures, α_W is additively optimal, and we have the next theorem.

THEOREM 25. $\mathcal{A}(2, 2, 2, 7) = 15$. *In other words,* 15 *additive operations are necessary and sufficient to compute the product of matrices of order two using no more than* 7 *bilinear multiplications.*

COROLLARY 26. *The obvious way of computing* (2, 2, 2) *products* (8 *multiplications,* 4 *additions*) *is optimal with respect to total arithmetic operations.*

5. Extensions and open problems. Several extensions of these results and suggestions for further research are given in [13].

One of the more natural extensions would seem to be from a product of two matrices to a product of n matrices, where n is any arbitrary integer greater than or equal to two.

By generalizing the above definitions and iterating applications of Theorem 6 (reflection operation) and Theorem 9 (rotation operation), the reader can derive the following.

THEOREM 27. *Suppose* α *is an algorithm with addition flow representation* F_α *which computes* $(m_1, \cdots, m_{n+1})$ *products* $(n \geqq 2)$ *using t multiplication steps. Then, using rotation and reflection operations on* F_α, *we can derive an algorithm which computes* $(p_1, p_2, \cdots, p_{n+1})$ *products in t multiplications, where* $(p_1, p_2, \cdots, p_{n+1})$ *is any dihedral permutation of* $(m_1, m_2, \cdots, m_{n+1})$. *Also, by using Corollaries* 7 *and* 10, *we can directly relate the additive cost of any algorithm for matrix products derivable from* α *to* $C^+(F_\alpha)$.

Thus the problem of multiplying n matrices by algorithms which have addition flow representations is actually a set of subproblems which may be partitioned into $n!/2$ equivalence classes of $(m_1, \cdots, m_{n+1})$ products. Two problems are in the same duality class if there is a dihedral transformation of a flow representation of an optimal algorithm for one into a representation of an algorithm for the other problem. All problems in a duality class have the same complexity; problems in distinct duality classes may or may not have the same complexity.

It should be recognized that addition flow representations or generalized versions of them facilitate the study of a wide variety of dual computations. One important subclass of this class of dual computations consists of computations for those problems which are preserved by duality transformations. Most of this paper has treated only one such problem, namely the multiplication of two matrices. The duality operators, R and D, applied to representations of matrix product computations yield representations of matrix product computations. Similar duality operators preserve the problem of computing sets of linear forms by linear algorithms [13]. Some other "problems of dual complexity" together with extensions of the type of duality presented above will be presented in a subsequent paper.

It may be possible to combine independence arguments on algorithms with addition flow representations to extend our knowledge of lower bounds on matrix multiplication.

To be more specific, we make the following conjecture.

Conjecture. If the standard algorithm with addition flow representation $F=\langle G_1, G_2, G_3\rangle$ is additively optimal for computing (n, n, n) products $(n\geqq 2)$ over algorithms which use t multiplications, then $C^+(G_1)=C^+(G_2)=G^+(G_3^D)$.

Note that the converse does not hold; F_{α_S} is a suitable counter-example.

In other words, we conjecture that any additively optimal algorithm will be uniformly additively optimal over each of the three additive stages, forming left-hand sides of the multiplication steps, forming right-hand sides, and summing the calculated multiplications to yield the matrix product elements. For example, using the operations of R and D on F_{α_W} could not yield an additive improvement over α_W in one of the three computation stages.

To use the conjecture, consider for now only the number of additive operations employed in the first stage of any t-multiplication algorithm to compute (n, n, n) products; i.e., consider $C^+(G_1)$. Let us restrict the class of algorithms under consideration to those algorithms with distinct left-hand sides of the t multiplications. This eliminates, for example, the classical algorithm from discussion, but permits α_S and α_W. Then we can immediately see that

$$C^+(G_1)\geqq t-n^2$$

since there are at most n^2 left-hand sides which consist of a single term. Thus we have the following.

LEMMA 28. $\mathscr{A}(n, n, n, t)\geqq 4t-4n^2$.

Proof. Assume $F_\alpha=\langle G_1, G_2, G_3\rangle$ is the addition flow representation corresponding to an optimal t-multiplication algorithm α for computing (n, n, n) products. Then, by the above argument and the conjecture, $C^+(G_2)=C^+(G_1)\geqq t-n^2$. By the conjecture, $t-n^2\leqq C^+(G_3^D)$, which equals $C^+(G_3)+|S_3|-|R_3|$. In other words,

$$\begin{aligned} C^+(G_3) &= C^+(G_3^D)-|S_3|+|R_3| \\ &\geqq (t-n^2)-n^2+t=2t-2n^2. \end{aligned}$$

Therefore $C^+(F_\alpha)\geqq 4t-4n^2$. But α was chosen additively optimal over t-multiplication algorithms.

Therefore $\mathscr{A}(n, n, n, t)\geqq 4t-4n^2$, as required.

This would demonstrate that the additive complexity of square matrix multiplication is essentially a linear multiple of the multiplicative complexity over this restricted class of algorithms, since by [9] the multiplicative complexity grows as fast as n^2.

In [1] it is claimed that $\mathcal{M}(n, n, n) \geqq 3n^2 - 3n + 1$. As a result, we would have (over the class of bilinear algorithms) the following corollary.

COROLLARY 29. $\mathcal{A}(n, n, n, \mathcal{M}(n, n, n)) \geqq 8n^2 - 12n + 4$.

Thus, if the conjecture holds, algorithms with distinct left-hand and right-hand sides of multiplications, which use as few multiplication steps as possible, must employ at least $8n^2 - 12n + 4$ additive operations to multiply two matrices of order n.

Recently, this claim has been corrected to $\mathcal{M}(n, n, n) \geqq 2n^2 - 1$. Thus, proof of our conjecture would yield a lower bound on the number of additive operations used by any multiplicatively optimal bilinear matrix multiplication algorithm of $4n^2 - 4$. In other words, $\mathcal{A}(n, n, n, \mathcal{M}(n, n, n)) \geqq 4n^2 - 4$.

6. Conclusions. In this paper, we introduced a graph-theoretic model of bilinear algorithms, called an addition flow representation. In particular, we used this model to investigate lower bounds on the number of additions and subtractions required to multiply matrices. By introducing operations of reflection and rotation on these representations, we related the additive complexity of one matrix multiplication problem to five symmetric problems in a constructive way. In other words, any additive savings we may discover for a related problem can be utilized to improve a given algorithm for the original matrix multiplication problem. As an application of the model, this "additive symmetry" was employed to prove that the essential additive complexity of multiplying 2×2 matrices using 7 multiplication steps is exactly 15 additive operations. This means, for example, that recursive schemes to multiply matrices based on the algorithm α_W given above are in principle as efficient as possible; further improvements in arithmetic efficiency can result only from improvements in implementation details (see, for example, [5]), or from finding a base bilinear algorithm for multiplying $m \times m$ matrices in fewer than $7^{\log m}$ multiplication steps.

Acknowledgments. The author is very grateful to Dr. Patrick Fischer for his suggestions and encouragement, especially during the author's doctoral research period, when most of these results were obtained. Also, the author would like to thank Dr. Charles Fiduccia for many stimulating discussions and insightful suggestions. Gratitude is also extended to Dr. Shmuel Winograd for conversations which aided the author's understanding of many of the problems in this area. Finally, the referees' comments were incisive and helpful and much appreciated.

REFERENCES

[1] D. DOBKIN AND R. BROCKETT, *On the optimal evaluation of a set of bilinear forms*, Proc. 5th Ann. Symp. on Theory of Computing, Austin, Texas, Association for Computing Machinery, New York, 1973, pp. 88–95.

[2] C. FIDUCCIA, *Fast matrix multiplication*, Proc. 3rd Ann. Symp. on Theory of Computing, Shaker Heights, Ohio, Association for Computing Machinery, New York, 1971, pp. 45–49.

[3] ———, *On obtaining upper bounds on the complexity of matrix multiplication*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 31–40.

[4] ———, *On the algebraic complexity of matrix multiplication*, Doctoral thesis, Brown University, Providence, R.I., 1973.

[5] P. C. FISHER AND R. L. PROBERT, *Efficient procedures for using matrix algorithms*, Lecture Notes in Computer Science 14, G. Goos and J. Hartmanis, eds., Springer-Verlag, Heidelberg, 1974, pp. 413–427.

[6] J. E. HOPCROFT AND L. B. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, Tech. Rep. 69–44, Cornell Univ., Ithaca, N.Y., 1969.

[7] J. E. HOPCROFT AND J. MUSINSKI, *Duality applied to the complexity of matrix multiplication and other bilinear forms*, Proc. 5th Ann. Symp. on Theory of Computing, Austin, Texas, Association for Computing Machinery, New York, 1973, pp. 73–87.

[8] D. KIRKPATRICK, *On the additions necessary to compute certain functions*, Tech. Rep. No. 39, Univ. of Toronto, 1972.

[9] ———, *A note on the complexity of independent rational functions*, Unpublished manuscript, 1974.

[10] Z. KEDEM, *The reduction method for establishing lower bounds on the number of additions*, MAC Tech. Memo. 48, Mass. Inst. of Tech., Cambridge, 1974.

[11] I. MUNRO, *Problems related to matrix multiplication*, Computational Complexity, R. Rustin, ed., Algorithmics Press, New York, 1973, pp. 137–151.

[12] R. L. PROBERT, *On the complexity of symmetric computations*, INFOR—Canad. J. Operational Res. and Information Processing, 12 (1974), pp. 71–86.

[13] ———, *On the complexity of matrix multiplication*, Tech. Rep. CS-73-27, Univ. of Waterloo, 1973.

[14] ———, *Commutativity, non-commutativity, and bilinearity*, Research Rep., Univ. of Saskatchewan, 1975.

[15] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[16] ———, *Evaluation of rational functions*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, 1972, pp. 1–10.

[17] ———, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

[18] S. WINOGRAD, *On the algebraic complexity of inner product*, IBM Res. Rep. RC-2729, T. J. Watson Research Center, Yorktown Heights, N.Y., 1969.

[19] ———, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

[20] ———, *On the algebraic complexity of functions*, IBM Res. Rep. T. J. Watson Research Center, Yorktown Heights, N.Y., 1970.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

GRAPH TRANSFORMATIONS FOR ROUNDOFF ANALYSIS*

WEBB MILLER†

Abstract. When analyzing a numerical algorithm, it is often possible to show that a rounding error at one floating-point operation is equivalent to errors at other operations. In such cases, it can be concluded that no generality is lost if certain operations are considered error-free. Sometimes this conclusion can be reached automatically and inexpensively compared to the cost of the ultimate roundoff analysis. It may be advantageous to use a preprocessor which performs this reduction before other automatic techniques are invoked.

In this report we consider a class of elementary rounding error reductions which are most naturally interpreted as graph transformations. This leads to questions concerning heuristics and optimal strategies for the application of these transformations.

Key words. automatic roundoff analysis, numerical stability

1. Introduction. In this paper we will consider a class of simple transformations which can be interpreted as relating rounding errors in various floating-point operations of an algorithm. Our motivation is to explore the possibility of automating the application of these transformations to lessen the cost of techniques for roundoff analysis like those of Miller [6]–[8]. One of the methods derived here is very inexpensive to implement and seems to be useful, especially for numerical algorithms in which any input or intermediate computed values is used as an operand in only a few subsequent operations.

To motivate later developments, let us consider the evaluation of $z(a, b, c) = ab + ac + bc$ by the algorithm

$$\begin{aligned} w &\leftarrow b + c \\ x &\leftarrow b \times c \\ y &\leftarrow a \times w \\ z &\leftarrow y + x\,. \end{aligned} \tag{1.0}$$

If the evaluation is performed in floating-point arithmetic, then relative errors δ_j are committed, so the values actually computed are

$$\begin{aligned} w &= (b + c)(1 + \delta_w), \\ x &= bc(1 + \delta_x), \\ y &= a[(b + c)(1 + \delta_w)](1 + \delta_y), \\ z &= \{a[(b + c)(1 + \delta_w)](1 + \delta_y) + bc(1 + \delta_x)\}(1 + \delta_z). \end{aligned} \tag{1.1}$$

The δ's are bounded by some miniscule machine constant $u > 0$.

This evaluation can be pictured as Fig. 1. The exact result of an operation (and later, an exact data item) is multiplied by the factor associated with that vertex (the factor 1 is assumed at unmarked vertices).

---

* Received by the editors October 17, 1974, and in revised form April 24, 1975.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. This work was supported in part by the National Science Foundation under Grant GJ-42968.

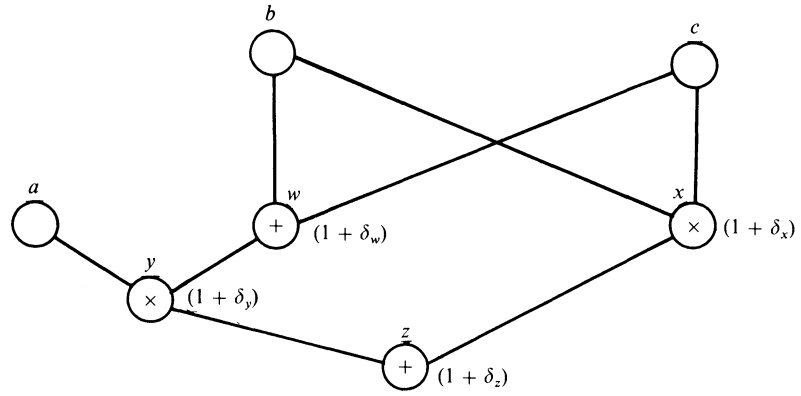


FIG. 1

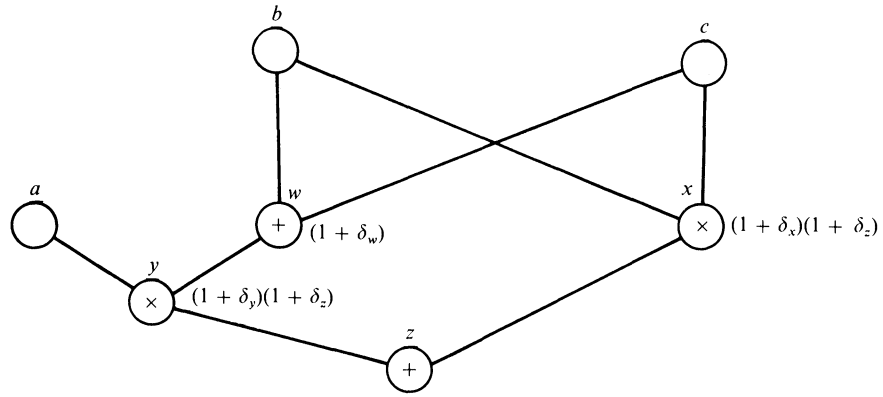


FIG. 2

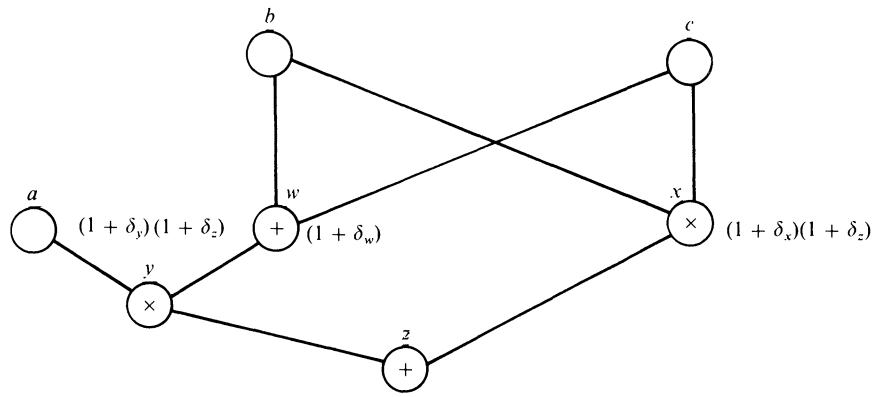


FIG. 3

Let us rewrite (1.1) as

$$z = a[(b + c)(1 + \delta_w)](1 + \delta_y)(1 + \delta_z) + bc(1 + \delta_x)(1 + \delta_z). \tag{1.2}$$

This corresponds to moving copies of $(1 + \delta_z)$ up the two arcs entering z (Fig. 2). The point to notice is this: for fixed a, b, c and arbitrary rounding errors δ_w, δ_x, δ_y, δ_z (at the corresponding vertices) giving a computed value z, we have that z is also computed with rounding errors δ_w in w, $\delta_y + \delta_z + \delta_y\delta_z$ in y (since $(1 + \delta_y) \cdot (1 + \delta_z) = 1 + (\delta_y + \delta_z + \delta_y\delta_z)$) and $\delta_x + \delta_z + \delta_x\delta_z$ in x. These last two errors are each bounded by $2u + u^2$ or, neglecting second-order terms, by $2u$. This means that for many purposes we need only worry about the sensitivity of z to rounding errors in the first three operations.

A direction commonly taken in roundoff analysis is to show the reducibility of rounding errors in the operations to rounding errors in the data. Rewriting (1.2) as

$$z = [a(1 + \delta_y)(1 + \delta_z)] \cdot [(b + c)(1 + \delta_w)] + bc(1 + \delta_x)(1 + \delta_z), \tag{1.3}$$

we have the graph interpretation that a rounding error has moved up to *one* of the operands of y (not to *both* since the operation is multiplicative not additive); see Fig. 3.

We can split the factor at x into two equal parts $(1 + \delta_x)^{1/2}(1 + \delta_z)^{1/2}$ which can be slid up to the operands b and c. The resulting effect upon w can be neutralized by a factor $(1 + \delta_x)^{-1/2}(1 + \delta_z)^{-1/2}$. Algebraically:

$$z = [a(1 + \eta_a)] \cdot [b(1 + \eta_b) + c(1 + \eta_c)] \cdot (1 + \eta_w) + [b(1 + \eta_b)] \cdot [c(1 + \eta_c)], \tag{1.4}$$

where

$$\begin{aligned}(1 + \eta_a) &= (1 + \delta_y)(1 + \delta_z),\\ (1 + \eta_b) &= (1 + \eta_c) = (1 + \delta_x)^{1/2}(1 + \delta_z)^{1/2},\\ (1 + \eta_w) &= (1 + \delta_w)(1 + \delta_x)^{-1/2}(1 + \delta_z)^{-1/2}.\end{aligned}$$

Graphically: see Fig. 4.

The equivalence of Figs. 1 and 4 intuitively implies that if we neglect those rounding errors which are merely equivalent to a few rounding errors in the data, then we can limit our attention to the error in the operation giving w. (Can the reader find a reduction which proceeds further and shows that the computed z is the exact result for slightly altered data, i.e., can the reader push all errors up to a, b and c? See § 3 for a solution.)

In the next section we will formalize a rule covering transformations which, like those above, "clear" a vertex of rounding errors. Instead of trying to survey the range of possible uses for such transformations, we will confine ourselves to a particular type of roundoff analysis discussed in §§ 3 and 5. We conclude by mentioning some extensions and open questions.

The reader may find it informative to compare this paper with Bauer [2], who touches on problems similar to those of this paper, namely: when can the effect of the rounding error of a step in a computation be ignored? In turn, the

"computational graphs" of that paper should be compared with the "process graphs" of McCracken and Dorn [4].

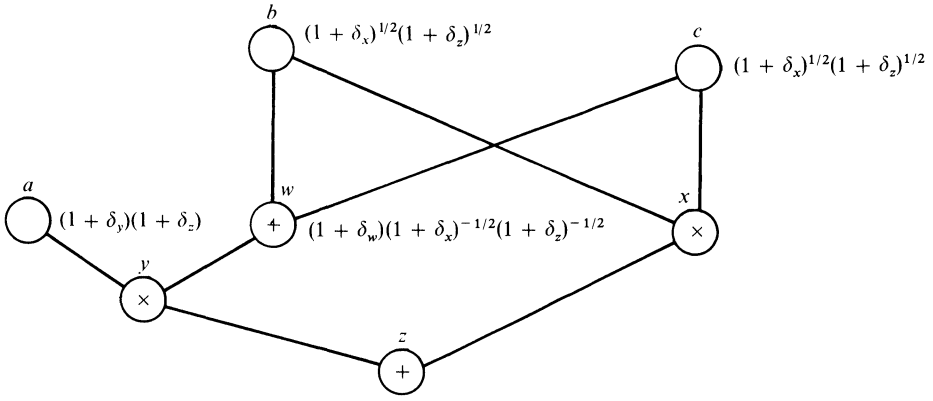


FIG. 4

2. The transformations. A graph specifying a sequence of arithmetic operations $+$, $-$, $\times$ or $/$ (e.g., Fig. 1, possibly without the labels $(1+\delta_j)$) will be called a *computation graph.* Any vertex is either an *input vertex* with no predecessors or an *operation vertex* with two predecessors. In addition, some of the vertices are specified as *output vertices.*

Define a relation Σ between vertices u and v of a computation graph: $u\Sigma v$ if there is a sequence $v_1, \cdots, v_k$ of vertices with $u = v_1$, $v = v_k$ and where for $i = 1, \cdots, k-1$, v_i and v_{i+1} are the operands in some $\pm$ operation (this is meant to imply $u\Sigma u$ for all u). For example, in Fig. 1 we have $b\Sigma c$ and $x\Sigma y$.

Σ is an equivalence relation whose equivalence classes are called Σ-*chains.* A *proper* Σ-chain has the additional property that if u and v are in S, then u is not an operand for v. The Σ-chains in Fig. 1 are $\{a\}$, $\{b, c\}$, $\{w\}$, $\{x, y\}$ and $\{z\}$. All are proper.

If S is a set of vertices, then ID(S) denotes the set of immediate descendants of elements of S. Thus a Σ-chain S is proper if $S \cap \mathrm{ID}(S)$ is empty. A *proper* Σ-*family* has the form $S \cup \mathrm{ID}(S)$, where S is a proper Σ-chain. Two of the Σ-families in Fig. 1 are $\{a, y\}$ and $\{b, c, w, x\}$.

A *floating-point graph* is a pair (G, ρ), where G is a computation graph and ρ assigns to every vertex v of G a rational expression $\rho(v)$ in the variables $\delta_1, \delta_2, \cdots$. There is a natural process for assigning to each v a rational expression VAL(v) in the δ_i and the (names of) the input vertices. Namely,

(i) if v is an input vertex, then $\mathrm{VAL}(v) = v \cdot \rho(v)$;

(ii) if v is an operation vertex $v \leftarrow u \# w$, then $\mathrm{VAL}(v) = [\mathrm{VAL}(u) \# \mathrm{VAL}(w)] \cdot \rho(v)$.

In Fig. 1, VAL(z) is equivalent to the expressions appearing on the right of (1.1)–(1.4).

DEFINITION 2.1. Let S be a proper Σ-chain in a floating-point graph (G, ρ) and let α be a rational expression in the δ's. The process of *lowering* α *from* S produces (G, ρ^*), where ρ^* is defined as follows.

1. $\rho^*(v) = \rho(v)$ for v not in $S \cup \mathrm{ID}(S)$.
2. $\rho^*(s) = \alpha^{-1} \cdot \rho(s)$ for s in S.
3. For t in $\mathrm{ID}(S)$, set
   (a) $\rho^*(t) = \alpha \cdot \rho(t)$ if t is a $\pm$-vertex or if $t \leftarrow s \cdot v$, $t \leftarrow v \cdot s$ or $t \leftarrow s/v$ with s in S, v not in S.
   (b) $\rho^*(t) = \alpha^{-1} \cdot \rho(t)$ if $t \leftarrow v/s$ with s in S, v not in S.
   (c) $\rho^*(t) = \alpha^2 \cdot \rho(t)$ if $t \leftarrow s \cdot r$ with s and r in S.
   (d) $\rho^*(t) = \rho(t)$ if $t \leftarrow s/r$ with s and r in S.

(Notice that 2 and 3 do not conflict since S is proper).

DEFINITION 2.2. Let S be as in Definition 2.1 and let $S \cup \mathrm{ID}(S)$ contain t. We exclude the case 3(d) $t \leftarrow s/r$ with s and r in S. According to Definition 2.1, there is an α such that lowering α from S *clears* t, i.e., yields $\rho^*(t) = 1$.

Let us return to Fig. 1 for an example. We can clear x either by lowering $(1 + \delta_x)^{-1/2}$ from $\{b, c\}$ or by lowering $(1 + \delta_x)$ from $\{x, y\}$.

The clearing of t has only a localized effect upon the floating-point graph; it changes $\mathrm{VAL}(s)$ for s in S, but otherwise leaves $\mathrm{VAL}(v)$ unchanged. For the following result, let ρ^* be gotten by lowering α from a proper Σ-chain S and let $\mathrm{VAL}^*(v)$ be gotten by using ρ^* instead of ρ and VAL^* instead of VAL in the recursive definition (i), (ii). By, e.g., $\mathrm{VAL}^*(v) = \mathrm{VAL}(v)$ we naturally mean that these rational expressions are equivalent (i.e., determine functions which agree whenever they are both defined), not that they are identical expressions.

THEOREM 2.1. *From the preceding assumptions we can conclude that* $\mathrm{VAL}^*(s) = \alpha^{-1} \cdot \mathrm{VAL}(s)$ *for s in S and that* $\mathrm{VAL}^*(v) = \mathrm{VAL}(v)$ *for v not in S.*

Proof. First suppose v is an input vertex. If v is not in $S \cup \mathrm{ID}(S)$, then $\mathrm{VAL}^*(v) = v \cdot \rho^*(v) = v \cdot \rho(v) = \mathrm{VAL}(v)$. The only other possibility is that v is in S, whence $\mathrm{VAL}^*(v) = v \cdot \rho^*(v) = v \cdot \alpha^{-1} \cdot \rho(v) = \alpha^{-1} \cdot \mathrm{VAL}(v)$.

Now suppose that v is defined by $v \leftarrow x \# y$. By induction we may assume the theorem true of x and y. If v is not in $S \cup \mathrm{ID}(S)$, then x and y are not in S and $\mathrm{VAL}^*(v) = [\mathrm{VAL}^*(x) \# \mathrm{VAL}^*(y)] \cdot \rho^*(v) = [\mathrm{VAL}(x) \# \mathrm{VAL}(y)] \cdot \rho(v) = \mathrm{VAL}(v)$. If v is in S, then neither x nor y is in S since S is proper, hence $\mathrm{VAL}^*(v) = [\mathrm{VAL}^*(x) \# \mathrm{VAL}^*(y)] \cdot \rho^*(v) = [\mathrm{VAL}(x) \# \mathrm{VAL}(y)] \cdot \alpha^{-1} \cdot \rho(v) = \alpha^{-1} \cdot \mathrm{VAL}(v)$. Lastly, suppose v is in $\mathrm{ID}(S)$. The proof proceeds by subcases corresponding to parts 3(a) to 3(d) of Definition 2.1. For instance, if 3(b) applies ($\#$ is $/$, x not in S, y in S), then $\mathrm{VAL}^*(v) = [\mathrm{VAL}^*(x)/\mathrm{VAL}^*(y)] \cdot \rho^*(v) = [\mathrm{VAL}(x)/(\alpha^{-1} \cdot \mathrm{VAL}(y))] \cdot \alpha^{-1} \cdot \rho(v) = [\mathrm{VAL}(x)/\mathrm{VAL}(y)] \cdot \rho(v) = \mathrm{VAL}(v)$. □

Intuitively, the motivation behind this section is the desire to make the simplest possible transfer of the rounding error at one operation to other operations. If a rounding error $\alpha = (1 + \delta_j)$ is removed from one computed value, then the effect upon operations using this value as an operand can be neutralized according to two cases. The easier case is that of a multiplicative (i.e., $\times$ or $/$) operation, since there small relative perturbations in one or both operands cause at most a small relative perturbation in the result. On the other hand, the easiest way to guarantee that the result of a dependent additive (i.e., $+$ or $-$) operation suffers only a small relative error is to introduce the same relative perturbation into *both* of its operands. This leads naturally to the notion of a Σ-chain and to the above procedure.

The reader may find it instructive to formalize and prove the following result. Consider a straight-line program using only the operations $+$, $-$, $\times$ and $/$ which has the properties that (i) no output value is used as an operand in a subsequent operation, and (ii) no input or intermediate value is used more than once. The computed outputs are the exact results for slightly altered data.

3. A graph game for roundoff analysis. Many applications of the above ideas, e.g., those discussed in § 5, have the following properties.

1. It is not necessary to keep track of the exact form of the error $\rho(v)$ associated with vertex v. Instead we need only remember whether v is known to have $\rho(v) = 1$ ("v is cleared") or not ("v is marked").

2. Marks (i.e., nontrivial ρ's) associated with input vertices can be ignored.

For such applications, the previous results suggest the following game.

GAME. Begin with a computation graph, all of whose operation vertices are "marked". A *move* begins by selecting a proper Σ-chain S and a marked vertex t in $S \cup \mathrm{ID}(S)$ subject to the constraints (i) S may not contain an output vertex, and (ii) t may not have the form $t \leftarrow s/r$ with s and r in S. The move is completed by marking all unmarked operation vertices in $S \cup \mathrm{ID}(S)$, then clearing t. The goal is to find a sequence of moves after which as many as possible of the vertices are cleared (if all of them can be cleared, then the algorithm enjoys a type of numerical stability discussed at the end of § 5 and in Miller [5]).

The fact that errors moved up to the input vertices can be ignored suggests the following strategy.

The bottom-up heuristic. Let the computation graph be associated with a straight-line program. Begin with the last instruction and work upward to the first as follows. At vertex $v \leftarrow x \# y$, try to make a move (S, v) with v in $\mathrm{ID}(S)$ which marks no vertex already cleared (for multiplicative operations where the operands lie in different Σ-chains S_1 and S_2, one is free to choose either (S_1, v) or (S_2, v)). If no such move exists, then by-pass v.

Let us denote a move by, e.g., $M = \{s_1, \cdots, s_n | i_1, \cdots, i_m\}$, where $S = \{s_1, \cdots, s_n\}$ and $\mathrm{ID}(S) = \{i_1, \cdots, i_m\}$, and let us denote in bold face the t in M which is cleared. Thus in the example of § 1, we essentially applied the bottom-up heuristic to clear x, y and z with the moves $\{x, y|\mathbf{z}\}$, $\{a|\mathbf{y}\}$ and $\{b, c|w, \mathbf{x}\}$.

For algorithm (1.0), the bottom-up heuristic does not produce an optimal strategy for the GAME. The sequence of moves $\{x, y|\mathbf{z}\}$, $\{b, c|w, \mathbf{x}\}$, $\{\mathbf{w}|y\}$, $\{a|\mathbf{y}\}$ clears all vertices, proving the numerical stability of (1.0).

Example 3.1. Consider the problem of solving for y in the system of linear equations

$$\begin{aligned} a_1 x + b_1 y &= f_1, \\ c_1 x + a_2 y + b_2 z &= f_2, \\ c_2 y + a_3 z &= f_3. \end{aligned}$$

Figure 5 is Gaussian elimination without pivoting, while Fig. 6 shows "two-sided" elimination from Babuška [1]. Figures 5 and 6 are the algorithms considered in Case Study I of Miller [6].

Applying the bottom-up heuristic to Fig. 5 clears all vertices except s_2 (or t_2, depending on which is cleared first). If all vertices could be cleared, it would

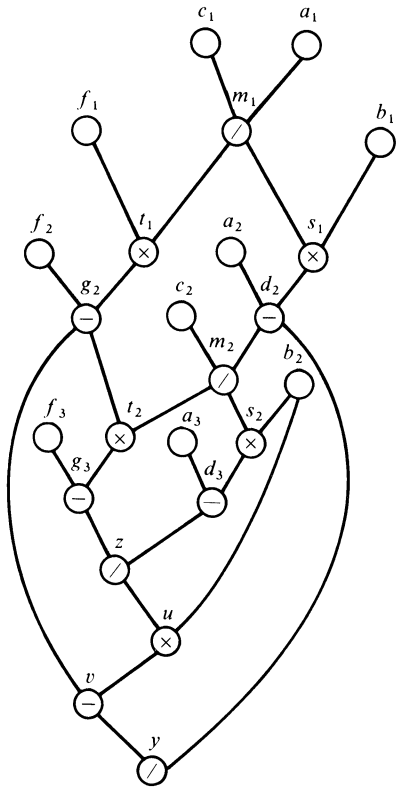

c_1
a_1
m_1
f_1
b_1
f_2
t_1
a_2
s_1
g_2
c_2
d_2
m_2
b_2
f_3
t_2
a_3
s_2
g_3
d_3
z
u
v
y

FIG. 5

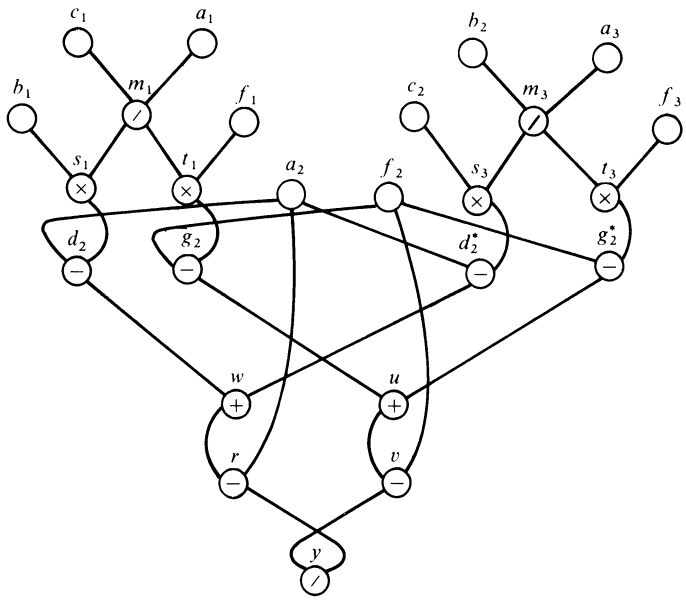

c_1
a_1
b_2
a_3
b_1
m_1
c_2
m_3
f_1
f_3
s_1
t_1
a_2
f_2
s_3
t_3
d_2
g_2
d_2^*
g_2^*
w
u
r
v
y

FIG. 6

contradict results in Babuška [1] and Case Study I of Miller [6]. Thus, not only does the bottom-up heuristic produce an optimal strategy for the GAME in this example, it in some sense gives an optimal strategy among all methods of roundoff analysis.

If the bottom-up heuristic is employed in Fig. 6, then d_2, d_2^*, g_2 and g_2^* remain marked. This represents an optimal strategy for GAME. To see why, consider d_2, d_2^* and w (similar remarks apply to g_2, g_2^* and u). The only Σ-families containing one of these vertices are $\{d_2, d_2^* | w\}$ and $\{s_1, a_2, s_3, w | d_2, d_2^*, r\}$. Thus clearing one of these vertices marks the other two.

However, in this case, the bottom-up heuristic is not an optimal strategy for roundoff analysis. A better job can be done if we use more powerful "moves" than those of the GAME (see § 6).

On certain occasions, the bottom-up heuristic is clearly not appropriate. For example, in Miller [8, § 2] we consider a case where one begins with only the initial vertices marked and tries to move all marks to operation vertices.

However, when the bottom-up heuristic is appropriate, it is easy to apply. In particular, the Σ-chains can be efficiently located by the algorithm given by Knuth [3, pp. 353–354].

4. Graph transformations in automatic roundoff analysis. Elsewhere we have considered the problem of automating proofs of "numerical correctness" of programs. In theory, this goal looks easier than that of automating general program correctness proofs since one can profitably limit attention to verification of decidable (in the technical sense) assertions which can be generated automatically with little, or no, specific understanding of the method being tested. See Miller [5] for further discussion of this point.

In practice, it turns out that while it seems impossible to completely automate roundoff analysis, automatic procedures are sometimes of definite assistance. In particular, the following approach is effective, as is shown in [6]–[8]. To each set **d** of data for a given numerical method we assign a number, say $\omega(\mathbf{d})$, which measures the effect of rounding error at **d**. This is done in such a way that (i) a large value of $\omega(\mathbf{d})$ signals numerical instability at **d**, (ii) $\omega(\mathbf{d})$ is easy to compute and (iii) ω is a continuous function of **d**. A "hill-climbing" routine is then used to search for large values of ω.

The values $\omega(\mathbf{d})$ are found by first numerically evaluating the partial derivatives of the computed values (of the program being tested) with respect to the data and to the rounding errors. Often the bulk of the computational cost lies in evaluating these derivatives. In such cases, the information that certain rounding errors can be ignored leads to a speed-up approaching the factor $(p + q)/(p + r)$ discussed in the next section.

5. Ignoring cleared vertices. The graph GAME of the previous section is directly applicable to the method of automatic roundoff analysis discussed in Miller [6]–[8]. We will now formulate and prove two results which make precise the connection. This section is a digression whose utility can be fully understood only by the reader familiar with [6] and [7].

Consider a computation graph with input vertices $d_1, \cdots, d_p$, operation vertices $v_1, \cdots, v_r$ and output vertices $z_1, \cdots, z_s$. Make this into a floating-point

graph by setting $\rho(v_j) = (1 + \delta_j)$, $\rho(d_i) \equiv 1$, and let

$$R(\mathbf{d}, \boldsymbol{\delta}) = (\mathrm{VAL}(z_1), \cdots, \mathrm{VAL}(z_s)),$$

where $\mathbf{d} = (d_1, \cdots, d_p)$ and $\boldsymbol{\delta} = (\delta_1, \cdots, \delta_r)$. Define $Z(\mathbf{d})$ to be the "exact value", $Z(\mathbf{d}) \equiv R(\mathbf{d}, \mathbf{0})$.

Fix $\mathbf{d}$ such that $Z(\mathbf{d})$ is defined. We sometimes wish to compute the number

$$\omega(\mathbf{d}) = \text{g.l.b.}\,\{\alpha \geqq 0 : \|\boldsymbol{\delta}\|_\infty \leqq u \Rightarrow R(\mathbf{d}, \boldsymbol{\delta}) = Z(\mathbf{d} + \boldsymbol{\pi}), \text{ where } \|\boldsymbol{\pi}\|_d \leqq \alpha u\}. \tag{5.1}$$

Here $\|\boldsymbol{\delta}\|_\infty = \max |\delta_j|$, u is again the machine roundoff level and $\|\cdot\|_d$ is a vector norm depending on $\mathbf{d}$. Informally, the computed value $R(\mathbf{d}, \boldsymbol{\delta})$ is the exact value $Z(\mathbf{d}^*)$, where the data perturbation is $\|\mathbf{d}^* - \mathbf{d}\|_d \leqq \omega(\mathbf{d}) \cdot u$. It is often advantageous to employ the "first-order" approximation to (5.1), namely, to redefine

$$\omega(\mathbf{d}) = \text{g.l.b.}\left\{\alpha \geqq 0 : \|\boldsymbol{\delta}\|_\infty \leqq 1 \Rightarrow \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\delta} = Z'(\mathbf{d})\boldsymbol{\pi}, \text{ where } \|\boldsymbol{\pi}\|_d \leqq \alpha\right\}. \tag{5.2}$$

Suppose that the v_j's in the original computation graph are marked and that the GAME is played. Let $u_1, \cdots, u_q$, $0 \leqq q \leqq r$, be an enumeration of the operation vertices which remain marked. Let us now *consider the case* $q \geqq 1$ (i.e., not all vertices are cleared), deferring the case $q = 0$ to the end of this section. Set $\rho(u_k) = (1 + \eta_k)$, $\rho(v) = 1$ elsewhere, and let

$$Q(\mathbf{d}, \boldsymbol{\eta}) = (\mathrm{VAL}\,(z_1), \cdots, \mathrm{VAL}\,(z_s)),$$

where the VAL's refer to the new floating-point graph.

For an example, return to Figs. 1 and 3. Rename a, b, c, w, x, y, z as $d_1, d_2, d_3, v_1, v_2, v_3, v_4$, respectively, and let $v_4 = z$ be the only output vertex. To get $R(\mathbf{d}, \boldsymbol{\delta})$, we translate (1.1) as

$$R(\mathbf{d}, \boldsymbol{\delta}) = \{d_1[(d_2 + d_3)(1 + \delta_1)](1 + \delta_3) + d_2 d_3(1 + \delta_2)\}(1 + \delta_4).$$

Thinking of the $(1 + \delta_j)$ as marks, we see that two moves clear v_3 and v_4 (Fig. 3). Associate $(1 + \eta_1)$ with v_1 and $(1 + \eta_2)$ with v_2, getting

$$Q(\mathbf{d}, \boldsymbol{\eta}) = d_1[(d_2 + d_3)(1 + \eta_1)] + d_2 d_3(1 + \eta_2).$$

The goal of this section is to compare ω with its analogue which arises when Q is substituted for R. Assuming that $Z(\mathbf{d})$ is defined, we set

$$\omega_Q(\mathbf{d}) = \text{g.l.b.}\left\{\alpha \geqq 0 : \|\boldsymbol{\eta}\|_\infty \leqq 1 \Rightarrow \frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\eta} = Z'(\mathbf{d})\boldsymbol{\pi}, \text{ where } \|\boldsymbol{\pi}\|_d \leqq \alpha\right\}. \tag{5.3}$$

Notice that $\omega(\mathbf{d})$ depends on the $s \times r$ matrix $\partial R/\partial \boldsymbol{\delta}$ and on the $s \times p$ matrix Z', whereas $\omega_Q(\mathbf{d})$ depends on the $s \times q$ matrix $\partial Q/\partial \boldsymbol{\eta}$ and on Z'. Thus, roughly speaking, evaluating ω_Q instead of ω reduces the cost by a factor $(p + q)/(p + r)$.

It is intuitively clear that the sensitivity to rounding errors cannot be increased if we ignore errors corresponding to cleared vertices. This is the thrust of Theorem 5.1.

THEOREM 5.1. *Let* ω *and* ω_Q *be as in* (5.2) *and* (5.3). *Then* $\omega_Q(\mathbf{d}) \leqq \omega(\mathbf{d})$ *whenever* $Z'(\mathbf{d})$ *is defined.*

Proof. Let $j(1), \cdots, j(q)$ be such that $u_k = v_{j(k)}$ for $1 \leqq k \leqq q$. For any $\boldsymbol{\eta} = (\eta_1, \cdots, \eta_q)$ define $\boldsymbol{\delta}(\boldsymbol{\eta}) = (\delta_1(\boldsymbol{\eta}), \cdots, \delta_r(\boldsymbol{\eta}))$ by $\delta_{j(k)}(\boldsymbol{\eta}) = \eta_k$, and $\delta_j(\boldsymbol{\eta}) = 0$ if $j = j(k)$ for no k. Clearly $Q(\mathbf{d}, \boldsymbol{\eta}) \equiv R(d, \boldsymbol{\delta}(\boldsymbol{\eta}))$ since they prescribe identical rounding errors. Differentiating this identity by the chain rule gives

$$\frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0}) = \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0}) \cdot \boldsymbol{\delta}'(\mathbf{0}). \tag{5.4}$$

Now $\boldsymbol{\delta}'(\mathbf{0})$ is the $r \times q$ matrix which has unit $(j(k), k)$-entries and zeros elsewhere. It follows readily that $\|\boldsymbol{\delta}'(\mathbf{0})\boldsymbol{\eta}\|_\infty \leqq \|\boldsymbol{\eta}\|_\infty$ for any $\boldsymbol{\eta}$.

Fix $\mathbf{d}$ such that $Z'(\mathbf{d})$ is defined and let α be such that

$$\|\boldsymbol{\delta}\|_\infty \leqq 1 \Rightarrow \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\delta} = Z'(\mathbf{d})\boldsymbol{\pi}, \quad \text{where } \|\boldsymbol{\pi}\|_d \leqq \alpha.$$

Suppose that $\|\boldsymbol{\eta}\|_\infty \leqq 1$. We need only show that

$$\frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\eta} = Z'(\mathbf{d})\boldsymbol{\pi}, \quad \text{where } \|\boldsymbol{\pi}\|_d \leqq \alpha,$$

since this means just $\alpha > \omega(\mathbf{d}) \Rightarrow \alpha \geqq \omega_Q(\mathbf{d})$, whence $\omega(\mathbf{d}) \geqq \omega_Q(\mathbf{d})$. But by (5.4),

$$\frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\eta} = \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0}) \cdot \boldsymbol{\delta}'(\mathbf{0})\boldsymbol{\eta} = \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\delta},$$

where $\|\boldsymbol{\delta}\|_\infty = \|\boldsymbol{\delta}'(\mathbf{0}) \cdot \boldsymbol{\eta}\|_\infty \leqq \|\boldsymbol{\eta}\|_\infty \leqq 1$. The result follows. □

We will next show that under certain conditions, the uniform boundedness of ω_Q is equivalent to that of ω. This occurs whenever, as is usually the case, the $\boldsymbol{\pi}$ are normed in such a way that small relative errors in the d_i correspond to small $\|\cdot\|_d$. In particular, we usually have

$$\|\operatorname{diag}(d_i) \cdot \boldsymbol{\pi}\|_d \leqq \|\boldsymbol{\pi}\|_\infty. \tag{5.5}$$

Here $\operatorname{diag}(d_i)$ denotes the diagonal matrix containing the d_i. Notice that if $\varphi_i = \pi_i d_i$ with $|\pi_i| \leqq \varepsilon$, $1 \leqq i \leqq p$, then (5.5) implies $\|\varphi\|_d = \|\operatorname{diag}(d_i) \cdot \boldsymbol{\pi}\|_d \leqq \varepsilon$.

THEOREM 5.2. *Let ω and ω_Q be as in* (5.2) *and* (5.3). *Suppose for all* $\mathbf{d}$ *in some set* D *that* $Z(\mathbf{d})$ *is defined and that the norm associated with* $\mathbf{d}$ *satisfies* (5.5). *There exist* β *and* γ *such that*

$$\omega(\mathbf{d}) \leqq \beta \cdot \omega_Q(\mathbf{d}) + \gamma$$

for all $\mathbf{d}$ *in* D.

Proof. It follows immediately from Theorem 2.1, using induction on the number of moves, that there exist numbers $a(i, j)$ and $b(k, j)$, $1 \leqq i \leqq p, 1 \leqq k \leqq q, 1 \leqq j \leqq r$, such that

$$R(\mathbf{d}, \boldsymbol{\delta}) \equiv Q(\mathbf{d}(\boldsymbol{\delta}), \boldsymbol{\eta}(\boldsymbol{\delta})), \tag{5.6}$$

where $\mathbf{d}(\boldsymbol{\delta}) = (d_1(\boldsymbol{\delta}), \cdots, d_p(\boldsymbol{\delta}))$ and $\boldsymbol{\eta}(\boldsymbol{\delta}) = (\eta_1(\boldsymbol{\delta}), \cdots, \eta_q(\boldsymbol{\delta}))$ satisfy

$$d_i(\boldsymbol{\delta}) = d_i \cdot \prod_{j=1}^{r} (1 + \delta_j)^{a(i,j)}, \qquad 1 + \eta_k(\boldsymbol{\delta}) \equiv \prod_{j=1}^{r} (1 + \delta_j)^{b(k,j)}.$$

Let $[a]$ and $[b]$ denote the matrices of $a(i, j)$ and $b(k, j)$. Thus if we again use Fig. 3 as an example, we have

$$[a] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad [b] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

We will show that the theorem holds with

$$\gamma = \max_i \left(\sum_{j=1}^{r} |a(i,j)| \right), \qquad \beta = \max_k \left(\sum_{j=1}^{r} |b(k,j)| \right). \tag{5.7}$$

The inequalities $\|[a] \cdot \boldsymbol{\delta}\|_\infty \leqq \gamma \cdot \|\boldsymbol{\delta}\|_\infty$ and $\|[b] \cdot \boldsymbol{\delta}\|_\infty \leqq \beta \cdot \|\boldsymbol{\delta}\|_\infty$ are well known (e.g., Stewart [9, p. 179]).

Applying the chain rule to (5.6), we get

$$\begin{aligned} \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0}) &= \frac{\partial Q}{\partial \mathbf{d}}(\mathbf{d}, \mathbf{0}) \cdot \mathbf{d}'(\mathbf{0}) + \frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0}) \cdot \boldsymbol{\eta}'(\mathbf{0}) \\ &= Z'(\mathbf{d}) \cdot (\operatorname{diag}(d_i) \cdot [a]) + \frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0}) \cdot [b], \end{aligned} \tag{5.8}$$

using the easy identities $(\partial Q/\partial \mathbf{d})(\mathbf{d}, \mathbf{0}) = Z'(\mathbf{d})$, $\mathbf{d}'(\mathbf{0}) = \operatorname{diag}(d_i) \cdot [a]$ and $\boldsymbol{\eta}'(\mathbf{0}) = [b]$.

Fix $\mathbf{d}$ in D and let α be such that

$$\|\boldsymbol{\eta}\|_\infty \leqq 1 \Rightarrow \frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\eta} = Z'(\mathbf{d})\boldsymbol{\pi}, \quad \text{where } \|\boldsymbol{\pi}\|_d \leqq \alpha.$$

Suppose that $\|\boldsymbol{\delta}\|_\infty \leqq 1$. We need only show that

$$\frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\delta} = Z'(\mathbf{d})\boldsymbol{\pi}, \quad \text{where } \|\boldsymbol{\pi}\|_d \leqq \beta\alpha + \gamma,$$

since this means just that $\alpha > \omega_Q(\mathbf{d}) \Rightarrow \beta\alpha + \gamma \geqq \omega(\mathbf{d})$, whence $\omega(\mathbf{d}) \leqq \beta \cdot \omega_Q(\mathbf{d}) + \gamma$. Using (5.8), we have

$$\begin{aligned} \frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\delta} &= Z'(\mathbf{d})\{\operatorname{diag}(d_i) \cdot [a] \cdot \boldsymbol{\delta}\} + \frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\{[b] \cdot \boldsymbol{\delta}\} \\ &= Z'(d)\mathbf{x} + \frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\mathbf{y}, \end{aligned}$$

where $\|\mathbf{x}\|_d = \|\operatorname{diag}(d_i) \cdot [a] \cdot \boldsymbol{\delta}\|_d \leqq \|[a] \cdot \boldsymbol{\delta}\|_\infty \leqq \gamma \cdot \|\boldsymbol{\delta}\|_\infty \leqq \gamma$ and $\|\mathbf{y}\|_\infty \leqq \|[b] \cdot \boldsymbol{\delta}\|_\infty \leqq \beta$. By our choice of α and the linearity of $\partial Q/\partial \boldsymbol{\eta}$, we have

$$\frac{\partial Q}{\partial \boldsymbol{\eta}}(\mathbf{d}, \mathbf{0})\mathbf{y} = Z'(\mathbf{d})\mathbf{z}, \quad \text{where } \|\mathbf{z}\|_d \leqq \beta\alpha.$$

Setting $\boldsymbol{\pi} = \mathbf{x} + \mathbf{z}$, we have

$$\frac{\partial R}{\partial \boldsymbol{\delta}}(\mathbf{d}, \mathbf{0})\boldsymbol{\delta} = Z'(\mathbf{d})\boldsymbol{\pi}, \quad \text{where } \|\boldsymbol{\pi}\|_d \leqq \|\mathbf{x}\|_d + \|\mathbf{z}\|_d \leqq \beta\alpha + \gamma. \qquad \square$$

The situation when $q = 0$, i.e., when all vertices are cleared, is simpler. For norms satisfying (5.5) and for γ defined as in (5.7), one can prove the bound $\omega(\mathbf{d}) \leqq \gamma$ by appropriate editing of the proof of Theorem 5.2.

6. Open questions and extensions. Here are some obvious problems concerning the GAME. Give an algorithm which produces an optimal sequence of moves (i.e., a sequence which clears as many vertices as possible) in time bounded by a polynomial in the size of the graph. Otherwise, show that no such algorithm exists. If none exists, then find and verify a polynomial-time strategy which always (or at least often) reduces the number of marks to near minimum.

To a certain extent these problems, while interesting, are not the real issues raised by the results of this paper. Reducing the number of marked vertices as far as can be done using only the moves of the GAME in no way guarantees that further reductions cannot be easily achieved by other means. In particular, it is sometimes possible to perform simple analyses on chunks of the graph larger than Σ-families.

For example, consider Fig. 6. The results of Miller [5] can be interpreted as showing that marks below s_1, t_1, a_2, f_2, s_3 and t_3 can be simultaneously moved up to $s_1, \cdots, t_3$. Then the moves of the GAME can be applied to clear all remaining marks. (Actually, the moves implied by [5] can clear *all* vertices in a single application.)

Examples covered by neither this paper nor [5] are easy to come by. For instance, it is not hard to see that the common operation of normalizing a vector $\mathbf{x}$, i.e., producing $(\sum x_i^2)^{-1/2} \cdot \mathbf{x}$, gives values with small relative errors. In other words, the computed vector can be gotten by using error-free operations if a few rounding errors are injected into the results. When applied to the 3×3 version of the Gram–Schmidt method (Case Study III of Miller [6]) this rule immediately shows that essentially no generality is lost if only 36 of the 60 rounding errors are considered.

Let us close on a more positive note by mentioning an extension of the results of this paper which is sometimes useful. Suppose that we may neglect rounding errors which are equivalent to a few rounding errors in the data and/or *in the computed values.* In terms of the GAME, this essentially means that we can extend (2) by ignoring marks on output vertices and that we can drop constraint (i) requiring S to be free of output vertices. The results of § 5 admit of a straightforward extension to this case.

The extended game can be applied to the graph for the 4×4 instance of the usual algorithm for triangular matrix inversion (making modifications to allow the unary operations of minus and inversion). One fairly natural extension of the "bottom-up heuristic" clears all but one node. This is optimal among *all* methods of analysis since this program is known to suffer from a certain form of instability (see the case study in Miller [7]).

Acknowledgment. We benefited from conversations with P. Downey, D. B. Johnson, E. L. Robertson and D. R. Stoutemyer and from the referees' comments.

REFERENCES

[1] I. BABUŠKA, *Numerical stability in problems of linear algebra*, SIAM J. Numer. Anal., 9 (1972), pp. 53–77.
[2] F. BAUER, *Computational graphs and rounding error*, SIAM J. Numer. Anal., 11 (1974), pp. 87–96.
[3] D. KNUTH, *The Art of Computer Programming*, vol. I. Addison-Wesley, Reading, Mass., 1968.
[4] D. MCCRACKEN AND W. DORN, *Numerical Methods and FORTRAN Programming*, John Wiley, New York, 1964.
[5] W. MILLER, *Remarks on the complexity of roundoff analysis*, Computing, 12 (1974), pp. 149–161.
[6] ———, *Software for roundoff analysis*, ACM Trans. Math. Software, 1 (1975), pp. 108–128.
[7] ———, *Computer search for numerical instability*, J. Assoc. Comput. Mach., 22 (1975), pp. 512–521.
[8] ———, *Roundoff analysis by direct comparison of two algorithms*, SIAM J. Numer. Anal., 13 (1976), pp. 382–392.
[9] G. STEWART, *Introduction to Matrix Computations*, Academic Press, New York, 1973.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

ON FAMILIES OF LANGUAGES DEFINED BY TIME-BOUNDED RANDOM ACCESS MACHINES*

I. H. SUDBOROUGH AND A. ZALCBERG†

Abstract. There are essentially two results described in this paper. First, it is shown that for any random access machine (RAM) time-constructable function $T(n) \geqq n$, there are languages L such that L can be recognized in time $O(T(n))$ by a RAM (using the unit cost measure), but L cannot be recognized by any deterministic multitape Turing machine in time $O(T(n))$.

Secondly, a family of random access stored program machines (RASP's) are considered. For these RASP's it is shown that there is an arbitrarily complex (infinitely often) partial recursive function $f(n)$ which has only 0–1 values (whenever defined) such that $f(n)$ can be computed in time $F(n)$ by some RASP, but cannot be computed in time $(1 - \varepsilon)F(n)$, for any $\varepsilon > 0$, by any RASP in this family.

Key words. random access machine, Turing machine, time bounds

1. Introduction. The results described in this paper concern families of languages recognized by random access machines (RAM's), random access stored program machines (RASP's), and deterministic multitape Turing machines operating under certain time-bounded restrictions. Time-bounded RAM's have been considered previously by Cook [1] and Cook and Reckhow [2]. Time-bounded RASP's were considered previously by Hartmanis in [3]. The multitape Turing machines we consider are the standard model as defined, for example, by Hopcroft and Ullman in [4].

In this section we review the definition of a random access machine as given in [2]. Cook and Reckhow considered different classes of RAM's by defining a cost function $l(n)$ which, roughly speaking, denoted the time required to store the number n. We shall consider RAM's in this paper in which the cost function $l(n)$ is identically one; in fact, we assume that each instruction takes one unit of execution time. Thus, throughout the paper, RAM means the unit cost RAM of [2]. Cook in [1] argues persuasively that the cost function $l(n)$ should be $\log(n)$, not $l(n) = 1$, in order to more accurately measure the inherent complexity of a problem. However, a unit cost RAM is often considered in the literature; for example, in [1] a linear time algorithm for two-way deterministic pushdown automata is described with $l(n) = 1$.

In § 2 we show that any $T(n)$-time-bounded k-tape deterministic Turing machine can be simulated by a (unit cost) RAM within time $c(k)T(n)$, where the constant $c(k)$ depends only upon k (the number of tapes). The best previous result on the amount of time for a RAM to simulate a multitape Turing machine required a constant which depended upon both the number of tapes and the number of tape symbols. Using this result and a "diagonalization" argument, we show that for "linearly honest" time bounds $T(n)$ there are languages recognized in time $O(T(n))$ by random access machines which cannot be recognized by

* Received by the editors August 22, 1974, and in revised form February 4, 1975. This work was supported in part by the National Science Foundation under Grants GU-3851 and GJ-43228.

† Department of Computer Sciences, Technological Institute, Northwestern University, Evanston, Illinois 60201.

any multitape Turing machine in time $O(T(n))$.[1] Thus a linear time simulation of a RAM by a multitape Turing machine is not possible in these cases.

In § 3 we show that there is a natural family of random access stored program machines (RASP's) such that arbitrarily complex (infinitely often) partial recursive functions which have only 0 or 1 values (whenever defined) can be described which can be computed in time $\Phi(n)$ but not in time $(1 - \varepsilon)\Phi(n)$ for any $\varepsilon \geqq 0$. Hartmanis in [3] considered other families of RASP's for which there were arbitrarily complex total recursive functions with optimal algorithms, but these functions were not 0–1 valued. In fact, it was shown that these functions had optimal algorithms by an argument based on the rate at which the size of the register values could be increased. An open question, first raised in [3], is whether or not there are arbitrarily complex 0–1 valued functions with optimal algorithms (in the sense indicated above) on any RASP model. We are able to show the existence of a partial recursive function with this property; our function is essentially a universal function and, therefore, is not defined on all input values. The method is to show that if the universal function could be speeded up by some constant factor, then all functions could be speeded up by some smaller factor. However, this latter statement contradicts a straightforward result obtained by "diagonalization."

We shall now review the definition of a RAM given in [2]. A random access machine consists of a finite program operating on an infinite sequence of registers. Each register can hold an arbitrary integer (positive, negative, or zero). The contents of the registers are denoted by $X_0, X_1, X_2, \cdots$.

The types of possible instructions are given in Table 1. The effect of most of the instructions should be evident. For example, $X_i \leftarrow C$ means that register i takes on the value C. The instruction TRA m if $X_j > 0$ means that the RAM transfers control to the mth instruction of the program if $X_j > 0$; otherwise, the RAM executes the next instruction in its usual manner.

A RAM is defined to accept a set L over a finite alphabet $\Sigma = \{a_1, a_2, \cdots, a_p\}$. An input string $W = a_{i_1}a_{i_2} \cdots a_{i_n}$ is given to the RAM as a sequence of integers $i_1, i_2, \cdots, i_n, 0$, where 0 indicates the end of the string. The input is provided with a pointer which initially points to the first input integer. Every time the instruction READ X_i is executed, the number currently pointed to is stored in register X_i and the pointer is advanced. A RAM M accepts the string $w = a_{i_1}a_{i_2} \cdots a_{i_n}$ if it eventually executes an ACCEPT instruction (in a computation with input $i_1, i_2, \cdots, i_n, 0$); otherwise, M rejects w.

The indirectly addressed instruction $X_i \leftarrow X_{x_j}$ means that register i takes on the value of the content of the X_jth register. That is, the content of X_j is used to define the address of the register to be copied into register i. If the content of a register used in an indirect address is negative, then the RAM halts. The instruction $X_{x_i} \leftarrow X_j$ has an analogous meaning. The indirectly addressed instructions are necessary in order for a fixed program to access an unbounded number of registers.

A RAM program is started at the first instruction, with all registers initially zero, and it halts when a transfer is made to a nonexistent instruction, when a negative indirect address is encountered, or when an ACCEPT instruction is

[1] A function $g(n)$ is said to be $O(f(n))$ if there is some constant c such that $g(n) \leqq c\, f(n)$ for all but some finite (possibly empty) set of nonnegative values for n.

TABLE 1
RAM instructions

$X_i \leftarrow C$, C any integer
$X_i \leftarrow X_j + X_k$
$X_i \leftarrow X_j - X_k$
$X_i \leftarrow X_{x_j}$
$X_{x_i} \leftarrow X_j$
TRA m if $X_j > 0$
READ X_i
ACCEPT

encountered. The computation time of a RAM (in this case, with $l(n) = 1$) is the number of instructions executed during a computation. A RAM M recognizes a set L of strings within time $T(n)$ if (i) for all but finitely many strings w, M halts on w within $T(|w|)$ steps, and (ii) M accepts w if and only if w is in L.

We shall need, also, to define the notion of a function which can be computed in an amount of time bounded by a linear multiple of its functional values. A function $T(n)$ on the positive integers is RAM *time-constructable* if there is some RAM program P such that, for all n, if P starts with n in one of its registers, P halts in $O(T(n))$ steps with $T(n)$ in some register. This notion of RAM time-constructable is essentially the same as that defined by Cook and Reckhow [2]. We shall also define a similar notion of time-constructable functions computed by random access stored program machines in § 3.

2. Time-bounded random access machines. Cook in [1] showed that every set recognized by a deterministic k-head two-way pushdown automaton can be recognized by a random access machine in time $O(n^k)$. It is not known, however, whether a multitape Turing machine can also recognize these k-head pda languages in time $O(n^k)$. The best known previous result relating time-bounded random access machines and time-bounded Turing machines was given by Cook and Reckhow in [2].

THEOREM 2.1 (Cook and Reckhow). *If a set A is recognized by a* RAM *within time $O(T(n))$, then A is recognized by some multitape Turing machine within time $(T(n))^3$. Conversely, if some multitape Turing machine recognizes a set A within time $T(n)$, then some* RAM *recognizes A within time $O(T(n))$.*

It is not known whether or not these results are optimal. It is possible, for example, that a multitape Turing machine could simulate a $T(n)$-time-bounded RAM in time $T(n)\log(T(n))$. We show here, however, that a linear time simulation is not possible. That is, for a large class of functions (the RAM time-constructable functions), there are sets recognized in time $O(T(n))$ by random access machines which cannot be recognized in time $O(T(n))$ by any multitape Turing machine.

This result is obtained by describing a revised version of the Cook–Reckhow simulation of a multitape Turing machine by a random access machine. The straightforward method involves determining which symbols the simulated Turing machine is scanning inside the fixed program of the RAM. Thus the program size of the RAM and the number of steps used to simulate one step of the Turing machine is dependent upon the number of symbols in the Turing machine's

tape alphabet. We avoid this problem by storing a representation of the Turing machine transition table inside the memory of the random access machine. The particular representation that we use enables the random access machine to simulate one step of the Turing machine computation in a number of steps which depends only upon the number of tapes the Turing machine possesses. Of course, additional time must be spent at the beginning of the RAM computation in setting up this representation of the Turing machine transition table, but the number of steps to do this is independent of the length of the input. The initialization phase consists of creating the representation of the transition table, giving initial values to a few registers (to be described), and setting up in the remaining registers a representation of the contents of each of the Turing machine tapes. The simulation phase consists of altering the contents of the registers to denote the successive configurations of the Turing machine. These comments are formalized in the proof of the following theorem.

THEOREM 2.2. *Let $T(n) \geqq n$. For $k \geqq 1$, there exists a constant $c(k)$ such that if $L \subseteq \Sigma^*$ is recognized by a deterministic k-tape Turing machine in time $T(n)$, then L is recognized by a* RAM *in time $c(k)T(n)$.*

Proof. Let $L \subseteq \Sigma^*$ and let Z be a k-tape Turing machine which recognizes L in time $T(n)$. Let Z have s states $\{1, 2, \cdots, s\}$ and t tape symbols $\{1, 2, \cdots, t\}$. Let $m = \max\{2, s, t\}$. Construct a random access machine M_Z which performs the following steps on any input $w \in \Sigma^*$.

Phase 1. M_Z constructs a representation of the transition table of Z in the first $(2k + 2)m^{k+1}$ registers of its memory. Information concerning the transition of Z when its current state is j and the scanned symbol on tape $1, 2, \cdots, k$ is $i_1, i_2, \cdots, i_k$, respectively, will be in consecutive registers beginning with register $(jm^k + i_1m^{k-1} + i_2m^{k-2} + \cdots + i_k)(2k + 2)$. The information will be in $2k + 2$ consecutive registers. If Z in state j scanning symbols $i_1, i_2, \cdots, i_k$ moves to state j', writes the symbols $i'_1, i'_2, \cdots, i'_k$, and moves heads $1, 2, \cdots, k$ to the right $\lambda_1, \lambda_2, \cdots, \lambda_k$ squares, respectively, then the contents of the $2k + 2$ registers in this block would be as pictured in Table 2.

TABLE 2

Block of $2k + 2$ registers representing one transition of Turing machine. $f(j') = 1$, if j' is a final state; otherwise, $f(j') = 0$.

Register	Contents
1	$j'(2k + 2)m^k$
2	$i'_1(2k + 2)m^{k-1}$
3	$i'_2(2k + 2)m^{k-2}$
$\cdots$	$\cdots$
$k + 1$	$i'_k(2k + 2)m^0$
$k + 2$	$\lambda_1 k$
$k + 3$	$\lambda_2 k$
$\cdots$	$\cdots$
$2k + 1$	$\lambda_k k$
$2k + 2$	$f(j')$

After creating the representation of the transition table of the Turing machine Z, M_Z initializes the contents of the next block of $k + 4$ registers. The first register in this block, called POINT, will be assigned the value $(jm^k + i_1m^{k-1} + i_2m^{k-2} + \cdots + i_k)(2k + 2)$ if the Turing machine Z begins in state j with the contents of the scanned square on tape $1, 2, \cdots, k$ being $i_1, i_2, \cdots, i_k$, respectively. In general, the contents of POINT will be the address of one of the registers which describe the next instruction of Z to be simulated. The next k registers, called POINT(1), POINT(2), $\cdots$, POINT(k), are initialized to contain the address of the register containing the information aout the scanned square of tape $1, 2, \cdots, k$, respectively. The information concerning the contents of each tape will be located immediately after the current $k + 4$ registers. For example, if d is the address of the last register used so far, then the contents of cell $1, 2, 3, \cdots$ of tape i will be in registers $d + i$, $d + k + i$, $d + 2k + i, \cdots$. Thus POINT(i) will be initialized to $d + i$. The last three registers in this block, called TOTAL, TEMP and ONE, are initialized to 0, 0 and 1, respectively. The content of TOTAL is used to calculate the address of the first of the block of registers describing the next Turing machine instruction to be simulated. The content of TEMP is used for temporary storage during the computation. The content of ONE will always be one and is used to allow addition by one.

The last stage of Phase 1 is to initialize the contents of the remaining registers to contain a representation of the k tapes of the Turing machine Z. As stated before, the contents of cells $1, 2, 3, \cdots$ of tape i will be represented in registers $d + i$, $d + k + i$, $d + 2k + i, \cdots$ (where d = the address of the last register used for previous quantities). If tape i contains the string of symbols $i_0i_1 \cdots i_p$, then registers $d + i$, $d + k + i$, $d + 2k + i, \cdots, d + pk + i$ will contain the quantities $i_0(2k + 2)m^{k-i}$, $i_1(2k + 2)m^{k-i}, \cdots, i_p(2k + 2)m^{k-i}$. Since initially only the first tape, containing the input string, will have nonblank symbols, this stage of Phase 1 can be completed by reading the input tape. All the other registers will contain zero, which denotes the blank symbol; this is so, since tapes of a Turing machine not used for the input are assumed to be blank.

Phase 2. In this phase, the RAM M_Z simulates the computation of the Turing machine Z on the input string w. The program to accomplish this simulation is contained in the Appendix. An outline of the basic steps is provided below:

(a) M_Z accesses the first register in the block describing the current transition of Z; see Table 2 above. The contents of this register (a value describing the next state of Z) is inserted into the register called TOTAL (used to accumulate the address of the first register of the block describing the next transition).

(b) M_Z accesses in sequence the next k registers in the current block; see Table 2 above (these registers describe the symbols printed by Z on its k tapes during this transition). M_Z stores these values into the appropriate registers currently "pointed to" by POINT(1), POINT(2), $\cdots$, POINT(k). This enables the registers of M_Z to accurately portray the new tape contents of Z.

(c) M_Z accesses in sequence the next k registers in the current block; see Table 2 above (these registers describe the head motions during the current transition). M_Z changes the contents of POINT(1), POINT(2), $\cdots$, POINT(k) as specified to accurately represent the new positions of the k heads of Z.

(d) M_Z adds to the register TOTAL the value currently in the registers pointed

to by POINT(1), POINT(2), $\cdots$, POINT(k). This will assign to TOTAL the address of the first register in the block describing the next transition of Z. (This follows since the current symbols scanned on tapes 1, 2, $\cdots$, k are represented by the values in the registers pointed to by POINT(1), POINT(2), $\cdots$, POINT(k).)

(e) M_Z checks the last register in the current block in order to determine whether Z accepts or not. If Z accepts, then M_Z accepts; otherwise, M_Z simulates the next transition of Z by repeating steps (a) through (e).

It should be clear from this description that M_Z accepts w if and only if Z accepts w. Thus Z and M_Z accept the same language.

The RAM M_Z requires some constant c_0 steps to set up the representation of the Turing machine Z's transition table and the $k + 4$ additional register values, i.e., POINT, POINT(i), etc. It requires an amount of time proportional to the length of the input tape to set up the register representation of the k tapes, say $c_1 n$ steps, where n is the length of the input. Finally, there is some constant c_2 such that it takes $c_2 k$ steps to simulate each step of the k-tape Turing machine Z. Since Z operates in time $T(n)$, M_Z operates in time $T'(n) = c_0 + c_1 n + c_2 k T(n)$. Since $T(n) \geqq n$, there is a constant $c(k)$ such that $T'(n) \leqq c(k)T(n)$, for all but finitely many n. □

A close inspection of the proof of Theorem 3 in [2, p. 363] allows one to obtain the following result for (unit cost) RAM's (by "diagonalization").

THEOREM 2.3. *For every* RAM *time-constructable function* $T(n)$, *where* $T(n) \geqq n$, *there is a constant* $c > 1$ *and a language* $L \subseteq \{0, 1\}^*$ *such that*

(i) L *is recognized by a* RAM *in time* $cT(n)$; *and*

(ii) L *is not recognized by any* RAM *operating in time* $T(n)$.

The following result then follows from Theorems 2.2 and Theorem 2.3.

THEOREM 2.4. *For every* RAM *time-constructable function* $T(n)$ *such that* $\liminf_{n\to\infty} T(n)/n = \infty$, *there is a language* $L \subseteq \{0, 1\}^*$ *such that*

(i) L *can be recognized by a* RAM *in time* $O(T(n))$; *and*

(ii) L *cannot be recognized by any deterministic multitape Turing machine in time* $O(T(n))$.

Proof. Let $T(n)$ be time-constructable. By Theorem 2.3 there is a constant $c > 1$ and a language $L \subseteq \{0, 1\}^*$ such that L is recognized in time $cT(n)$ by some RAM, but L is not recognized in time $T(n)$ by any RAM.

Suppose L were recognized in time $c_0 T(n)$, for some constant c_0, by a k-tape Turing machine Z_0. By the classical speed-up theorem for multitape Turing machines, it follows that there is a k-tape Turing machine Z_1 that recognizes L in time[2] $T_1(n) = \lfloor T(n)/c_1 \rfloor$, where c_1 is some constant greater than or equal to $c(k)$, ($c(k)$ is the constant given by Theorem 2.2). Theorem 2.2 then states that L is recognized in time $T(n)$ by some RAM, contrary to our assumption. □

The reader should note that the proof as stated of Theorem 2.4 is not sufficient to show that there are languages recognized in linear time by RAM's which cannot be recognized in linear time by multitape Turing machines. Basically the problem is that, for any constant c, if the time bound is $T(n) = cn$, then $c(k)$ will be larger than c for some value of k and a Turing machine cannot recognize (nontrivial) languages in time $T(n)/c(k)$, for this value of k. However, one can achieve the linear time result by noticing that, for all k, the constant $c(k)$ in Theorem 2.2 may be

[2] $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

made smaller than some fixed constant c_0.[3] This may be done by compressing the size of the input tape when the simulating RAM stores the tape's representation initially into certain registers. In this way the new RAM may perform many steps of the Turing machine in the same amount of time as the RAM in the proof of Theorem 2.2 performed one step. (This is the same basic construction as in the classic speed-up result for multitape Turing machines.) Thus there are languages recognized in linear time by RAM's which cannot be recognized by any deterministic multitape Turing machine in linear time.

The authors are unable to extend these results to RAM's with arbitrary cost functions $l(n)$, such as the perhaps more natural $l(n) = \log(n)$ discussed in [2]. We have also not been able to describe a more efficient simulation of a RAM by a multitape Turing machine than that previously described in Theorem 2.1.

3. Random access stored program machines. Classes of functions computed by time-bounded random access stored program machines were originally investigated by Hartmanis in [3]. Hartmanis described some arbitrarily complex functions which have optimal algorithms (with respect to execution time) on some natural RASP models. These functions were such that their values grew rapidly and could be shown to have optimal algorithms based on the rate at which register values could be increased on the RASP models considered. Hartmanis then asked the question: are there arbitrarily complex 0–1 valued functions which have optimal algorithms (in the same sense)?

We shall describe in this section a family of RASP machines (called RASP3's) and a partial recursive function $g(n)$ which can be computed by a RASP3 program in an amount of time given by the partial recursive function $G(n)$. However, for any $\varepsilon > 0$, $g(n)$ cannot be computed in time $(1 - \varepsilon)G(n)$. This partial recursive function is essentially a universal function. It cannot be speeded up by any constant factor, since such a speed-up would imply (as we shall demonstrate) that all RASP3 programs could be speeded up by a similar factor. That not all RASP3 programs can be speeded up by a fixed constant factor can easily be shown by "diagonalization".

The function $g(n)$ is constructed so that its value (whenever defined) is either 0 or 1. This partial recursive function, therefore, has an optimal RASP3 program (in the sense indicated) for reasons unrelated to its rate of growth. The authors are unable, however, to show that there are total recursive 0–1 valued functions which are arbitrarily complex and have optimal algorithms on this (or any other general) RASP model.

We shall give a brief review of the definition of RASP machines. The reader is referred to [3] for further details. A random access stored program machine consists of a memory M and finite set of instructions I. The memory M of a RASP will consist of a finite set of special registers, such as the *accumulator* (AC) and the *instruction counter* (IC), and an infinite sequence of auxiliary memory registers R1, R2, R3, $\cdots$. Each register is capable of holding an arbitrary nonnegative integer. The content of the registers will be denoted by $\langle \mathrm{AC} \rangle$, $\langle \mathrm{IC} \rangle$, $\langle \mathrm{R}n \rangle$ or $\langle n \rangle$, and (in general) $\langle P \rangle$ for a register P. We shall extend the number of special registers

[3] The authors are indebted to one of the referees for this observation and a correction of an earlier version of Theorem 2.4.

from the two indicated above (the only ones defined in [3]) to include a *base register* (BR) and a *remainder register* (RR).

The instructions of a RASP program will be encoded as nonnegative integers and stored in consecutive registers of memory beginning with register R1. If there are less than m instruction types in a particular RASP model, then each instruction will be assigned a machine code (c_1, c_2, c_3), where c_1 is the integer operand, $c_2 = 0, 1$ or 2, depending upon whether the operand is a constant, an address of a register or an indirect address, and $1 \leqq c_3 < m$ describes which instruction type is coded. Each machine code (c_1, c_2, c_3) will then be assigned the numeric code $c_1m^2 + c_2m + c_3$.

A program will consist of a finite sequence of instructions and initial data values stored in memory. We assume that at the start of a computation $\langle \text{IC} \rangle = 1$ and that the content of all memory registers not specified by the program is zero. The input to the RASP is the integer placed in AC at the start of the computation and the result (if defined) is either 0 or 1, depending upon whether the RASP executes a HALT-0 or HALT-1 instruction, respectively. The program is executed by performing the instructions in the order indicated by the instruction counter starting with $\langle \text{IC} \rangle = 1$ until either a HALT-0 or HALT-1 instruction is encountered.

Hartmanis in [3] considered many classes of RASP's by specifying various sets of instructions. We shall specify a set of instructions for a class of RASP's called RASP3. The base register (BR), which we have added to the set of special registers, is used in RASP3 to offset the addresses given in each RASP3 instruction. The content of the BR is added to each operand address to obtain the actual location specified. In this respect it is similar to the index registers present in many present-day computers. This base register was added in order to allow a program to be shifted in memory without changing its running time. We are unable, however, to show that the addition of this special register adds any additional computing power in time-bounded computation. Since our RASP3 machines will have an integer divide instruction, we have provided for another special register, called the remainder register (RR), to contain the remainder after a division operation. The RASP3 instructions are of the following types:

	Name	*Code*	*Meaning*[4]
(1)	TRA, n TRA, $\langle n \rangle$	$(n, 1, 1)$ $(n, 2, 1)$	Transfer control to register $\text{R}(n + \langle \text{BR} \rangle)$ and $\text{R}\langle n + \langle \text{BR} \rangle \rangle$, respectively.
(2)	TRZ, n TRZ, $\langle n \rangle$	$(n, 1, 2)$ $(n, 2, 2)$	If $\langle \text{AC} \rangle = 0$, then transfer control to register $\text{R}(n + \langle \text{BR} \rangle)$ and $\text{R}\langle n + \langle \text{BR} \rangle \rangle$, respectively. If $\langle \text{AC} \rangle \neq 0$, then continue to next instruction.
(3)	STO, n STO, $\langle n \rangle$	$(n, 1, 3)$ $(n, 2, 3)$	Store $\langle \text{AC} \rangle$ in $\text{R}(n + \langle \text{BR} \rangle)$ and $\text{R}\langle n + \langle \text{BR} \rangle \rangle$, respectively.
(4)	CLA, n CLA, $\langle n \rangle$ CLA, $\langle\langle n \rangle\rangle$	$(n, 0, 4)$ $(n, 1, 4)$ $(n, 2, 4)$	n, $\langle n + \langle \text{BR} \rangle \rangle$, $\langle\langle n + \langle \text{BR} \rangle \rangle\rangle$, respectively, is stored in AC.

[4] If $x \geqq y$, then $x \dot{-} y = x - y$; otherwise, $x \dot{-} y = 0$.

(5)	ADD, n ADD, $\langle n \rangle$ ADD, $\langle\langle n \rangle\rangle$	$(n, 0, 5)$ $(n, 1, 5)$ $(n, 2, 5)$	$\langle AC \rangle$ is replaced by $\langle AC \rangle + n$, $\langle AC \rangle + \langle n + \langle BR \rangle\rangle$, and $\langle AC \rangle + \langle\langle n + \langle BR \rangle\rangle\rangle$, respectively.
(6)	SUB, n SUB, $\langle n \rangle$ SUB, $\langle\langle n \rangle\rangle$	$(n, 0, 6)$ $(n, 1, 6)$ $(n, 2, 6)$	$\langle AC \rangle$ is replaced by $\langle AC \rangle \dot{-} n$, $\langle AC \rangle \dot{-} \langle n + \langle BR \rangle\rangle$, and $\langle AC \rangle \dot{-} \langle\langle n + \langle BR \rangle\rangle\rangle$, respectively.
(7)	MLT, n MLT, $\langle n \rangle$ MLT, $\langle\langle n \rangle\rangle$	$(n, 0, 7)$ $(n, 1, 7)$ $(n, 2, 7)$	$\langle AC \rangle$ is replaced by $\langle AC \rangle \times n$, $\langle AC \rangle \times \langle n + \langle BR \rangle\rangle$, $\langle AC \rangle \times \langle\langle n + \langle BR \rangle\rangle\rangle$, respectively.
(8)	DIV, n DIV, $\langle n \rangle$ DIV, $\langle\langle n \rangle\rangle$	$(n, 0, 8)$ $(n, 1, 8)$ $(n, 2, 8)$	$\langle AC \rangle$ is replaced by $\lfloor \langle AC \rangle \div n \rfloor$, $\lfloor \langle AC \rangle \div \langle n + \langle BR \rangle\rangle \rfloor$, and $\lfloor \langle AC \rangle \div \langle\langle n + \langle BR \rangle\rangle\rangle \rfloor$, respectively, and $\langle RR \rangle$ is replaced by $\langle AC \rangle - \lfloor \langle AC \rangle \div n \rfloor \times n$, $\langle AC \rangle - \lfloor \langle AC \rangle \div \langle n + \langle BR \rangle\rangle \rfloor \times \langle n + \langle BR \rangle\rangle$, $\langle AC \rangle - \lfloor \langle AC \rangle \div \langle\langle n + \langle BR \rangle\rangle\rangle \rfloor \times \langle\langle n + \langle BR \rangle\rangle\rangle$, respectively.
(9)	STOR, n STOR, $\langle n \rangle$	$(n, 1, 9)$ $(n, 2, 9)$	Store $\langle RR \rangle$ in $R(n + \langle BR \rangle)$ and $R\langle n + \langle BR \rangle\rangle$, respectively.
(10)	INCB, n	$(n, 0, 10)$	$\langle BR \rangle$ is replaced by $\langle BR \rangle + n$.
(11)	HALT-0 HALT-1	$(0, 0, 11)$ $(1, 0, 11)$	Halt with output of 0 and 1, respectively

Hartmanis gave the name RASP1 to the class of RASP machines with the instruction types (1)–(7) indicated in the previous list (without the use of a base register and with output of any integer via the AC at the end of a computation). Hartmanis proved the following theorem in [3].

THEOREM 3.1. *There exist arbitrarily complex functions $f_i(n)$ which can be computed in time $T_i(n)$ on* RASP1 *but cannot be computed in time $(1 - \varepsilon)T_i(n)$ on any* RASP1, *for any $\varepsilon > 0$.*

As was indicated earlier, the functions $f_1, f_2, f_3, \cdots$ (indicated in Theorem 3.1) are fast growing functions; whether or not there are arbitrarily complex 0–1 valued functions with optimal algorithms on any RASP model is not known. We show, however, that there is an (infinitely often) arbitrarily complex partial recursive function which has only 0–1 values (when defined) with an optimal algorithm on RASP3.

Let us fix some standard numeric encoding of RASP3 programs so that (i) each RASP3 program is assigned some nonnegative integer, (ii) each nonnegative integer corresponds to a unique RASP3 program and (iii) the RASP3 program corresponding to an integer n can be determined and stored in consecutive registers of RASP3 in no more than n steps, for all but finitely many n. Let $\varphi_0, \varphi_1, \varphi_2, \cdots$ be an enumeration of the functions computed by the RASP3 programs encoded by $0, 1, 2, \cdots$. We shall say that i is an index for φ_i if the program whose index is i (denoted by P_i) computes φ_i. Let $\Phi_1, \Phi_2, \Phi_3, \cdots$ be an enumeration of the functions describing the running times of $P_1, P_2, P_3, \cdots$.

In order to describe the subsequent "diagonalization" arguments on RASP3 machines, we shall specify the following numeric encoding of RASP3 programs. Each instruction (c_1, c_2, c_3) will be assigned the integer code $c_1m^2 + c_2m + c_3$ ($m = 12$, since RASP3 has only 11 instruction types). Let P be a RASP3 program consisting of the successive integer codes (values) $I_1, I_2, I_3, \cdots, I_k$, and let σ_2 be the "pairing" function defined by $\sigma_2(x, y) = 2^x(2y + 1) - 1$. The numeric code for P will then be chosen to be $\sigma_2(k, \sigma_2(I_1, \sigma_2(I_2, \cdots, \sigma_2(I_{k-1}, I_k) \cdots)))$.

Let $\pi_1(n)$ be the maximum power of two that divides $n + 1$ and let $\pi_2(n) = (((n + 1) \div 2^{\pi_1(n)}) - 1) \div 2$. That is, $n = \sigma_2(\pi_1(n), \pi_2(n))$, $x = \pi_1(\sigma_2(x, y))$, and $y = \pi_2(\sigma_2(x, y))$.

A RASP3 program can obtain the program P_n corresponding to n by: (i) obtaining $k = \pi_1(n)$, which corresponds to the number of instructions in P_n, and (ii) obtaining $\pi_1(\pi_2(n))$, $\pi_1(\pi_2(\pi_2(n)))$, $\cdots$, $\pi_1(\pi_2^{k-1}(n))$, and $\pi_2^k(n)$, which are the numeric codes of the successive instructions. To obtain $\pi_1(n)$, a RASP3 program need only count the number of times 2 divides $n + 1$. To obtain $\pi_2(n)$, the RASP3 need only divide $n + 1$ by $2^{\pi_1(n)}$, subtract one and divide by two. It is therefore clear that, with the encoding, a RASP3 can obtain P_n from n in a number of steps no greater than n, for all but finitely many n.

After obtaining P_n from n as described, it may be desired to simulate the program P_n on n, i.e., obtain $\varphi_n(n)$. There are at least two ways to do this: (i) store P_n in consecutive registers, starting with some register d, and then transfer control to P_n with n in AC and $d - 1$ in the base register; or (ii) store P_n in consecutive registers and simulate each instruction of P_n by detecting the instruction in the current register and performing whatever action is specified. The first method requires the use of a base register, since the program P_n is designed to operate (by hypothesis) as if the first instruction were in register one, but P_n has been shifted in memory to begin with register d. However, by storing $d - 1$ initially in the base register, the RASP3 program may execute P_n starting at location d in the same manner as if P_n were located at register one with zero initially in the base register. The second method of simulating P_n on n will also be used. This method is desired when a parallel computation is required (such as decrementing the "clock" value in order to stop the simulation after a specified number of steps). Clearly a RASP3 can (by division) obtain c_1, c_2 and c_3 from $c_1m^2 + c_2m + c_3$ and simulate the action of instruction type c_3 on the argument c_1 (c_2 indicates what type of argument this is) in a fixed constant number of steps.

A function $T(n)$ whose values are not necessarily 0 and 1 can also be computed by a RASP3 by defining, in this case, the result of a RASP3 program to be the integer contained in the accumulator when either type of HALT instruction is encountered. We shall need to consider this type of computation as well, as is illustrated in the following definition.

DEFINITION. A function $T(n)$ on the natural numbers is RASP3 *time-constructable* if there is a RASP3 program P which computes $T(n)$ in time $O(T(n))$.

A RASP3 can compute $T(n) = n^{2^m}$, for example, in $T'(n) = 2m$ steps by m iterations of (i) storing the content of AC, and (ii) multiplying the AC by the stored value. Thus $T(n) = n^{2^m}$, for any m, is time-constructable. A RASP3 can compute $T(n) = 2^{\Phi_i(n)}$, for any time complexity function $\Phi_i(n)$, in $c\Phi_i(n)$ steps, for some constant c. Thus there are arbitrarily large time-constructable functions. By showing

that the family of time-constructable functions is closed under various operations, such as composition and multiplication, the reader may convince himself that the family of time-constructable functions is rich indeed. We note that if $T(n)$ is a time-constructable function and $\alpha > 0$ is a rational number, then $\alpha T(n)$ is also time-constructable. If $\alpha = m/n$, then a RASP3 program need only compute $T(n)$, divide by n, and multiply by m.

THEOREM 3.2. *For every* RASP3 *time-constructable function* $T(n)$ *such that* $T(n) \geqq n$, *there is a* 0–1 *valued function* $\psi(n)$ *which can be computed by a* RASP3 *program in time* $O(T(n))$ *but cannot be computed by any* RASP3 *program in time* $T(n)$.

Proof. The function ψ is defined as follows:

$$\psi(n) = \begin{cases} 1 \dot{-} \varphi_n(n) & \text{if } \Phi_n(n) \leqq T(n), \\ 0 & \text{otherwise.} \end{cases}$$

It follows that ψ is not computed by any RASP3 program in time $T(n)$, since if i is an index for ψ and $\Phi_i(n) \leqq T_i(n)$ for all n, then we have the contradiction $\varphi_i(i) = \psi(i) = 1 \dot{-} \varphi_i(i)$. We now show that ψ is computed by a RASP3 program in time $cT(n)$, for some constant $c > 1$.

Let $T(n)$ be computed by the RASP3 program P' in time $T'(n)$, where $T'(n)$ is $O(T(n))$. A RASP3 program P may compute ψ as follows (on input n):

1. P computes $T(n)$ using the RASP3 program P' and stores this result in a memory register, called the "clock" register.

2. P obtains the RASP3 program P_n corresponding to the input n and stores this instruction by instruction in consecutive memory registers.

3. P simulates P_n on n. After each instruction of P_n is detected and executed, P decreases the clock register by one. If the content of the clock register ever becomes zero, then P halts with the value 0. Otherwise, if P_n halts with value $\varphi_n(n)$, then P halts with the value $1 \dot{-} \varphi_n(n)$.

The RASP3 program P operates in an amount of time bounded by $T'(n) + n + c_0 T(n)$, for some constant c_0. That is, $T'(n)$ steps are required in step 1, at most n steps are required for step 2, and each instruction of P_n can be detected (by division) and executed in some small number of steps. Since $T(n) \geqq n$, P computes ψ in time $cT(n) \geqq T'(n) + n + c_0 T(n)$ for some constant c and almost all n.[5] □

COROLLARY 3.1. *For every* $\varepsilon > 0$ *there exist arbitrarily complex* 0–1 *valued functions which can be computed by a* RASP3 *program in time* $T(n)$ *but cannot be computed in time* $(1 - \varepsilon)T(n)$ *by any* RASP3.

Proof. The result will follow from the fact that there are arbitrarily large RASP3 time-constructable functions. If $F(n)$ is RASP3 time-constructable, then $\alpha F(n)$ is RASP3 time-constructable, for any rational number $\alpha > 0$. Without loss of generality, let $\varepsilon > 0$ be a rational number. Consider the sequence $F(n), (1 - \varepsilon)F(n), (1 - \varepsilon)^2 F(n), \cdots$. By Theorem 3.2 there is a constant $c > 1$ such that some 0–1 valued function $\psi(n)$ can be computed in time $F(n)$ but cannot be computed in time $\lfloor F(n)/c \rfloor$. Since $(1 - \varepsilon)^i < c^{-1}$, for some $i \geqq 1$, there must be a j such that ψ is computed in time $(1 - \varepsilon)^j F(n)$ but is not computed in time $(1 - \varepsilon)^{j+1} F(n)$. Let $T(n) = (1 - \varepsilon)^j F(n)$. □

[5] The reader should note that if a RASP3 program P computes $\psi(n)$ in time $T(n)$ for all $n \geqq n_0$, then a RASP3 program P' which computes $\psi(n)$ in time $T(n)$ for all n can be constructed. P' need only store a table of values of $\psi(n)$ and test if the input number n is greater than n_0 or not.

We note that the RASP3 program given in the proof of Theorem 3.2 is a fixed program (it does not modify its instructions during the course of a computation). Corollary 3.1 states that for every $\varepsilon > 0$ there is an arbitrarily complex total recursive 0–1 valued function f such that f can be computed in time $T(n)$ but not $(1 - \varepsilon)T(n)$. We can obtain a better result, if we agree to consider partial recursive functions. For self-modifying RASP3 programs, the result can be sharpened to say: there is an arbitrarily complex partial recursive function f which has only 0 or 1 values (whenever defined) such that f can be computed in time $T(n)$ but cannot be computed in time $(1 - \varepsilon)T(n)$, for any $\varepsilon > 0$.

Consider the function defined by $g(n) = \varphi_{\pi_1(n)}(\pi_2(n))$. We will show that $g(n)$ can be computed by a RASP3 in time $G(n) = c\pi_1(n) + \Phi_{\pi_1(n)}(\pi_2(n))$, for some constant $c \geqq 1$, and that no RASP3 can compute $g(n)$ in time $(1 - \varepsilon)G(n)$, for any $\varepsilon > 0$.

LEMMA 3.1. *The function $g(n)$ can be computed in time $G(n) = c\pi_1(n) + \Phi_{\pi_1(n)}(\pi_2(n))$, for some constant $c \geqq 1$.*

Proof. A RASP3 program P to compute $g(n)$ may be constructed as follows: P stores in consecutive registers $P_{\pi_1(n)}$, starting with some register d occurring after the registers used by the resident program P, puts $\pi_2(n)$ in the accumulator and $d - 1$ in the base register, and transfers control to register d.

There exists a constant $c \geqq 1$ such that the number of steps executed by P before control is passed to $P_{\pi_1(n)}$ is at most $c\pi_1(n)$. Thus the total amount of time is $G(n) = c\pi_1(n) + \Phi_{\pi_1(n)}(\pi_2(n))$. □

Next we show that the function $G(n)$ is infinitely often arbitrarily large.

LEMMA 3.2. *For every recursive function $r(n)$ there are infinitely many n such that $G(n) > r(n)$.*

Proof. Let $r'(n)$ be a monotone nondecreasing recursive function such that $r'(n) \geqq r(n)$, for all n. Let j be an index such that $\Phi_j(n) > r'(2^n)$, for all but finitely many n. For large m, let $n_m = 2^j(2m + 1) - 1$; then (for some constant $c \geqq 1$)

$$\begin{aligned} G(n_m) &= c\pi_1(n_m) + \Phi_{\pi_1(n_m)}(\pi_2(n_m)) \\ &= cj + \Phi_j(m) > r'(2^m) \geqq r(n_m) \end{aligned}$$

□

Lemma 3.2 does not imply immediately that $g(n)$ is arbitrarily complex. However, after $G(n)$ is shown to be the optimal time for computing $g(n)$ (as will be done in the following theorem), this fact will follow.

LEMMA 3.3. *The function $g(n)$ cannot be computed in time $(1 - \varepsilon)G(n)$, for any $\varepsilon < 0$.*

Proof. Suppose P were a RASP3 program which computes $g(n)$ in time $(1 - \varepsilon)G(n)$, for some $0 < \varepsilon < 1$. Let P_j be a RASP3 program such that $\varphi_j(n)$ cannot be computed in time $(1 - \varepsilon^2)\Phi_j(n)$; i.e., P_j cannot be speeded up by a factor of $(1 - \varepsilon^2)$. By Corollary 3.1 there are many such RASP3 programs. Construct next a RASP3 program P'_j to compute $\varphi_j(n)$ as follows. P'_j on input n does the following:

1. P'_j forms $2^j(2n + 1)$ in the accumulator and continues by executing step 2.
2. P'_j transfers control to a copy of the RASP3 program P which computes $g(\langle \mathrm{AC} \rangle)$.

Note that P'_j computes $\varphi_j(n)$, since $g(2^j(2n + 1)) = \varphi_j(n)$.

About six steps are sufficient for step 1; thus P'_j operates in no more than $T(n) = 6 + (1 - \varepsilon)G(2^j(2n + 1))$ steps. Since $G(n) = c\pi_1(n) + \Phi_{\pi_1(n)}(\pi_2(n))$, we have:

$$T(n) = 6 + (1 - \varepsilon)[cj + \Phi_j(n)]$$
$$< (1 - \varepsilon^2)\Phi_j(n),$$

for all but finitely many n. This contradicts the fact that no RASP3 program can compute $\varphi_j(n)$ in time $(1 - \varepsilon^2)\Phi_j(n)$. Therefore the assumption that a RASP3 program P exists which computes $g(n)$ in time $(1 - \varepsilon)G(n)$ is untenable. □

We have shown by the preceding lemmas that $g(n)$ is a complex partial recursive functions with only 0–1 values and $g(n)$ is computed by a RASP3 in time $G(n)$ but cannot be computed in time $(1 - \varepsilon)G(n)$, for any $\varepsilon > 0$. The following theorem summarizes these results.

THEOREM 3.3. *There are arbitrarily complex partial recursive functions* $g(n)$ *which* (i) *take only* 0–1 *values (when defined), and* (ii) *can be computed by a* RASP3 *in time* $G(n)$ *but cannot be computed by any* RASP3 *in time* $(1 - \varepsilon)G(n)$, *for any* $\varepsilon > 0$.

Appendix. The following is the RAM program described in Theorem 2.2. The initial contents of POINT, POINT(1), POINT(2), ⋯, POINT(k), etc., and their significance are described in Theorem 2.2.

```
A: TOTAL    ← X_POINT
   POINT    ← POINT + ONE
   TEMP     ← X_POINT                    "Change symbol on tape 1"
   X_POINT(1) ← TEMP
   POINT    ← POINT + ONE
   TEMP     ← X_POINT                    "Change symbol on tape 2"
   X_POINT(2) ← TEMP
            ...
   POINT    ← POINT + ONE
   TEMP     ← X_POINT                    "Change symbol on tape k"
   X_POINT(k) ← TEMP
   POINT    ← POINT + ONE
   TEMP     ← X_POINT                    "Change position of head 1"
   POINT(1) ← POINT(1) + TEMP
   POINT    ← POINT + ONE
   TEMP     ← X_POINT                    "Change position of head 2"
   POINT(2) ← POINT(2) + TEMP
            ...
   POINT    ← POINT + ONE
   TEMP     ← X_POINT                    "Change position of head k"
   POINT(k) ← POINT(k) + TEMP
   TEMP     ← X_POINT(1)
   TOTAL    ← TOTAL + TEMP
   TEMP     ← X_POINT(2)
   TOTAL    ← TOTAL + TEMP               "Calculate address of next instruction"
            ...
   TEMP     ← X_POINT(k)
   TOTAL    ← TOTAL + TEMP
```

```
    POINT   ← POINT + ONE
    TEMP    ← X_POINT
    TRA     B if TEMP > 0      } "Has a final state been reached?"
    POINT   ← TOTAL
    TRA     A if ONE > 0
B:  ACCEPT
```

The reader should note that in the above program, an instruction such as TEMP $\leftarrow X_{\text{POINT}}$ is an indirectly addressed instruction, since POINT is a register.

Acknowledgment. The authors are indebted to the anonymous referees for their many significant comments, which hopefully have created a more readable paper.

REFERENCES

[1] S. A. COOK, *Linear time simulation of deterministic two-way pushdown automata*, Information Processing 71, North-Holland, Amsterdam, 1972, pp. 75–80.

[2] S. A. COOK AND R. A. RECKHOW, *Time bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[3] J. HARTMANIS, *Computational complexity of random access stored program machines*, Math. Systems Theory, 5 (1971), pp. 232–245.

[4] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

NONCANONICAL EXTENSIONS OF BOTTOM-UP PARSING TECHNIQUES*

THOMAS G. SZYMANSKI† AND JOHN H. WILLIAMS‡

Abstract. A bottom-up parsing technique which can make nonleftmost possible reductions in sentential forms is said to be *noncanonical.* Nearly every existing parsing technique can be extended to a noncanonical method which operates on larger classes of grammars and languages than the original technique. Moreover, most of the resulting parsers run in time linearly proportional to the length of their input strings.

Several such extensions are defined and analyzed from the points of view of both power and decidability. The results are presented in terms of a general bottom-up parsing model which yields a common decision procedure for testing membership in many of the existing and extended classes.

Key words. linear time parsing, noncanonical parsing, nondeterministic languages, bottom-up parsing

1. Introduction. Since 1965 when Knuth suggested that the power of formal parsing techniques might be increased by allowing them to reduce not only handles but other phrases of sentential forms as well, there have been few investigations exploring this possibility. In this paper we consider this technique which we call noncanonical parsing. We present a descriptive framework in which to consider bottom-up parsing techniques in general and noncanonical techniques in particular, and we use this framework to describe some existing parsing techniques and to suggest their noncanonical extensions.[1]

Very simply stated, the class of grammars that can be parsed by bottom-up reduction methods operating under a left to right scan has been progressively approximated by the BRC notion of Floyd [4], then the LR(k) notion of Knuth [7] and finally by the LR-regular notion of Cohen and Culik [2] (we overlook the right to left transduction of the latter method in making this oversimplified summary). Essentially, each of these methods generalizes its predecessor in that it enlarges the class of grammars (and sometimes the class of languages) to which it is applicable. This generality has been obtained by discovering ways to increase the amount of context used in making decisions without increasing the complexity of the decisions so much that the parser can no longer parse in linear time. Note that each of these methods has required the parser to produce a canonical parse.

Our approach has been to remove the restriction that a parser must always reduce the handle of a sentential form and to study the properties of the resulting noncanonical parsers. In §§ 2 and 3 we present basic definitions and develop a general model of bottom-up parsing techniques with which we can describe and characterize some of the existing methods. Then, in § 4, we consider the noncanonical extensions of these methods and consider to what degree these extensions

* Received by the editors January 10, 1975, and in revised form June 30. This research was supported in part by the National Science Foundation under Grants GJ 42-512 and GJ 33171X.

† Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.

‡ Department of Computer Science, Cornell University, Ithaca, New York 14850.

[1] A preliminary version of this paper was presented at the 1973 SWAT conference [10].

possess the useful properties of (i) membership in the class being decidable, and (ii) membership in the class implying the existence of a linear time parser for the grammar. Finally, we demonstrate that the noncanonical versions of the various methods considered are applicable to more grammars than their canonical counterparts, and we compare the various classes of languages thus obtained.

In order to motivate the more formal treatment of § 3, we will attempt to give the flavor of noncanonical techniques by considering the non-LR-regular language

$$L_1 = \{a^n b^n c^m d^{m+l} | n, m, l \geqq 1\} \cup \{a^n b^{2n} c^m d^m | n, m \geqq 1\}$$

and constructing a BCP grammar for it. A BCP grammar is essentially a grammar which can be parsed by a noncanonical, bounded context parser [12]. We emphasize that this is intended to be merely an intuitive discussion; the formal definition appears in § 4.

L_1 is not LR-regular since any grammar for L_1 will require the handle of some sufficiently long sentence to be at the a-b interface, and the context required to distinguish between the two alternative parses, namely, whether there are more d's than c's in the remaining portion of the string, cannot be determined by a partition into regular sets. L_1 is a BCP language, however, since the grammar G_1 given in Fig. 1 is a BCP(1, 1) grammar for it.

Intuitively, G_1 allows a noncanonical parser to postpone any decisions about the a's and b's until the c's and d's have been reduced. It is then a simple matter to tell whether there were originally more d's than c's. This information is then transmitted back to the a-b interface via the B and B' one-productions. In Fig. 1, we associate with each production, $A_i \to x_i$, of G_1 a set of "parsing contexts," i.e., contexts in which it is always correct to reduce x_i to A_i. For example, any occurrence of the string Yd can always be reduced to a Y (production 7), but the string b can be reduced to a B (production 12) only when it occurs next to a B or a Y.

Actually, the parsing contexts listed in Fig. 1 are only a subset of those produced by the general BCP analyzer. For example, the string Zd could be reduced to a Y (production 8) not only when it follows a b but also whenever it

	production	parsing contexts
G_1: 1.	$S \to XY$	$(\vdash, \dashv)$
2.	$S \to X'Y'$	$(\vdash, \dashv)$
3.	$X \to aXB$	(Λ, Λ)
4.	$X \to aB$	(Λ, Λ)
5.	$X' \to aX'B'B'$	(Λ, Λ)
6.	$X' \to aB'B'$	(Λ, Λ)
7.	$Y \to Yd$	(Λ, Λ)
8.	$Y \to Zd$	(b, Λ)
9.	$Y' \to Z$	$(\Lambda, \dashv)$
10.	$Z \to cZd$	(Λ, Λ)
11.	$Z \to cd$	(Λ, Λ)
12.	$B \to b$	$(\Lambda, B), (\Lambda, Y)$
13.	$B' \to b$	$(\Lambda, B'), (\Lambda, Y')$

FIG. 1. *A BCP grammar for* L_1

follows a B. The subset we have presented is sufficient to allow the correct parsing of any sentence of G_1. We illustrate this by considering the parse of the sentence

$$a^2b^2c^2d^3$$

in Fig. 2. At each step we have underlined the leftmost string which occurs in a parsing context, and we indicate the appropriate reduction.

step	string	applicable reduction
1.	$\vdash aabbc\underline{cd}dd\dashv$	11
2.	$\vdash aabb\underline{cZ}dd\dashv$	10
3.	$\vdash aabb\underline{Zd}\dashv$	8
4.	$\vdash aab\underline{bY}\dashv$	12
5.	$\vdash aa\underline{bB}Y\dashv$	12
6.	$\vdash a\underline{aB}BY\dashv$	4
7.	$\vdash \underline{aXB}Y\dashv$	3
8.	$\vdash \underline{XY}\dashv$	1
9.	$\vdash S\dashv$	

FIG. 2. *A sample parse of a sentence in* $L(G_1)$

Notice that even though only one character of left and right context is ever used, the important information can be passed back to the crucial a-b interface by rippling back over the b's via the one-production, $B \to b$. This is the technique that gives the noncanonical methods their additional power. It will be seen in the next section that this increase in power has not been achieved at the expense of the two important properties mentioned earlier, namely, decidability of membership in the class and linear time parsing.

We want to reemphasize that the purpose of this example was to give an intuitive feeling for the formal treatment to follow. To this end, the set of parsing contexts exhibited in the preceeding example has been considerably simplified from those contexts required by the formal definition of BCP parsing which will be given in § 4. In particular, it will be seen that BCP parsers can be applied to arbitrary sentential forms as well as sentences. This simplification was made here in order to expose the essential features of noncanonical parsing.

2. Basic definitions. In this section we exhibit our notation. First, we need the usual grammatical concepts.

DEFINITION 2.1. A *context-free grammar* G is a quadruple (V, Σ, P, S) where V and Σ are finite sets called, respectively, the *vocabulary* and *terminals* of G (the set $N = V - \Sigma$ is called the *nonterminals* of G), P is a finite subset of $N \times V^*$ called the *productions* of G, and $S \in N$ is called the *start symbol* of G.

We adhere to the usual convention of using $A, B, C, \cdots$ to denote elements of N, $X, Y, Z, \cdots$ to denote elements of V and $\alpha, \beta, \gamma, \cdots$ to represent elements of V^*. Note that we distinguish between φ, an element of V^*, and $\varnothing$, the empty set.

DEFINITION 2.2. We define the binary relation $\Rightarrow$ on V^* by saying that $\alpha \Rightarrow \beta$ iff $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \beta_1 \alpha_2$ and $A \to \beta_1$ is in P for some $A \in N$ and $\alpha_1, \alpha_2, \beta_1 \in V^*$. The set of *sentential forms* of G is the set $\mathrm{SF}(G) = \{\alpha \in V^* | S \overset{*}{\Rightarrow} \alpha\}$. The *language* of G is the set of terminal sentential forms, that is, $L(G) = SF(G) \cap \Sigma^*$.

In dealing with parsers, we need the concept of a phrase. We assume that the reader is familiar with the concept of a derivation tree for a grammar. The next definition is basic to this paper.

DEFINITION 2.3. Let T be a derivation tree for some sentential form $\beta\alpha\gamma$ of the context-free grammar G. We say that the ordered pair $(A \to \alpha, i)$ is a *phrase of* T if $A \to \alpha \in P$ and $i = |\beta\alpha|$ and there exists a derivation corresponding to T of the form $S \overset{*}{\Rightarrow} \beta A\gamma \Rightarrow \beta\alpha\gamma$.

Notice that phrases are defined only in terms of derivation trees. If G is an unambiguous grammar, then and only then will it make sense to talk about the phrases of a sentential form.

We next linearize the concept of a derivation tree in two ways. A *description language* for a grammar will be a set of bracketed strings denoting a (unique) derivation tree, and the *phrase language* for a grammar will describe the location of all the phrases of a sentential form relative to a given derivation tree.

DEFINITION 2.4. Let $G = (V, \Sigma, P, S)$ be a CFG. Let $\mathcal{B} = \{\,]_i | 1 \leqq i \leqq |P|\}$ be a set of new characters. We call $\mathcal{B}$ the set of *brackets* for G. Let $\bar{N} = \{\bar{A} | A \in N\}$ also be a new set of characters. The *description language* for G is the language DL(G) generated by the grammar $G' = (\bar{N} \cup V \cup \mathcal{B}, V \cup \mathcal{B}, P', \bar{S})$, where $P' = \{\bar{A} \to Y_1 \cdots Y_{n_i}]_i | A \to X_1 \cdots X_{n_i}$ is the ith production of P and $Y_j = X_j$ or $\bar{X}_j$ for $1 \leqq j \leqq n_i\}$. The *phrase language* PL(G) for G is defined by the context-free grammar $G'' = (\bar{N} \cup V \cup \mathcal{B}, V \cup \mathcal{B}, P'', \bar{S})$, where $P'' = \{\bar{A} \to Y_1 \cdots Y_{n_i} | A \to X_1 \cdots X_{n_i}$ is the ith production of P, $Y_j = X_j$ or $\bar{X}_j$ for $1 \leqq j \leqq n_i$ and at least one $Y_j = \bar{X}_j\} \cup \{\bar{A} \to X_1 \cdots X_{n_i}]_i | A \to X_1 \cdots X_{n_i}$ is the ith production of $P\}$.

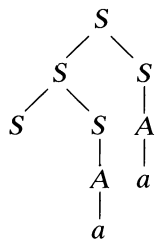
For example, consider the grammar whose productions are $S \to SS | A$ and $A \to a$. DL(G) is defined by the set of productions

$$\bar{S} \to \bar{S}\bar{S}]_1 | \bar{S}S]_1 | S\bar{S}]_1 | SS]_1$$
$$\bar{S} \to A]_2 | \bar{A}]_2$$
$$\bar{A} \to a]_3.$$

PL(G) is defined by the productions

$$\bar{S} \to \bar{S}\bar{S} | \bar{S}S | S\bar{S} | SS]_1$$
$$\bar{S} \to \bar{A} | A]_2$$
$$\bar{A} \to a]_3.$$

Consider now the following derivation tree according to G:



This tree is represented by the strings

$$\varphi_1 = Sa]_3]_2]_1 a]_3]_2]_1 \in \mathrm{DL}(G)$$

and

$$\varphi_2 = Sa]_3a]_3 \in \mathrm{PL}(G).$$

Notice that the brackets in a string of DL(G) capture the complete structure of some derivation tree, whereas the brackets of a string in PL(G) describe only the locations of the phrases for that tree. Notice also that if G is ambiguous, there may exist more than one string in DL(G) or PL(G) corresponding to a given sentential form of G. The mapping between SF(G) and DL(G) or PL(G) is provided by the string homomorphism defined below.

DEFINITION 2.5. Let $G = (V, \Sigma, P, S)$ be a CFG and $\mathcal{B}$ be the bracket set for G. Let $\#$ be a special character not in $V \cup \mathcal{B}$. Define $m: (V \cup \mathcal{B} \cup \{\#\})^* \to (V \cup \{\#\})^*$ by

$$m(\alpha) = \begin{cases} Y \cdot m(\gamma) & \text{if } \alpha = Y\gamma, \quad Y \in V \cup \{\#\}, \\ m(\gamma) & \text{if } \alpha = Y\gamma, \quad Y \in \mathcal{B}, \\ \Lambda & \text{if } \alpha = \Lambda; \end{cases}$$

define $h : (V \cup \mathcal{B} \cup \{\#\})^* \to (V \cup \mathcal{B})^*$ by

$$h(\alpha) = \begin{cases} Y \cdot h(\gamma) & \text{if } \alpha = Y\gamma, \quad Y \in V \cup \mathcal{B}, \\ h(\gamma) & \text{if } \alpha = Y\gamma, \quad Y = \#, \\ \Lambda & \text{if } \alpha = \Lambda. \end{cases}$$

The maps m and h are respectively referred to as the *bracket erasing* and *marker erasing homomorphisms.* The reader may verify that the restrictions of m to DL(G) or PL(G) are one-to-one if and only if G is unambiguous. The reader may further verify that $m(\mathrm{DL}(G) \cup \{S\}) = m(\mathrm{PL}(G) \cup \{S\}) = \mathrm{SF}(G)$.

Throughout this paper we shall be dealing with various parameterized classes of grammars and languages. All parameters are considered to be free variables unless explicitly bound. Thus the term "class of LR(k) grammars" means "$\{G | \exists k$ such that G is LR(k)$\}$," and the statement "it is undecidable whether a grammar is LR(k)" means "it is undecidable whether there exists a k such that a given grammar is LR(k)."

3. A model of bottom-up parsing. In the sequel it will be necessary to discuss the operation of parsers which make noncanonical reductions. For this purpose we define a general model of bottom-up parsing which emphasizes the context used by a parser when making reductions. No restriction will be placed on where in a subject string the parser chooses to make a reduction.

DEFINITION 3.1. Let $G = (V, \Sigma, P, S)$ be a CFG. A *reduction pattern* for G is a pair $(R, A \to \alpha)$, where $A \to \alpha \in P$ and $R \subseteq V^*\alpha \# V^*$. The reduction pattern is said to *apply* to the string φ if φ is of the form $\beta_1\beta_2\alpha\beta_3\beta_4$ and $\beta_2\alpha \# \beta_3 \in R$. The string $\varphi' = \beta_1\beta_2 A\beta_3\beta_4$ is said to be a *reduction of* φ *implied* by $(R, A \to \alpha)$.

The set R thus specifies those contexts in which a given reduction may be made.

Note that no restrictions have been made on the set R as far as finiteness or even recursiveness are concerned. Of course, the interesting cases will involve "nice" sets in the sense of being regular, bounded to the right of the $\#$, etc.

In order for a reduction pattern to be useful, we must place some additional qualifications upon it. In particular, let us require that an implied reduction must be correct in the sense that the substring "pruned" by the pattern from a sentential form φ is in fact a phrase of φ relative to *all* derivation trees for φ.

DEFINITION 3.2. Let $(R, A \to \alpha)$ be a reduction pattern for a CFG $G = (V, \Sigma, P, S)$ in which $A \to \alpha$ is the ith production. We say that $(R, A \to \alpha)$ is a *parsing pattern* for G if

(i) $\psi \in \mathrm{PL}(G)$,

(ii) $m(\psi) = \beta_1\beta_2\alpha\beta_3\beta_4$, and

(iii) $\beta_2\alpha \# \beta_3 \in R$

imply $\psi \in m^{-1}(\beta_1\beta_2)\alpha]_i m^{-1}(\beta_3\beta_4)$.

Consider, for example, the grammar G whose productions are $S \to SS|A$ and $A \to a$. The reduction pattern $(\{a \#\}, A \to a)$ is a parsing pattern because any "a" can only be derived from an "A." On the other hand, the pattern $(\{SS \#\}, S \to SS)$ is not a parsing pattern because the pair of S's in the sentential form SSA is not always an immediate S derivative. This is captured by the definition in that both $\psi_1 = SS]_1 A]_2$ and $\psi_2 = SSA]_2$ are elements of $\mathrm{PL}(G)$, yet SS is not a phrase of SSA relative to *all* derivation trees (e.g., $\psi_2 \notin m^{-1}(\Lambda)SS]_1 m^{-1}(A)$).

At this point we are ready to give a language-theoretic characterization of those reduction patterns which are also parsing patterns.

LEMMA 3.3. *Let G be a* CFG *and $\mathscr{B}$ be the set of brackets for G. Let $A \to \alpha$ be the i-th production of G and $(R, A \to \alpha)$ be a reduction pattern for G. Define the set*

$$M = h^{-1}(\mathrm{PL}(G)) \cap m^{-1}(V^*RV^*) \cap m^{-1}(V^*)V \# (\mathscr{B} -]_i)^* Vm^{-1}(V^*).$$

Then $(R, A \to \alpha)$ is a parsing pattern for G if and only if $M = \varnothing$.

Proof. Loosely speaking, M is the set of "mistakes" committed by the reduction pattern, and a parsing pattern will be a reduction pattern whose set of mistakes is empty. We proceed more formally.

Only if. Suppose $M \neq \varnothing$. We must show that $(R, A \to \alpha)$ is not a parsing pattern. Let $\psi' \in M$. ψ' can be uniquely written as $\varphi_1 X_1 \# \gamma X_2 \varphi_2$, where $\varphi_1, \varphi_2 \in m^{-1}(V^*)$, $X_1, X_2 \in V$, $\gamma \in (\mathscr{B} -]_i)^*$. Let $\psi = h(\psi')$. Then $\psi \in \mathrm{PL}(G)$. By definition of M, $\exists \beta_1, \beta_2, \beta_3, \beta_4 \in V^*$ such that $m(\varphi_1 X_1) = \beta_1\beta_2\alpha$, $m(X_2\varphi_2) = \beta_3\beta_4$, and $\beta_2\alpha \# \beta_3 \in R$. (Note that since $\psi \in \mathrm{PL}(G)$, $m(\psi) = \beta_1\beta_2\alpha\beta_3\beta_4 \in \mathrm{SF}(G)$.) Notice that X_1 is the last character of $\beta_1\beta_2\alpha$ and that X_2 is the first character of $\beta_3\beta_4$. If $\psi = \varphi_1 X_1 \gamma X_2 \varphi_2 \in m^{-1}(\beta_1\beta_2)\alpha]_i m^{-1}(\beta_3\beta_4)$, then γ would have to contain $]_i$. But by hypothesis, $\gamma \in (\mathscr{B} -]_i)^*$. Hence $\psi \notin m^{-1}(\beta_1\beta_2)\alpha]_i m^{-1}(\beta_3\beta_4)$, and we conclude that $(R, A \to \alpha)$ is not a parsing pattern.

If. Suppose $(R, A \to \gamma)$ is not a parsing pattern. We must show that $M \neq \varnothing$. Since $(R, A \to \alpha)$ is not a parsing patter, $\exists \psi \in \mathrm{PL}(G)$ such that $m(\psi) = \beta_1\beta_2\alpha\beta_3\beta_4$ with $\beta_2\alpha \# \beta_3 \in R$ but $\psi \notin m^{-1}(\beta_1\beta_2)\alpha]_i m^{-1}(\beta_3\beta_4)$. By picking $X_1 =$ last character of $\beta_1\beta_2\alpha$ and $X_2 =$ first character of $\beta_3\beta_4$, we can uniquely write $\psi = \varphi_1 X_1 \gamma X_2 \varphi_2$, with $\varphi_1, \varphi_2 \in (V \cup \mathscr{B})^*$, $X_1, X_2 \in V$, $\gamma \in \mathscr{B}^*$, $m(\varphi_1 X_1) = \beta_1\beta_2\alpha$, and $m(X_2\varphi_2) = \beta_3\beta_4$. Moreover, since $\psi \notin m^{-1}(\beta_1\beta_2)\alpha]_i m^{-1}(\beta_3\beta_4)$, we must have $\gamma \in (\mathscr{B} -]_i)^*$. (Note that in $\mathrm{PL}(G)$, any occurrence of $]_i$ must be immediately preceded by α with no intervening brackets.) Now consider the string $\psi' = \varphi_1 X_1 \# \gamma X_2 \varphi_2$. Since $h(\psi') = \psi$ and $\psi \in \mathrm{PL}(G)$, we have $\psi' \in h^{-1}(\mathrm{PL}(G))$. Since $m(\psi') = \beta_1\beta_2\alpha \# \beta_3\beta_4$ and $\beta_2\alpha \# \beta_3 \in R$ by hypothesis, we have $\psi' \in m^{-1}(V^*RV^*)$. Finally, since

$\varphi_1, \varphi_2 \in m^{-1}(V^*)$, $X_1, X_2 \in V$ and $\gamma \in (\mathcal{B} -]_i)^*$, we have $\psi' \in m^{-1}(V^*)V \# (\mathcal{B} -]_i)^* V m^{-1}(V^*)$. Thus $\psi' \in M$ and $M \neq \varnothing$, as was to be shown. □

Since the set M defined in Lemma 3.3 is a context-free set whenever the set R is regular, we have a corollary.

COROLLARY 3.4. *It is decidable whether a regular reduction pattern is a parsing pattern.*

Notice that all commonly used parsing methods use reduction patterns which are in fact regular sets. This observation is easily verified for any of the precedence or bounded context algorithms. The LR(k) method of Knuth [7] essentially uses patterns which are regular sets but which have only k or fewer characters following the #. The LR-regular technique of Cohen and Culik [2] uses a pattern set which is of unbounded length to either side of the # but which is always regular. All of the above methods, however, are strictly canonical in operation. Our model thus not only includes them but also allows for their extension to noncanonical operation.

A single parsing pattern allows us to make reductions for but one production of the grammar at hand. In order to parse arbitrary sentences, we need a pattern for each production along with some guarantee that the parsing process will not "block." In other words, the set of parsing patterns must cover the set of sentential forms which are generated by the parsing process itself while processing strings of $L(G)$. Since this set of sentential forms is not necessarily a "cleanly" structured set,[2] we will substitute a simpler requirement, namely, the ability to cover the *entire* set of sentential forms. We thus are led to the following.

DEFINITION 3.5. A *parsing scheme* for a grammar G is a finite collection of reduction patterns such that

(i) each reduction pattern is a parsing pattern, and
(ii) for every sentential form of G other than S, there exists some pattern in the collection which applies to it.

This concept left unrestricted is sufficient to allow us to parse any unambiguous context-free grammar. Moreover, we have Theorem 3.6.

THEOREM 3.6. *Let $G = (V, \Sigma, P, S)$ be a* CFG *in which $S \overset{+}{\Rightarrow} S$. Then there exists a parsing scheme for G if and only if G is unambiguous.*

Proof. Only if. Suppose we have a parsing scheme $\mathcal{S}$ for G but that G is ambiguous. Therefore there exists some sentential form φ with two distinct derivation trees T_1 and T_2. We may further assume that T_1 and T_2 have no common phrases (if they do, we can "prune" the common phrases until the desired condition occurs). The sentential form φ cannot be S (because we have hypotheized that $S \overset{+}{\Rightarrow} S$), and therefore some reduction pattern in $\mathcal{S}$ applies to φ. The "phrase" located by this pattern cannot be a phrase of both T_1 and T_2, and thus the reduction pattern fails to be a parsing pattern. Thus $\mathcal{S}$ could not have been a parsing scheme after all.

If. If G is unambiguous, we can meaningfully talk about the phrases of a sentential form instead of just the phrases of a derivation tree. So now we take as a reduction pattern for the ith production, $A \to \alpha$, the set $R_i = \{\beta\alpha \# \gamma \mid S \overset{+}{\Rightarrow} \beta A \gamma\}$. This (context-free) pattern is a parsing pattern for G because whenever $\varphi = \beta\alpha\gamma \in$

[2] Indeed, with regular reduction patterns, this set need not be context-free.

SF(G) and $\beta\alpha \# \in R_i$, then there is but one derivation tree for φ and $(i, |\beta\alpha|)$ must be a phrase of that tree. Furthermore, the collection of parsing patterns $\{R_i | 1 \leqq i \leqq |P|\}$ clearly covers every sentential form except for S. □

An immediate and somewhat unfortunate corollary to the above theorem is given next.

COROLLARY 3.7. *It is undecidable whether there exists a parsing scheme for an arbitrary context-free grammar.*

Thus it appears as if our original conception of a parsing scheme is too powerful to work with. The obvious restriction (with an eye toward meaningful implementation) is to require that each reduction pattern be a regular set. This restriction yields a class of parsing schemes which can be tested for correctness as described in the following theorem.

THEOREM 3.8 [Decidability theorem for correctness of bottom-up parsers]. *Let G be a* CFG *and $\mathcal{S}$ be a finite collection of regular reduction patterns for G. Then it is decidable whether $\mathcal{S}$ is a parsing scheme for G.*

Proof. We first check that each member of the finite collection $\mathcal{S}$ is a parsing pattern. Corollary 3.4 tells us that this is decidable. We must next check that some pattern applies to each sentential form. Let $\mathcal{R}$ be the regular set $\bigcup_{R \in \mathcal{S}} V^* R V^*$. Then every sentential form is reducible if $\mathrm{SF}(G) - \{S\} \subseteq \mathcal{R}$. Since the set on the left of the $\subseteq$ is a context-free set and $\mathcal{R}$ is regular, this question is also decidable. □

In the next few paragraphs let us consider the implementation of a parser based on regular reduction patterns. If we require that only canonical reductions be made, then we essentially have a model of the LR-regular parsing process [2] and can implement the parser on a one-stack DPDA which is allowed to perform an initial right to left finite state transduction of its input. If noncanonical reductions are allowed, then one stack is no longer sufficient, and we are forced to turn to a two stack implementation. We thus have Theorem 3.9.

THEOREM 3.9. *Let G be a* CFG *and $\mathcal{S}$ be a parsing scheme for G, all of whose patterns are regular. Then $L(G)$ can be parsed by a deterministic two-stack pushdown transducer. Furthermore, there exists a constant c depending only on G such that this transducer makes at most cn^2 moves while parsing a string of length n.*

Proof. The two stacks are used to hold those portions of the current sentential form which are, respectively, to the left or right of the current "point of interest." Each reduction can be made by a left to right sweep followed by a right to left sweep over the sentential form recording states of the regular patterns as we go. Let us sketch the details of this process.

Let $M_i = (Q_i, \delta_i, V, q_{i,0}, F_i)$ be a finite state automaton which accepts the parsing pattern for the ith production. Assume that $Q_i \cap Q_j = \varnothing$ whenever $i \neq j$. Let $Q = \sum_i Q_i$. Suppose that $\psi = X_1 \cdots X_m$ is the sentential form we wish to parse and that ψ is initially stored completely in one stack with X_1 on top.

During the first left to right pass, we "pour" ψ into the other stack, converting it to $\psi' = S_1 X_1 S_2 X_2 \cdots S_m X_m$, where each S_j is a set of states from Q. Specifically,

$$q_{i,k} \in S_j \quad \text{iff} \quad q_{i,k} \in \delta_i(q_{i,0}, X_1 \cdots X_j \#).$$

Similarly, during the second pass, we "pour" ψ' back into the original stack, producing $\psi'' = S_1 X_1 T_1 S_2 X_2 T_2 \cdots S_m X_m T_m$, where

$$q_{i,k} \in T_j \quad \text{iff} \quad \delta_i(q_{i,k}, X_{j+1} \cdots X_m) \in F_i.$$

At this point there will be at least one i, j and k such that $q_{i,k} \in S_j \cap T_j$; in other words, the ith production may be reduced at position j of ψ. We perform this reduction by making an additional "pour," discarding all S and T sets as we go. We have thus produced a new sentential form $Y_1 \cdots Y_m$, stored completely in one stack, and may repeat the entire process until we have reduced the original form to the start symbol.

The details of recording the sequence of reductions on the output tape of the transducer are left to the reader. Not only must we output the index for each reduction, but also an indication as to where the reduction is to be made.

To establish the time bound, observe that if a grammar is unambiguous, then there exists a constant c_1 such that any derivation of a string of $L(G)$ of length n takes at most $c_1 n$ steps. Furthermore, no intermediate sentential form of this derivation is of length longer than $c_1 n$. Hence the parsing process described above takes at most time $(c_1 n)^2$. □

If we limit ourselves to reduction patterns, the second component of which are bounded on one or the other side of #, we can establish a linear time bound for parsing. Such patterns can be implemented in the simple automaton of Definition 3.10 below.

DEFINITION 3.10. Let $G = (V, \Sigma, P, S)$ be a CFG. A *reducing automaton* for G is a quadruple (Q, k, δ, q_0), where Q is a (not necessarily finite) set of *states*, k, the *lookahead factor*, is an integer $\geqq 1$, δ, the *move function*, is a map $\delta : Q \times V^k \to Q \cup \{i | 1 \leqq i \leqq |P|\}$, $q_0 \in Q$ is the start state. The extension of δ to $Q \times V^k V^*$ is defined inductively as follows:

$$\hat{\delta}(p, \alpha\beta X) = \begin{cases} \delta(\hat{\delta}(p, \alpha\beta), \beta X) & \text{if } |\beta X| = k \text{ and } \hat{\delta}(p, \alpha\beta) \in Q, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

A reducing automaton functions in a manner similar to that of a finite state automaton except (i) each move consults the next k characters of input, and (ii) the machine stops as soon as the move function yields the index of a production instead of a new state.

A natural collection of reduction patterns can be associated with any reducing automaton. If this collection constitutes a parsing scheme, then we call the automaton a *parsing automaton*. More formally, we note the following definition.

DEFINITION 3.11. Let $A = (Q, k, \delta, q_0)$ be a reducing automaton for $G = (V, \Sigma, P, S)$. Let $R_i = \{\beta \# \gamma | \beta \in V^*, \gamma \in V^k \text{ and } \hat{\delta}(q_0, \beta\gamma) = i\}$. Let $\mathcal{S} = \{R_i | 1 \leqq i \leqq |P|\}$. A is said to be a *parsing automaton* (PA) for G if $\mathcal{S}$ is a parsing scheme for G.

Restricting a parser to a fixed amount of lookahead beyond the right end of a phrase yields a linear time parser.

THEOREM 3.12. *Let $\mathcal{A}$ be a parsing automaton for G. Then there exists a deterministic pushdown transducer which parses sentences of G in time $O(n)$, where n is the length of the input string.*

Proof. Suppose that $\varphi = \beta_1\beta_2\alpha\gamma_1\gamma_2$ is a string with $|\beta_2| = |\gamma_1| = k$. Suppose further that $\hat{\delta}(q_0, \beta_1\beta_2\alpha\gamma_1) = i$ and the ith production is $A \to \alpha$.This means that $\hat{\delta}(q_0, \beta_1\beta_2)$ is well-defined and is a member of Q. If we now reduce φ to $\varphi' = \beta_1\beta_2 A\gamma_1\gamma_2$ and reapply $\mathcal{A}$ to find another phrase, the found phrase must be to the right of β_1. Thus in implementing $\mathcal{A}$ as a parser, after each reduction we

need only "back up" k characters in the sentential form before continuing a left to right scan.

The total time spent during parsing can thus be divided into three parts: t_1 = the time spent backing up, t_2 = the time spent moving forward over regions of the string already scanned and t_3 = the time spent advancing over previously unscanned characters. The time t_3 is clearly equal to n. The time t_1 is bounded above by $c_1 n(k+l)$, where $c_1 n$ bounds the total number of reductions made (see proof of Theorem 3.9) and l is the length of the longest right side of a production. Similarly t_2 is at most $c_1 n(k+1)$. In any case, since c_1 and l depend only on G, we conclude that $t_1 + t_2 + t_3 = O(n)$. □

Thus the parsing automaton is just the model which we need to develop the desired linear time, noncanonical extensions of existing parsing algorithms.

4. Noncanonical extensions. In this section we will show how several existing parsing methods can be extended to operate noncanonically while remaining within the framework of the theory developed in § 3. We will then analyze the generative power of these extensions with respect to both classes of parsable grammars and classes of recognizable languages.

Let us start with Floyd's bounded context grammars [4]. A grammar is m, n bounded context (BC(m, n)) if every phrase of every sentential form is uniquely distinguished by the m characters to its left and the n characters to its right. Extending this idea, a grammar is said to be *m, n bounded context parsable* (BCP(m, n)) if at least one phrase of every sentential form is uniquely distinguished by the m characters to its left and n characters to its right [12]. This extension is actually noncanonical because a BC parser was intended to operate in a strictly left to right fashion.

An equivalent definition may be phrased in the terminology of § 3. Thus a grammar is BCP(m, n) iff there exists a parsing scheme for it such that every reduction pattern takes the form of an m character string, followed by the right side of the production in question, followed by a # and then an n-character string.

The results of § 3 immediately yield the following.

THEOREM 4.1. *It is decidable whether a grammar is* BCP(m, n) *for fixed m and n.*

Proof. One simply generates all possible sets of reduction patterns which fit the BCP condition and tests their correctness using Theorem 3.8. Such a procedure is guaranteed to halt by the fact that there are but a finite number of such patterns. □

A simple example of a BCP(1, 1) grammar is G_1. It has also been shown that the BCP grammars are sufficiently powerful to generate a proper superset of the deterministic languages [12].

We mention in passing that the work of Colmerauer [3] was intended to be a noncanonical extension of simple precedence parsing. It has been shown [1], however, that the resulting class of grammars is too powerful in the sense of including some ambiguous grammars. This will never be the case with our parsers as was shown in Theorem 3.6.

Let us consider next the noncanonical extension of LR(k) parsing. Knuth [7] suggested a partially noncanonical extension by defining an LR(k, t) *grammar* as an unambiguous CFG in which every sentential form has the property that one of

its t leftmost phrases is uniquely distinguished by its left context and first k characters of right context. An example of an LR(1, 2) grammar is provided by

$$G_2 : S \to A\bar{A} | B\bar{B}$$
$$A \to a$$
$$B \to a$$
$$\bar{A} \to b\bar{A} | b\bar{B}$$
$$\bar{B} \to b\bar{B}c | bc.$$

The reader should note that G_2 is not an LR(k) grammar or even an LR-regular grammar.

Parsers for LR(k, t) grammars can be constructed using a technique similar to that used for building LR(k) parsers (see [1] for details of this latter construction). The only major difference is that the lookahead string associated with an individual LR(k) item is allowed to contain nonterminals as well as terminals. Whenever an inadequate item set (i.e., one in which the correct parsing action is undefined due to the presence of conflicting items) is reached, the parser postpones any implied reduction(s) and shifts to a new set of items. This new item set includes in its closure any items produced by expanding the leading nonterminal in the lookahead string of any postponed items. Complete details of this construction appear in [10]. The construction also serves as a test for LR(k, t)-ness.

Since the process of postponing an individual item can be repeated at most t times, no lookahead string need have a length which exceeds kt. Hence the above construction is finite and we have the next theorem.

THEOREM 4.2. *It is decidable whether an arbitrary* CFG *is* LR(k, t) *for fixed values of k and t. Furthermore, an* LR(k, t) *grammar can be parsed in linear time.*

Proof. The construction sketched above gives rise to a set of regular reduction patterns whose adequacy can be tested by Theorem 3.8. The linear time result then follows immediately from Theorem 3.12. However, since an LR(k, t) parser has to back up at most t times in succession and since each back up is of distance k, we can claim a stronger result, namely, that LR(k, t) grammars can be parsed by a DPDA. □

Note that parsing with a DPDA implies that all LR(k, t) languages are in fact deterministic languages; that is, every LR(k, t) grammar is equivalent to some LR(k) grammar. This suggests that we should consider instead the fully noncanonical generalization of the LR(k) grammars.

Accordingly, we define an LR(k, ∞) *grammar* as an unambiguous CFG in which every sentential form has some phrase which can be uniquely distinguished by its left context and first k characters of right context. G_1 is an example of an LR(1, ∞) grammar which is not LR(k, t) for any k and t.

The generalization actually introduces more parsing power than intended. More specifically, if one considers the parsing patterns induced by an LR(k, ∞) parser, one finds that these sets can in fact be nonregular context-free sets. This in turn implies the need for a PA with infinitely many states in order to do parsing. Even more serious is the fact that the following theorem holds.

THEOREM 4.3. *Let k be any nonnegative integer. The class of* LR(k, ∞) *grammars is not recursively enumerable.*

Proof. Consider the set $C = \{(G', G'') | G'$ and G'' are LR(k) and $L(G') \cap L(G'') = \emptyset\}$. It is easy to show that C is not recursively enumarable. For instance, consider the "standard" linear grammars describing potential solutions to Post's correspondence problem.

We next show that given any two LR(k) grammars G_1 and G_2, we can effectively construct a new grammar $\bar{G}(G_1, G_2)$ such that $\bar{G}$ is LR(k, ∞) iff $(G_1, G_2) \in C$. Accordingly, let $G_1 = (V_1, \Sigma, P_1, S_1)$ and $G_2 = (V_2, \Sigma, P_2, S_2)$. Without loss of generality, assume that G_1 and G_2 have disjoint sets of nonterminals. For each $\sigma_i \in \Sigma$, we replace all occurrences of σ_i in P_1 by a new nonterminal σ_i' and all occurrences of σ_i in P_2 by a new nonterminal σ_i''. At the same time, we add $\sigma_i' \to \sigma_i$ to P_1 and $\sigma_i'' \to \sigma_i$ to P_2, and call the resulting sets of productions P_1' and P_2'. Now we let $\bar{G}$ be the grammar

$$\bar{G} = (V_1 \cup V_2 \cup \Sigma' \cup \Sigma'' \cup \{S\}, \Sigma, P_1' \cup P_2' \cup \{S \to S_1 | S_2\}, S).$$

It is well known that with this standard construction, $\bar{G}$ is unambiguous iff $L(G_1) \cap L(G_2) = \emptyset$. We claim that $\bar{G}$ is LR(k, ∞) iff $\bar{G}$ is unambiguous. The only if direction is obvious. For the if direction, note that if $\bar{G}$ is unambiguous, then $L(G_1)$ and $L(G_2)$ are disjoint. Therefore in every sentence of $L(\bar{G})$, the last character, σ_i, will always be able to be reduced to a σ_i' or a σ_i'' considering the left context (the entire rest of the sentence) and the right context (the $\dashv$). Once a sentential form contains a nonterminal, that nonterminal will serve as sufficient context for its neighbor until the entire string has been reduced to a word in either Σ' or Σ''. At this point, the parsing schemes of the original LR(k) grammars can complete the reduction to $\vdash S_1 \dashv$ or $\vdash S_2 \dashv$. Hence $(G_1, G_2) \in C$ iff $\bar{G}(G_1, G_2)$ is LR(k, ∞). Since C is not recursively enumerable, the class of LR(k, ∞) grammars cannot be recursively enumerable either. □

COROLLARY 4.4. *It is undecidable whether a* CFG *is* LR(k, ∞) *even for fixed k.*

Suppose, then, that we try to capture the favorable aspects of the LR(k, ∞) method, i.e., the ability to reduce arbitrary phrases of sentential forms, while still preserving the applicability of the theory developed in the previous section. This suggests that we must weaken the discriminatory power of the left contexts by restricting them to be regular sets. If a grammar is parsable (perhaps noncanonically) by using regular reduction patterns that are k-bounded on the right, then we say that it is FSPA(k) (the abbreviation follows from the fact that such a grammar has a finite state parsing automaton using k characters of lookahead). The results of § 3 guarantee the next theorem.

THEOREM 4.5. *An* FSPA(k) *grammar is unambiguous and is parsable in linear time on a* 2PDA.

Proof. The theorem follows directly from Theorems 3.6 and 3.12. □

Let us next investigate whether this restriction to regular parsing patterns yields a decidable class of grammars. The BCP(m, n) grammars formed a recursive class (for fixed m and n) because of the fact that only a bounded number of potential reduction patterns needed to be checked via Theorem 3.8. On the other

hand, since there are arbitrarily many regular reduction patterns that are candidtates for being FSPA(k) parsing patterns, such an argument will no longer work. In fact, the question of whether an arbitrary CFG is FSPA(k) for any fixed value of k will be shown to be undecidable by the next two theorems.

The argument turns on the following general problem for deterministic languages first posed in [2]:

> Is it decidable of two arbitrary deterministic languages, L_1 and L_2, whether they are *regularly separable*, i.e., whether there exists a regular set R such that $L_1 \subseteq R$ and $L_2 \subseteq \bar{R}$?

Regular separability implies the existence of an f.s.a. which accepts all strings from L_1, rejects all strings from L_2, and can do what it pleases with strings in $\Sigma^* - (L_1 \cup L_2)$. Obviously, a necessary condition for regular separability is that L_1 and L_2 be disjoint.

The regular separability problem can be shown to be undecidable by modifying Ogden's proof [8] that it is undecidable whether an arbitrary CFG is LR-regular.

THEOREM 4.6. *It is undecidable whether two arbitrary deterministic* CFL's *are regularly separable.*

Proof. Let M be an arbitrary Turing machine with tape alphabet Σ and state set Q. Let $\mathrm{id}_i \in \Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$ represent the ith instantaneous description of M when started on blank tape. Using well established techniques [5], [9], we may define the following disjoint DCFL's.

$$L_1 = \{w_1 \# w_2 \# \cdots \# w_{2k} \# a^k \mid k \geqq 1,\ w_i \in \Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^* \text{ for } 1 \leqq i \leqq 2k,$$
$$w_{2i-1} \underset{M}{\vdash} w_{2i}^R \text{ for } 1 \leqq i \leqq k\};$$

$$L_2 = \{w_1 \# w_2 \# \cdots \# w_{2k} \# a^{2k} \mid k \geqq 1,\ w_1 = \mathrm{id}_0,$$
$$w_i \in \Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^* \text{ for } 2 \leqq i \leqq 2k,$$
$$w_{2i}^R = w_{2i+1} \text{ for } 1 \leqq i \leqq k-1\}.$$

To establish the theorem, we will show that L_1 and L_2 are regularly separable if and only if M never halts. We will thus have reduced the halting problem to the regular separability problem. The essential idea to be used here is that if M diverges, L_1 and L_2 will have arbitrarily long common prefixes which will "confuse" any f.s.a. which attempts to separate them.

Case 1. Suppose M diverges. Assume A is an $(n-1)$-state f.s.a. which separates L_1 from L_2. Consider the string

$$z = \mathrm{id}_0 \# \mathrm{id}_1^R \# \mathrm{id}_1 \# \mathrm{id}_2^R \# \cdots \# \mathrm{id}_{n!-1} \# \mathrm{id}_{n!}^R \# .$$

Then $w_1 = za^{n!} \in L_1$, and $w_2 = za^{2n!} \in L_2$. Suppose that $\delta_A(q_0, z) = p$ and $\delta_A(p, a^{n!}) = r$. Then by the usual repeated state arguments, there exists an s such that $1 \leqq s \leqq n$ and $\delta_A(p, a^{n!+is}) = r$ for any value of i. In particular, choose $i = n! \div s$ to see that $\delta_A(p, a^{2n!}) = r$. Thus $\delta_A(q_0, w_1) = \delta_A(q_0, w_2)$, and hence A cannot possibly separate L_1 and L_2.

Case 2. Suppose M halts in n steps. Thus $\mathrm{id}_0 \vdash \mathrm{id}_1 \vdash \cdots \vdash \mathrm{id}_n \nvdash \mathrm{id}_{n+1}$ and the length of each $\mathrm{id}_i \leqq n+1$. Suppose $z = w_1 \# w_2 \# \cdots \# w_{2k} \# a^j \in L_1 \cup L_2$. Then z can be classified by means of the following algorithm:

if $k \leqq n$ **then**
  if $j = k$ **then** $z \in L_1$ **else** $z \in L_2$
 else
  for $i = 1$ **to** $n+1$ **do**
    if $w_{2i-1} \neq \mathrm{id}_{i-1}$ **then** $z \in L_1$
    if $w_{2i} \neq \mathrm{id}_i^R$ **then** $z \in L_2$
  end

The above algorithm can be performed by an f.s.a., and hence L_1 and L_2 are regularly separable. □

In Theorem 4.3 we constructed a grammar which was LR(k, ∞) iff its two "halves" were disjoint (i.e., separable by a context-free set). Exactly this same construction can be used to combine two LR(1) grammars G_1 and G_2 into a new grammar G which is FSPA(1) iff $L(G_1)$ and $L(G_2)$ are regularly separable. We thus conclude the following.

THEOREM 4.7. *It is undecidable* (*even for fixed* k) *whether or not an arbitrary grammar is* FSPA(k).

The classes of LR(k), FSPA(k) and LR(k, ∞) constitute a natural hierarchy. Each class is more powerful than its predecessor (with respect to both grammars and languages). The price paid for this power is the loss of manageable decision procedures. Thus, for any value of k, the classes of LR(k), FSPA(k) and LR(k, ∞) grammars are, respectively, recursive, recursively enumerable (r.e.) and non-r.e.

If we consider context-free grammars without Λ-rules, we can show that the classes are related as depicted in the set inclusion lattice of Fig. 3. For the most part, these inclusions are reasonably straightforward consequences of the definitions. It is also not difficult to show that each inclusion is proper. One simply constructs a grammar that requires the additional parsing power of the higher class. The necessary examples are included in the Appendix.

Let us next turn our attention to a comparison of the generative power of these classes of grammars. We say that a context-free language L is of type x if there exists some grammar of type x which generates L. It frequently turns out that unequal classes of grammars generate equal classes of languages. For example, we have the following.

THEOREM 4.8. *The following classes of languages are equivalent*: (a) *deterministic* CFL's, (b) LR(k) *languages*, (c) BRC *languages*, (d) LR(k, t) *languages*.

Proof. The equivalence of (a), (b) and (c) is due to Knuth. The reason for the equivalence of (a) and (d) was alluded to in the proof of Theorem 4.2. Details may be found in [10]. □

Closely related to the deterministic languages are the LR-regular languages which are essentially those languages which can be mapped into LR(0) languages by means of a right to left gsm transduction. Cohen and Culik have shown that the LR-regular languages properly include the deterministic languages [2]. The LR-regular languages in turn are properly contained within the BCP languages, a fact which is proven below.

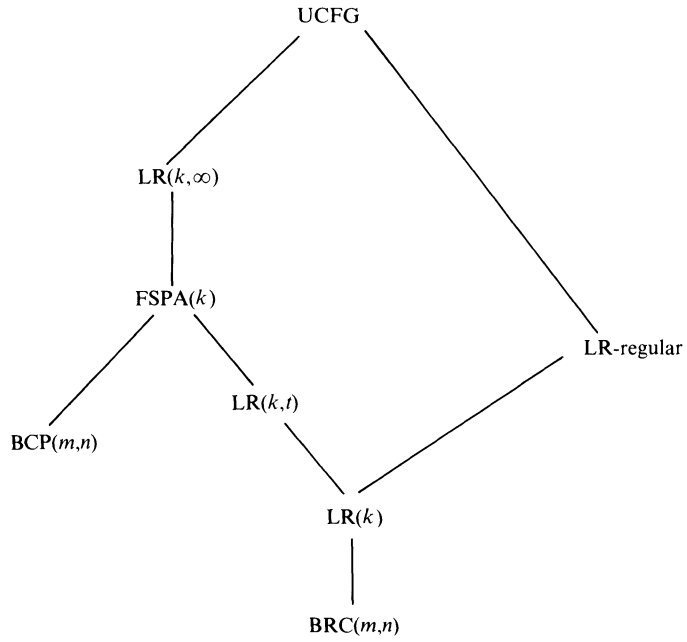


FIG. 3. *Inclusion diagram for Λ-free grammars*

THEOREM 4.9. *The class of* BCP *languages properly contains the class of* LR-*regular languages.*

Proof. (a) We will first demonstrate containment. Let L be any LR-regular language and g be its associated right to left gsm. Suppose that the alphabet of L is Σ and the set of states of g is Q. Suppose further that the start state q_0 of g is not in the range of the move function δ_g. Define a new language L' with alphabet $Q\times\Sigma\times Q$ by

$$L'=\{p_0, a_1, p_1)(p_1, a_2, p_2)\cdots(p_{k-1}, a_k, p_k)\,|\,k\geqq 1,\ a_1a_2\cdots a_k\in L,\ p_k=q_0,\ \delta_g(p_i, a_i)=p_{i-1}\text{ for }1\leqq i\leqq k\}.$$

L' then is an LR(0) language. Williams has shown that every deterministic language is a BCP language [12]. So, let $G'=(N', Q\times\Sigma\times Q, P', S')$ be a BCP grammar generating L'.

We next define a new grammar

$$G''=(N'\cup Q\times\Sigma\times Q, \Sigma, P'\cup P'', S')$$

by adding some productions to G'. More specifically, we let

$$P''=\{(p, a, q)\to a\,|\,\delta_g(q, a)=p\}.$$

Clearly $L(G'')=L$. We also claim that G'' is a BCP grammar. The necessary reduction patterns are

(i) for $\Pi\in P'$ the appropriate pattern for parsing G';

(ii) for $(p, a, q)\to a\in P''$, then $\{a\,\#\,(q, b, s)b\in\Sigma, s\in Q\}$ if $q\neq q_0$, $\{a\,\#\,\dashv\}$ if $q=q_0$.

Thus G'' is a BCP grammar for L.

(b) To see that the containment is proper, consider $L(G_1)$ from § 1. $L(G_1)$ has been shown not to be LR-regular [2], yet clearly is BCP(1, 1) as demonstrated in § 1. □

Thus the ability to parse noncanonically significantly broadens the class of recognizable languages beyond the deterministic languages. It is obvious from the definitions that the BCP languages are contained within the FSPA languages. However, we have not been able to prove or disprove that this containment is proper.

We now turn to analyzing the power of the LR(k, ∞) languages. Our results are presented in a sequence of lemmas.

LEMMA 4.10. *Every* FSPA(k) *language is an* LR(k, ∞) *language.*

Proof. Recall that the FSPA(k) grammars were defined by a restriction applied to the LR(k, ∞) grammars. Hence every FSPA(k) grammar is an LR(k, ∞) grammar, and the lemma follows immediately. □

LEMMA 4.11. *The sets* $\{a^ncb^n | n \geqq 1\}$ *and* $\{a^ncb^{2n} | n \geqq 1\}$ *are not separable by any regular set.*

Proof. The proof is left to the reader.

LEMMA 4.12. *If a one-tape off-line Turing machine M performs all of its computations within time Kn, where K is a constant and n is the length of its input tape, then the set accepted by M is regular.*

Proof. This result was first shown by Hennie [6]. □

LEMMA 4.13. *There exist* LR(1, ∞) *languages which are not* FSPA(k) *for any value of k.*

Proof. Consider $L = \{a^ncb^n | n \geqq 1\} \cup \{a^ncb^{2n} | n \geqq 1\}$. L can easily be shown to have an LR(1, ∞) grammar. We will show that *no* grammar for L can be FSPA(k).

Suppose that some grammar G for L were FSPA(k) for some k. Let us analyze the properties of this grammar. Assume without loss of generality that G is reduced.

1. If A is a nonterminal of G which can generate infinitely many strings, then it must be the case that any terminal string produced by A contains a c. If this were not the case, some sentential form of G would contain a string such as wAy, where w and y are strings of terminals and y (say) contains a c. By letting A derive a string of length $2 \times |y|$, we produce a string which certainly is not in L.

2. If α is a sentential form of G, then there can be at most one nonterminal in α which generates an infinite set because otherwise we could produce a string with multiple c's.

3. Suppose that A is a nonterminal and that $A \overset{+}{\Rightarrow} a^mAb^n$ for some values of m and n. We claim that either $n = m$ or $n = 2m$. Using 1 above and the fact that G is reduced, there exist q, r, s, t such that $S \overset{+}{\Rightarrow} a^qAb^t$ and $A \overset{+}{\Rightarrow} a^rcb^s$. Therefore, since $A \overset{+}{\Rightarrow} a^{km}Ab^{kn}$ for all k, we conclude that $a^qa^{km}a^rcb^sb^{kn}b^t \in L(G) \forall k$. Examine the ratio of number of b's to number of a's in such a form. This ratio is $[(s+t)+kn]/[(q+r)+km]$ and must for any value of k be either 1 or 2. The only way that this can happen is for $n = m$ or for $n = 2m$.

4(a). Let $A \overset{+}{\Rightarrow} a^mAb^{2m}$ for some m. Then A never can occur in a sentential form α which derives a string in $\{a^ncb^n | n \geqq 1\}$.

Suppose otherwise. Thus $\alpha = \beta_1 A \beta_2$ and $a \overset{*}{\Rightarrow} a^n c b^n$ for some n. Then there exist q, r, s, t such that $\beta_1 \overset{*}{\Rightarrow} a^q$, $A \overset{*}{\Rightarrow} a^r c b^s$ and $\beta_2 \overset{*}{\Rightarrow} b^t$. Furthermore, $q + r = s + t$. Since $\beta_1 A \beta_2$ is a sentential form, so are $a^q A b^t$, $a^q a^{km} A b^{2km} b^t$, and $a^q a^{km} a^r c b^s b^{2km} b^t$. Examining the ratio of b's to a's in this latter sentence, we have

$$f = \frac{s + t + 2km}{q + r + km}.$$

In particular, letting $k = q + r$ and recalling that $s + t = q + r$, we have

$$f = \frac{(q+r) + 2(q+r)m}{(q+r) + (q+r)m} = \frac{1 + 2m}{m};$$

but then $2 > f > 1$ and the string could not have been in $L(G)$.

4(b). Similarly, if $A \overset{+}{\Rightarrow} a^m A b^m$ for some m, then no sentential form containing an A can ever derive a string of the form $a^n c b^{2n}$.

5. We can now partition the vocabularly of G into four subsets:

$V_1 = \{A \in V | L(G_A)$ is finite$\}$,

$V_2 = \{A \in V | L(G_A)$ is infinite but A is not recursive$\}$,

$V_3 = \{A \in V | A$ is recursive and $A \overset{+}{\Rightarrow} a^m A b^m$ for some $m\}$,

$V_4 = \{A \in V | A$ is recursive and $A \overset{+}{\Rightarrow} a^m A b^{2m}$ for some $m\}$.

6. There exists q such that any sentential form whose length is greater than q can only be derived using recursive nonterminals. Pictorially then, a tree for a long string looks like Fig. 4, where only elements of $V_2 \cup V_3$ occur along the "core" or else elements of $V_2 \cup V_4$ occur along the "core." All characters on the side branches are elements of V_1.

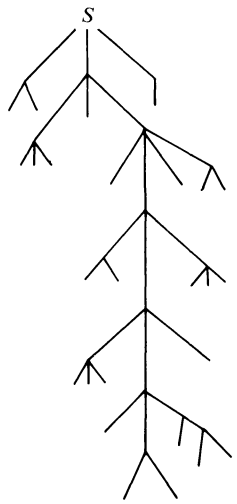


FIG. 4.

7. Suppose G is FSPA(k) with machine M (a 2PDA) as its parser. We will generate a two-stack acceptor M' which will simulate M on any input whose length is greater than q. However, the first time that M produces an element of V_3 or V_4, M' will halt. Furthermore, M' will indicate acceptance of its input if and only if the nonterminal which caused the termination is in V_3. On the other hand, if the input string has length $\leqq q$, then M' accepts it if and only if it is of the form $\{a^n cb^n \mid 1 \leqq n \leqq q/2\}$. Thus M' separates $\{a^n cb^n \mid n \geqq 1\}$ from $\{a^n cb^{2n} \mid n \geqq 1\}$. Furthermore, M' does this in linear time on a 2PDA.

8. M' can be simulated by a Turing machine M'' which also runs in linear time. When M'' makes a reduction such as $A \to \mathrm{BCD}$ on its tape, it will actually change the instance of BCD to Abb. Thus M'' does not attempt to "shrink" its tape as the two-stack device M' does. Since the side trees occurring on a derivation tree belonging to G are bounded in size (i.e., width) by some value p, we can never produce more than $2p$ blanks in a row before producing one of the nonterminals in V_3 or V_4. Thus, M'' runs slower than M' by at most a constant factor, namely, $2p$.

9. $T(M'')$ is a regular set by Lemma 4.12. However, since $T(M'') = T(M')$, we conclude that $T(M'')$ is a regular set which separates $\{a^n cb^n \mid n \geqq 1\}$ from $\{a^n cb^{2n} \mid n \geqq 1\}$.

This, of course, contradicts Lemma 4.11, so we conclude that G could not have existed in the first place. Hence no grammar for G is FSPA(k). □

The last four lemmas can be combined to give us the desired result as follows.

THEOREM 4.14. *The class of* LR(k, ∞) *languages properly includes the* FSPA(k) *languages.*

Proof. Inclusion follows from Lemma 4.10. The fact that this inclusion is proper is a consequence of Lemma 4.13. □

The relationships between the classes of languages induced by the parsing methods we have studied is depicted in Figure 5. Solid lines indicate proper containment, whereas dashed lines indicate a containment which has not yet been shown to be proper. It is interesting to note that the fairly "messy" lattice of inclusions of grammar classes shown in Fig. 3 collapses into a linear order when viewed in language space.

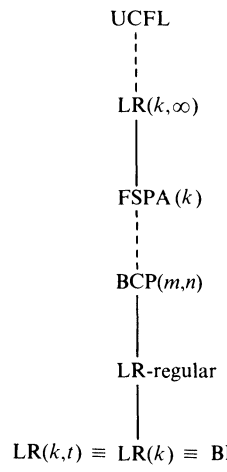


FIG. 5. *Inclusion diagram for some classes of Λ-free languages*

4. Conclusions and open problems. We have attempted to construct a general framework for bottom-up parsers and, within that framework, to examine the noncanonical extensions of some existing methods. It has been shown that the only noncanonical extension to enjoy both linear time parsability and decidability is the BCP technique. We intend to consider further restrictions to the FSPA method to see if we can achieve decidability with some class significantly more general than the BCP grammars. Even so, we conjecture that in language space, the two classes will prove to be identical, probably for the same reason that BRC languages are the same as LR(k) languages.

Additional extensions are possible within the current framework. Define a grammar G as being *regular pattern parsable* (RPP) if there exists a parsing scheme for G, each of whose reduction patterns is a regular set (possibly unbounded on both sides of the #). The class of RPP grammars is essentially the noncanonical extension of the LR-regular grammars. This class is nonrecursive because it contains the FSPA(k) grammars yet certainly is recursively enumerable (Theorem 3.8). Several interesting questions can now be asked. Does every unambiguous context-free language have an RPP grammar? Can RPP grammars be parsed in linear time and if so, what sort of computer is needed to achieve this bound?

We finally point out that the requirement that a parser be able to locate a phrase in *any* sentential form is overly restrictive. What is really desired is that every sentential form which arises *during parsing* be reducible. Is it possible to refine the theoretical framework of § 3 to incorporate this idea? As an example, one can easily construct a grammar G for $\{ww^R | w \in \{a, b\}^*\}$ and a parser for G which utilizes the order in which reductions are made to produce a parser while still only using a finite number of states.

Appendix. Grammars used in constructing the lattice of Fig. 3.

LEMMA A.1. G_1 *of the paper is* BCP *but not* LR(k, t) *or* LR-*regular.*

LEMMA A.2. G_2 *of the paper is* LR(k, t) *but not* LR-*regular.*

LEMMA A.3. G_3 *below is* LR($1, \infty$) *but not* FSPA(k).

$$
\begin{aligned}
G_3 : S &\to A|B \\
A &\to aA\bar{A}|a\bar{A} \\
B &\to aB\bar{B}|a\bar{B} \\
\bar{A} &\to b \\
\bar{B} &\to bb
\end{aligned}
$$

LEMMA A.4. G_4 *below is* LR-*regular and* FSPA(1) *but not BCP or* LR(k, t).

$$
\begin{aligned}
G_4 : S &\to aAa|bAb|aBb|bBa \\
A &\to \bar{A}A|c \\
B &\to \bar{B}B|c \\
\bar{A} &\to c \\
\bar{B} &\to c
\end{aligned}
$$

LEMMA A.5. *G_5 below is unambiguous but neither* LR(k, ∞) *nor* LR-*regular.*

$$G_5 : S \to aSa \mid bSb \mid aa \mid bb$$

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling*, vols. 1 and 2, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[2] K. CULIK, II AND R. COHEN, *LR-regular grammars—An extension of LR(k) grammars*, J. Comput. System Sci., 7 (1973), pp. 66–96.

[3] A. COLMERAUER, *Total precedence relations*, J. Assoc. Comput. Mach., 17 (1970), pp. 14–30.

[4] R. W. FLOYD, *Bounded context syntactic analysis*, Comm. ACM, 7 (1964), pp. 62–67.

[5] J. HARTMANIS, *Context free languages and Turing machine computations*, Proc. Symp. in Appl. Math., vol. 19, American Mathematical Society, Providence, R.I., 1967.

[6] F. C. HENNIE, *One-tape, off-line Turing machine computations*, Information and Control, 8 (1965), pp. 553–578.

[7] D. E. KNUTH, *On the translation of languages from left to right*, Ibid., 8 (1965), pp. 607–639.

[8] W. F. OGDEN, Unpublished memorandum, Dec. 1971.

[9] D. J. ROSENKRANTZ AND R. E. STEARNS, *Properties of deterministic top down grammars*, Information and Control, 17 (1970), pp. 226–256.

[10] T. G. SZYMANSKI, *Generalized bottom-up parsing*, Ph.D. thesis, Dept. of Computer Sci., Cornell Univ., Ithaca, N.Y., 1973.

[11] T. G. SZYMANSKI AND J. H. WILLIAMS, *Non-canonical parsing*, Proc. Symp. on Switching and Automata Theory, IEEE Computer Society, Iowa City, Ia., 1973, pp. 122–129.

[12] J. H. WILLIAMS, *Bounded context parsable grammars*, Tech. Rep. 72-127, Dept. of Computer Sci., Cornell Univ., Ithaca, N.Y., 1972.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

TERMINATION PROPERTIES OF GENERALIZED PETRI NETS*

Y. EDMUND LIEN†

Abstract. A generalization of Petri nets and vector addition systems, called GPN and MGPN, is introduced in this paper. Termination properties of this generalized formalism are investigated. Four subclasses of GPN's are considered. We distinguish forward-conflict-free, backward-conflict-free, forward-concurrent-free and backward-concurrent-free GPN's. We also study the concept of strongly connected, strongly repetitive and conservative GPN's. The main results obtained are (i) every strongly connected, strongly repetitive and forward- (or backward-) conflict-free GPN must be conservative, and (ii) every strongly connected, conservative and forward- (or backward-) concurrent-free GPN must be strongly repetitive. Since the class of forward-conflict-free GPN's contains properly the structures of computation graphs of Karp and Miller and marked graphs, some results appearing in these studies can be obtained as corollaries. Specialized versions of our results for the case of Petri nets are also included.

Key words. Petri net, vector addition system, vector replacement system, computation graph, marked graph, termination property

1. Introduction. Among models for parallel computation, Petri nets have been considered a simple but elegant formalism for concurrent and asynchronous events. Theoretical aspects of Petri nets have been presented mainly in the work of Petri [11], Holt [3], Holt and Commoner [4], Commoner, Holt, Even and Pnueli [1] and Genrich [2]. Some applications can be found in Shapiro and Saint [13], Patil [10] and Noe [12].

Another important model is the vector addition systems introduced by Karp and Miller [5]. Although developed independently, this model is closely related to Petri nets. The similarity of these two models has been observed by Keller [7]. Karp and Miller have also studied another similar structure called computation graph, which was introduced as a model for a restricted type of parallel computation. Keller generalized the concept of vector addition systems and introduced vector replacement systems [7]. Independently, Lien [8] presented the same generalized notion in a different formalism. This formalism was called transition systems in [9]. Transition systems can also be considered generalized Petri nets.

The central topic of this paper is on the termination properties of several subclasses of the generalized Petri nets. The results are an extension of the work in [1] and [6].

2. Generalized Petri nets (GPN). In order to study the structural aspects of Petri nets, we need to distinguish Petri nets and marked Petri nets. A *Petri net* is here defined to be a bipartite directed graph $\langle P \cup T, A\rangle$, where nodes in P are called *places* and nodes in T *transitions*. Arcs in A connect places to transitions or transitions to places. It is assumed that P, T and A are all finite sets. A *marked* Petri net is a triple $\langle P \cup T, A, x_0\rangle$ where $\langle P \cup T, A\rangle$ is a Petri net and x_0 is a nonnegative integer-valued function, which assigns for every place a number of *tokens*.[1]

* Received by the editors August 21, 1973, and in revised form February 21, 1975.

† Department of Computer Science, University of Kansas, Lawrence, Kansas 66045. This research was supported in part by Kansas General Research Grant 3298-5038.

[1] This definition is different from the ordinary way of defining Petri net. What is called a marked Petri net here is usually called a Petri net.

The notion of Petri nets can be generalized by introducing labeled arcs. However, we choose to present our generalized Petri nets in a different manner. Let $P=\{A_1, A_2, \cdots, A_n\}$ be a finite set of symbols. Let $+$ be a commutative and associative binary operation. We shall define a set P^* recursively as follows:

(i) Λ (called *empty state*) is in P^*,

(ii) $\forall i$, A_i is in P^*,

(iii) if x and y are in P^*, so is $x+y$.

Since $+$ is assumed to be commutative and associative, a distinct element x in P^* can be identified by the number of times each A_i $(1 \leqq i \leqq n)$ appears in x. In other words, elements in P^* can be represented in the general form $x_1A_1+x_2A_2+\cdots+x_nA_n$, where the x_i's $(1 \leqq i \leqq n)$ are nonnegative integers. We abbreviate this general form by $\sum_{i=1}^{n} x_iA_i$. If $x=\sum_{i=1}^{n} x_iA_i$ and $y=\sum_{i=1}^{n} y_iA_i$, then $x+y$ is the element $\sum_{i=1}^{n} (x_i+y_i)A_i$. We say that $x \leqq y$ if $x_i \leqq y_i$ for $1 \leqq i \leqq n$. When $x \leqq y$, $y-x$ denotes the element $\sum_{i=1}^{n} (y_i-x_i)A_i$. Λ is the element $\sum_{i=1}^{n} z_iA_i$, where $z_i(1 \leqq i \leqq n)$ is zero.

DEFINITION 1. A *generalized Petri net* (GPN) is a pair $\langle P, T\rangle$, where

(i) P is a finite set of symbols. Each symbol is called a *place*;

(ii) T is a finite subset of $P^* \times P^*$. Each element in T is called a *transition*.

An element (x, y) in T will be written as $x \to y$. Also, each transition will be given a name t_j, $1 \leqq j \leqq m$. In the sequel, we will assume that P is the set $\{A_1, A_2, \cdots, A_n\}$, T is the set $\{t_1, t_2, \cdots, t_m\}$ and each transition t_j is of the following form:

$$t_j: \sum_{i=1}^{n} a_{ij}A_i \to \sum_{i=1}^{n} b_{ij}A_i,$$

where $\sum_{i=1}^{n} a_{ij}A_i$ and $\sum_{i=1}^{n} b_{ij}A_i$ are sometimes referred to as $\mathrm{LHS}(t_j)$ and $\mathrm{RHS}(t_j)$, respectively. Although a transition can have Λ as its LHS and RHS, the removal of this type of transitions will not affect most of the properties we studied. In case we consider the size of the set T, it is assumed that no transition of the kind is included.

DEFINITION 2. A *marked* GPN (MGPN) is a triple $\langle P, T, x_0\rangle$, where

(i) $\langle P, T\rangle$ is a GPN, and

(ii) x_0 is an element in P^*, called the *initial state* of the MGPN.

Since a MGPN has a GPN as its underlying structure, most of the definitions related to GPN's can be extended naturally to MGPN's.

Starting from the initial state, an MGPN can enter into other states. To specify the rules of state transitions, we define the relation $\Rightarrow$ and $= * \Rightarrow$ on P^*. Each element in P^* is called a *state*. If x is a state and t_j is a transition such that $\mathrm{LHS}(t_j) \leqq x$, we say that t_j is *enabled* or *fireable* in state x, or write $t_j \in E(x)$. The action of firing t_j in state x results in a new state $y=x-\mathrm{LHS}(t_j)+\mathrm{RHS}(t_j)$. Here x and y are said to be $x \Rightarrow y$ or $x=t_j \Rightarrow y$.

A state y is said to be *reachable* from a state x, or $x = * \Rightarrow y$, if there exist a sequence of states $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \cdots, \boldsymbol{\alpha}_k$ and a sequence of transitions $\boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \cdots, \boldsymbol{\tau}_{k-1}$ such that $\boldsymbol{\alpha}_i=\boldsymbol{\tau}_i \Rightarrow \boldsymbol{\alpha}_{i+1}$ for $1 \leqq i \leqq k-1$, $\boldsymbol{\alpha}_1=x$ and $\boldsymbol{\alpha}_k=y$. We also say that $x=\boldsymbol{\tau}_1\boldsymbol{\tau}_2\cdots\boldsymbol{\tau}_{k-1} \Rightarrow y$, and the sequence of transitions is called a *firing sequence* in x, or simply a firing sequence.

An MGPN can fire an enabled transition to reach a new state. In a given state, many transitions may be fireable. We leave it unspecified as to which transition is actually fired. It is to be noted, however, that a transition firing may lead to a situation that some other transitions can never be fireable. This is one of the reasons for studying the termination properties of MGPN and GPN. Similar work has been done in a special class of MGPN [6], which we shall discuss later.

3. Graph representation and characteristic matrix. We can generalize the graph notation of Petri nets to GPN's. The generalization is a natural one. Each GPN corresponds to a labeled directed graph. Places are represented by circles and transitions by line segments. Each arc is associated with a positive integer. A place is an *input place* of a transition if there is an arc from the place to the transition. An *output place* is defined similarly.

Each place is capable of holding some number of tokens. The initial state of a MGPN can thus be represented by a token distribution.

Example 1. Let $P=\{A, B, C\}$, $T=\{t_1, t_2, t_3, t_4\}$, with

$$t_1: \quad A \to B,$$

$$t_2: \quad A \to C,$$

$$t_3: \quad 2B \to B+A,$$

$$t_4: \quad 2C \to C+A.$$

Let the initial state x_0 be $2A$. The graph representation is given in Fig. 1. A token is shown as a dot.

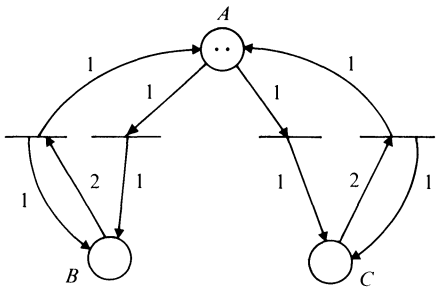


FIG. 1. *An* MGPN

The firing of a transition t_j is exhibited by first removing from each input place A_i of the transition some number of tokens specified by the integer associated with the arc from A_i to t_j, and then adding to each output place A_k tokens according to the number associated with the arc from t_j to A_k. We obtain another token distribution after firing a transition.

Here we shall adopt some basic definitions in graph theory. A sequence of nodes $n_1, n_2, \cdots, n_k$ is said to be a *path* between n_1 and n_k if there is an arc between n_i and n_{i+1} for $1 \leqq i \leqq k-1$. The path is called a *directed path* from n_1 to n_k if the orientation of the arcs is from n_i to n_{i+1} for $1 \leqq i \leqq k-1$. The directed path becomes a *directed circuit* if n_1 and n_k coincide. A directed graph is said to be *connected* if there is a path between any pair of nodes. A directed graph is *strongly*

connected if there is a directed path from any node to any node. These definitions also apply to bipartite directed graphs.

DEFINITION 3. A GPN is said to be *connected* (or *strongly connected*) if its graph representation is connected (or strongly connected).

We now turn to another important concept. Let P be the set of places $\{A_1, A_2, \cdots, A_n\}$ and T be transitions $\{t_1, t_2, \cdots, t_m\}$. Any element $x = \sum_{i=1}^{n} x_i A_i$ in P^* can be represented by an n-dimensional vector whose ith component is x_i. We shall denote this vector by $\tilde{x}$. The jth component of $\tilde{x}$ is indicated by $(\tilde{x})_j$. Let transition t_j be $\sum_{i=1}^{n} a_{ij}A_i \to \sum_{i=1}^{n} b_{ij}A_i$. The *characteristic matrix* $\mathbf{\Gamma}$ of a GPN is defined to be an $n \times m$ matrix such that $(\mathbf{\Gamma})_{ij} = b_{ij} - a_{ij}$.

We shall see in the later part of this paper that a GPN H is closely related to another GPN H^t whose characteristic matrix is the transpose of that of H. The *transpose* H^t of a GPN H is constructed from the graph representation of H as follows. First convert all the places in H into transitions and transitions into places. Next, reverse the direction of every arc. It is easy to see that the characteristic matrix of H^t is indeed the transpose of that of H.

4. Some preliminary results. In order to analyze the termination properties of generalized Petri nets, we need some definitions. Associated with each transition t_j is a function F_j. For every firing sequence $\boldsymbol{\sigma}$, $F_j(\boldsymbol{\sigma})$ gives the number of times that t_j appears in $\boldsymbol{\sigma}$. When t_j appears infinitely many times in $\boldsymbol{\sigma}$, $F_j(\boldsymbol{\sigma})$ is defined to be $\boldsymbol{\omega}$. For convenience, we say that $\boldsymbol{\omega}$ is larger than any integer. For any sequence $\boldsymbol{\sigma}$ of transitions, $F(\boldsymbol{\sigma})$ is defined to be a vector whose jth component is the value $F_j(\boldsymbol{\sigma})$. The next result follows from our definition.

LEMMA 1. (a) *If* $x = \boldsymbol{\sigma} \Rightarrow y$, *then* $x + \mathbf{\Gamma} \cdot F(\boldsymbol{\sigma}) = y$.

(b) *If* $\tilde{F}$ *is a nonnegative* m*-vector, then there exists a state* x *and a firing sequence* $\boldsymbol{\sigma}$ *in* x *such that* $F(\boldsymbol{\sigma}) = \tilde{F}$.

DEFINITION 4. A transition t_j is *terminating* in a state x if there is an integer k such that for every firing sequence $\boldsymbol{\sigma}$ in x, $F_j(\boldsymbol{\sigma}) \leqq k$. Otherwise, t_j is *nonterminating* in x.

A similar definition first appeared in [6]. The definition also appeared in Keller [7]. In fact, t_j is nonterminating in x is the same as warm$(\{t_j\}, x)$ of Keller.

A state x is said to be *terminating* if all transitions are terminating in x. Otherwise, we say that x is *nonterminating*.

The order of the quantifiers $(\exists k)$, $(\forall \boldsymbol{\sigma})$, $(\exists t_j)$, etc., in the previous definitions seems to be important.[2] In general, we have the following observations.

THEOREM 1. *The following three statements are equivalent*:

(A) *A state* x *is nonterminating, i.e.,*

$$\exists t_j, \forall k, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad F_j(\boldsymbol{\sigma}) > k.$$

(B) *There exists an infinite firing sequence in* x, *i.e.,*

$$\exists t_j, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall k, F_j(\boldsymbol{\sigma}) > k.$$

(C) *There are infinitely many firing sequences in* x, *i.e.,*

$$\forall k, \exists t_j, \exists \boldsymbol{\sigma} \quad \text{such that} \quad F_j(\boldsymbol{\sigma}) > k.$$

[2] It was kindly pointed out by the referees that the quantifiers $(\exists k)$ and $(\forall \boldsymbol{\sigma})$ in Definition 4 are not reversible. Theorem 2 is inspired by their suggestions.

Proof. The equivalence of (A) and (B) was shown in [9]. It was also shown independently in [7]. That (A) implies (C) is obvious. If every transition is terminating, say t_j, for $1 \leqq j \leqq m$, appears no more than k_j times in every firing sequence, then no firing sequence can be longer than $k_1 + k_2 + \cdots + k_m$. The number of firing sequences in x is thus finite. This concludes the proof that (C) implies (A). Q.E.D.

A state x is said to be *strongly nonterminating* if there is an infinite firing sequence $\boldsymbol{\sigma}$ in x such that for any transition t_j, any integer k, $F_j(\boldsymbol{\sigma}) > k$. (Again, this definition also appeared in [7]. That a state x is strongly nonterminating is the same as hot(T, x), where T is the set of all transitions.)

It is interesting to know that the order of the quantifiers in the previous definition makes a difference. Consider the following statements:

(D) A state x is strongly nonterminating, i.e.,

$$\exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall t_j \text{ and } \forall k,\ F_j(\boldsymbol{\sigma}) > k.$$

(E) In a state x every transition can be fired infinitely many times, i.e.,

$$\forall t_j,\ \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall k,\ F_j(\boldsymbol{\sigma}) > k.$$

(F) In a state x,

$$\forall k,\ \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall t_j,\ F_j(\boldsymbol{\sigma}) > k.$$

(G) Every transition is nonterminating in x, i.e.,

$$\forall t_j,\ \forall k,\ \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad F_j(\boldsymbol{\sigma}) > k.$$

The following theorem shows the interrelationship of these statements.

THEOREM 2. (i) (D) *implies* (E), *but the converse is not true.*

(ii) (E) *implies* (G), *but the converse is not true.*

(iii) (D) *implies* (F), *but the converse is not true.* (This is due to J. Bruno.)

(iv) (F) *implies* (G), *but the converse is not true.*

(v) *It is not true that* E *implies* F *or the converse.*

Proof. (i) It is obvious that (D) implies (E). The MGPN given in Example 1 is a counterexample to the converse. Let x be the initial state x_0. Note that $(t_1)(t_1t_3)^k$, for an arbitrarily large k, is a firing sequence in x. Here we use $\boldsymbol{\sigma}^k$ to denote the concatenation of k identical copies of $\boldsymbol{\sigma}$. Also $(t_2)(t_2t_4)^k$, for an arbitrarily large k, is a firing sequence. However t_3 and t_4 can never appear in the same firing sequence in x. Hence x is not strongly nonterminating.

(ii) Again, (E) implies (G) is trivial. For the converse, we have the following counterexample. Let P be $\{A, B, C\}$ and T be the following:

$$t_1: \quad 2A \to 2A + B,$$

$$t_2: \quad A \to C,$$

$$t_3: \quad B + 2C \to A + C.$$

Let x be $2A$. For any k, $(t_1)^{k+1}(t_2)^2(t_3t_2)^{k+1}$ is a firing sequence in x which fires t_1, t_2 and t_3 for more than k times. Hence no transition is terminating. On the other hand, once t_2 is fired, the number of times that t_2 or t_3 can be fired is bounded by the number of times that t_1 has been fired. Therefore, there is no firing sequence in x in which t_2 or t_3 appears infinitely many times.

(iii) That (D) implies (F) is trivial. The example in the proof of (ii) is a counterexample to the converse. The proof of (iii) is first due to J. Bruno, as reported in [7]. A different example was given there.

(v) For Example 1, (E) is true and (F) is not. For the MGPN in the proof of (ii), (F) is true and (E) is not.

(iv) Again, (F) implies (G) is trivial. Since (E) implies (G), "(G) implies (F)" contradicts (v). Hence it is not true that G implies F. Q.E.D.

It is conjectured that (E) and (F) is a necessary and sufficient condition that a state x is strongly nonterminating. We leave this for the future investigation.

According to Theorem 1, a nonterminating state x has an infinite firing sequence $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, \boldsymbol{\sigma}_3, \cdots$, where $\boldsymbol{\sigma}_i$ denotes a transition. Correspondingly, there exists an infinite sequence of states $s_0, s_1, s_2, \cdots$, such that $x = s_0$, and $s_{i-1} = \boldsymbol{\sigma}_i \Rightarrow s_i$ for $i \geqq 1$. For our next result, we need the following lemma from [5].

LEMMA 2. *Let $\tilde{s}_0, \tilde{s}_1, \tilde{s}_2, \cdots$ be an infinite sequence of n-dimensional vectors of nonnegative integers. There must exist an infinite subsequence $\tilde{s}_{i_1}, \tilde{s}_{i_2}, \tilde{s}_{i_3}, \cdots$ such that $\tilde{s}_{i_j} \leqq \tilde{s}_{i_{j+1}}$. The relation $\leqq$ is a componentwise comparison.*

Hence, if x is nonterminating, there must exist two states s_i and s_j such that $s_i \leqq s_j$. A necessary and sufficient condition for a state to be nonterminating can thus be given as follows.

THEOREM 3. *A state x is nonterminating if and only if there exist two states y and z such that $y \leqq z$, $x = * \Rightarrow y$ and $y = \boldsymbol{\sigma} \Rightarrow z$ for some nonempty $\boldsymbol{\sigma}$.*

COROLLARY 1. *A transition t_j can be fired infinitely many times in a state x if and only if there exist states y and z such that $y \leqq z$, $x = * \Rightarrow y$ and $y = \boldsymbol{\sigma} \Rightarrow z$ with $F_j(\boldsymbol{\sigma}) \geqq 1$.*

COROLLARY 2. *A state x is strongly nonterminating if and only if there exist states y and z such that $y \leqq z$, $x = * \Rightarrow y$ and $y = \boldsymbol{\sigma} \Rightarrow z$, where $\boldsymbol{\sigma}$ is a firing sequence with $F_j(\boldsymbol{\sigma}) \geqq 1$ for every transition t_j.*

Whether a transition is terminating in a state x depends not only on x but also on the structure of the GPN. In order to study the structure of a GPN and its effect on termination properties of transitions, we introduce the following definitions. The termination properties so established are therefore independent of the choice of initial states.

DEFINITION 5. A transition t_j in a GPN is *terminating* if for every state x, t_j is terminating in x. t_j is *nonterminating* if there is a state x such that t_j is nonterminating in x.

DEFINITION 6. A GPN $\langle P, T \rangle$ is said to be *nonterminating* (or *strongly nonterminating*) if there is a state x in P^* such that x is nonterminating (or strongly nonterminating) with respect to $\langle P, T \rangle$.

Therefore a strongly nonterminating GPN has the potential to fire every transition arbitrarily many times, provided an initial state is properly chosen. The following result relates the termination properties of a GPN to the properties of each individual transition.

Consider the following statements.

(D′) A GPN is strongly nonterminating, i.e.,

$$\exists x, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall t_j, \forall k, F_j(\boldsymbol{\sigma}) > k.$$

(E′a) $$\exists x, \forall t_j, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall k, F_j(\boldsymbol{\sigma}) > k.$$

(E′b) $$\forall t_j, \exists x, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall k, F_j(\boldsymbol{\sigma}) > k.$$

(F′) $$\exists x, \forall k, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad \forall t_j, F_j(\boldsymbol{\sigma}) > k.$$

(G′a) $$\exists x, \forall t_j, \forall k, \exists \boldsymbol{\sigma} \text{ in } x \quad \text{such that} \quad F_j(\boldsymbol{\sigma}) > k.$$

(G′b) Every transition is nonterminating, i.e.,

$$\forall t_j, \exists x \quad \text{such that} \quad \forall k, \exists \boldsymbol{\sigma} \text{ in } x \text{ with } F_j(\boldsymbol{\sigma}) > k.$$

It is not difficult to see that quantifiers $(\exists x)$ and $(\forall t_j)$ are reversible when appeared in tandem. We shall use (E′) to represent both (E′a) and (E′b) and (G′) for (G′a) and (G′b).

THEOREM 4. *The statements* (D′), (E′), (F′) *and* (G′) *are equivalent.*

Proof. It is clear that (D′) implies (E′) and (F′), (E′) or (F′) implies G′. it remains to show that (G′) implies (D′). Suppose that in a state x every transition t_j appears exactly k ($k \geqq 1$) times in a finite firing sequence $\boldsymbol{\sigma}_{jk}$. Let $x = \boldsymbol{\sigma}_{jk} \Rightarrow y_{jk}$. By Lemma 2, we find from $y_{j1}, y_{j2}, \cdots$ an infinite sequence $y_{jk_1}, y_{jk_2}, \cdots$ such that $y_{jk_i} \leqq y_{jk_{i+1}}$ for all $i \geqq 1$. Note that $k_i < k_{i+1}$ for all $i \geqq 1$. For each firing sequence $\boldsymbol{\sigma}_{jk_i}$ we construct an m-dimensional vector $\tilde{\boldsymbol{\sigma}}_{jk_i}$ such that $F_l(\boldsymbol{\sigma}_{jk_i})$ appears in the lth component. By Lemma 2 again, we can find two vectors $\tilde{\boldsymbol{\sigma}}_{jp}$ and $\tilde{\boldsymbol{\sigma}}_{jq}$ such that $p < q$ and $\tilde{\boldsymbol{\sigma}}_{jp} \leqq \tilde{\boldsymbol{\sigma}}_{jq}$. Note that $\tilde{\boldsymbol{\sigma}}_{jp} \neq \tilde{\boldsymbol{\sigma}}_{jq}$ since they must differ in jth component. Take $\tilde{\boldsymbol{\tau}}_j$ to be $\tilde{\boldsymbol{\sigma}}_{jq} - \tilde{\boldsymbol{\sigma}}_{jp}$. We must have $\boldsymbol{\Gamma} \cdot \tilde{\boldsymbol{\tau}}_j = \tilde{y}_{jq} - \tilde{y}_{jp} \geqq 0$, where $\boldsymbol{\Gamma}$ is the characteristic matrix of the GPN. For each transition t_j we can find such a nonnegative vector $\tilde{\boldsymbol{\tau}}_j$ such that $\boldsymbol{\Gamma} \cdot \tilde{\boldsymbol{\tau}}_j \geqq 0$ and the jth component of $\tilde{\boldsymbol{\tau}}_j$ is nonzero. Consider the componentwise summation $\tilde{\boldsymbol{\tau}} = \Sigma \tilde{\boldsymbol{\tau}}_j$ over all transitions. By Lemma 1(b), there must exist a state x', a firing sequence $\boldsymbol{\tau}$ in x' such that for every j, $F_j(\boldsymbol{\tau}) = (\tilde{\boldsymbol{\tau}})_j \geqq 1$. Since $\boldsymbol{\Gamma} \cdot \boldsymbol{\tau} \geqq 0$, by Theorem 3, $(\boldsymbol{\tau})^k$ for any k is a firing sequence in x'. Q.E.D.

COROLLARY 1. *A transition t_j is nonterminating in a* GPN *if and only if there is a state x in which t_j can be fired infinitely many times.*

COROLLARY 2. *A* GPN *is strongly nonterminating if and only if $\boldsymbol{\Gamma} \cdot y \geqq 0$ has a positive solution, that is, $(y)_j \geqq 1$ for each j.*

When compared to Theorem 2, the previous result reveals that the choice of an initial state can be very critical, for it may result in different termination properties. Basically, we have identified those GPN's which have the potential to have strongly nonterminating initial states.

5. Strongly connected GPN. When MGPN's are used to model systems of concurrent events, two major features are of concern. The first feature is the termination property and the other is the finiteness property. Karp and Miller presented a model for parallel computation in [6], which is basically a restricted form of MGPN except for some minor deviations. In their model, a transition describes a computation step and a place is viewed as a queue which holds data items (corresponding to tokens) to be processed by a computation step. They have

investigated the problems of determining whether a certain computation step will become active and whether a certain queue has an adequate storage capacity.

We observe that a GPN with a strongly nonterminating state will mean that no computation step will be indefinitely blocked as a result of other computation steps if computation steps are carefully scheduled. In an MPGN with the initial state x_0, let the set of states reachable from x be called the *reachability set* of the MPGN. We also observe that when the reachability set is finite, no queue will have the danger of "explosion", i.e., requiring an infinite storage space to hold data items. Therefore, it is reasonable to assume that many computation structures described in this type of framework can be modeled by MGPN's with a strongly nonterminating initial state and a finite reachability set. In this section, we consider these two features in the context of GPN's.

Based on the same reason for introducing Definition 6, we define finite GPN's.

DEFINITION 7. A GPN is *finite* if for every state x the reachability set $\{y | x = * \Rightarrow y\}$ with respect to x is finite.

It is conceivable that some MGPN $\langle P, T, x\rangle$ may have a finite reachability set and yet the underlying GPN $\langle P, T\rangle$ is not finite.

For convenience we shall say that a GPN is *well-behaved* if it is both strongly nonterminating and finite. The following necessary and sufficient condition for a GPN to be well-behaved was given in [8] and [9].

THEOREM 5. *A GPN is well-behaved if and only if* $\Gamma \cdot \mathbf{g} = 0$ *and* $\mathbf{f} \cdot \Gamma = 0$ *have positive integer solutions for variables* $\mathbf{g}$ *and* $\mathbf{f}$. Γ *is the characteristic matrix of the* GPN.

When there exists a positive integer vector $\mathbf{f}$ with $\mathbf{f} \cdot \Gamma = 0$, we say that the GPN in question is *conservative*. The vector $\mathbf{f}$ is called a *weight function* which assigns a weight $(\mathbf{f})_i$ for a token in place A_i. The total weight of a MGPN in a state $x = \sum_{i=1}^{n} x_i A_i$ is defined to be $\sum_{i=1}^{n} x_i (\mathbf{f})_i$, or $\mathbf{f}(x)$ for short. Note that the total weight of a conservative MGPN in every state of the reachability set is a constant.

When a GPN is conservative, it must be finite. If a state x of the GPN is also strongly nonterminating, then x must reach a state which can return to itself, after a nonvoid sequence of transition firings. Thus we consider the following definition.

DEFINITION 8. A state x is *repetitive* if $x = \boldsymbol{\sigma} \Rightarrow x$ for a nonempty sequence $\boldsymbol{\sigma}$. If $\boldsymbol{\sigma}$ contains every transition, then x is said to be *strongly repetitive*.

DEFINITION 9. A GPN $\langle P, T\rangle$ is *repetitive* (or *strongly repetitive*) if there is a state in P^* which is repetitive (or strongly repetitive).

It is easy to show that a GPN is strongly repetitive if and only if $\Gamma \cdot \mathbf{g} = 0$ has a positive solution.

It was also proved in [8] and [9] that a conservative, strongly repetitive and connected GPN must be strongly connected. However, not every strongly connected and conservative GPN is strongly repetitive. Such an example is easy to find. We therefore turn our attention to a restricted class of strongly connected GPN's. It is hoped that the study of this class of GPN's can reveal the graph-theoretic properties of well-behaved GPN's.

We consider conservative and strongly connected GPN's which are minimal in the sense that the removal of any transition from a GPN in this class results in a GPN which is not strongly connected.

DEFINITION 10. A GPN $\langle P, T\rangle$ is said to be *primal* if (i) it is conservative, (ii) it is strongly connected, and (iii) there exists no proper subset T' of T such that $\langle V', T'\rangle$ is strongly connected, where V' consists of all places which appear in the transitions in T'.

Condition (iii) in the previous definition ensures that the GPN considered is "minimally" strongly connected. It is easy to find a strongly connected GPN which is not minimally strongly connected.

It is interesting to know that statements (D), (E), (F) and (G) are all equivalent with respect to primal MGPN's. In fact, a state which is nonterminating must be strongly nonterminating. Essentially, we have the following results.

THEOREM 6. *Let $H=\langle P, T\rangle$ be a primal* GPN *and* $\mathbf{\Gamma}$ *be its characteristic matrix. If H is nonterminating, then*

(i) *H is strongly repetitive,*

(ii) *if $\mathbf{\Gamma}\cdot\mathbf{g}_1=0$ and $\mathbf{\Gamma}\cdot\mathbf{g}_2=0$, then $\mathbf{g}_1$ and $\mathbf{g}_2$ are linearly dependent, and*

(iii) $|P|\geqq|T|$.

Proof. (i) If H is conservative, then the reachability set $\{y|x=*\Rightarrow y\}$ with respect to any state x must be finite. If H is also nonterminating, then H must be repetitive. Suppose H is not strongly repetitive. We can find a state x such that $x=\boldsymbol{\sigma}\Rightarrow x$, where $\boldsymbol{\sigma}$ contains some transitions in T but not all. Let T' be the set of transitions included in $\boldsymbol{\sigma}$. Let P' be the set of places present in the transition in T'. We observe that $\langle P', T'\rangle$ is conservative and strongly repetitive. Therefore, $\langle P', T'\rangle$ is strongly connected, which violates the condition (iii) in Definition 10.

(ii) Assume to the contrary that $\mathbf{g}_1$ and $\mathbf{g}_2$ are linearly independent. We consider two cases.

(a) $(\mathbf{g}_1)_i=1$ for $1\leqq i\leqq|T|$. Let $\min\{(\mathbf{g}_2)_i|1\leqq i\leqq|T|\}=(\mathbf{g}_2)_j=h$. Define a new vector $\mathbf{g}$ to be $(\mathbf{g})_i=(\mathbf{g}_2)_i-h(\mathbf{g}_1)_i$. Note that $\mathbf{\Gamma}\cdot\mathbf{g}=0$ and $(\mathbf{g})_j=0$. With the same argument in (i), we conclude that H is not primal, a contradiction.

(b) Suppose not every component of $\mathbf{g}_1$ (or $\mathbf{g}_2$) is 1. We will transform H into another GPN H' with the same graph representation except the positive integers associated with the arcs in the graph. Note that merely altering these integers cannot change a nonprimal but conservative GPN into a primal one. If t_j in H is

$$\sum_{i=1}^{n} a_{ij}A_i \rightarrow \sum_{i=1}^{n} b_{ij}A_i,$$

then H' has t_j' as follows:

$$\sum_{i=1}^{n} [(\mathbf{g}_1)_j\cdot a_{ij}]A_i \rightarrow \sum_{i=1}^{n} [(\mathbf{g}_1)_j\cdot b_{ij}]A_i.$$

Let $\mathbf{\Gamma}'$ be the characteristic matrix of H'. Define two vectors $\mathbf{h}_1$ and $\mathbf{h}_2$ as follows:

$$(\mathbf{h}_1)_i=1 \quad \text{for } 1\leqq i\leqq|T|,$$

$$(\mathbf{h}_2)_i=[(\mathbf{g}_2)_i/(\mathbf{g}_1)_i]\cdot l \quad \text{for } 1\leqq i\leqq|T|,$$

where l is the least common multiple of $\{(\mathbf{g}_1)_i|1\leqq i\leqq|T|\}$. It should be clear that $\mathbf{\Gamma}'\cdot\mathbf{h}_1=0$, $\mathbf{\Gamma}'\cdot\mathbf{h}_2=0$ and $\mathbf{h}_1$ and $\mathbf{h}_2$ are linearly independent. We have reduced the

problem to case (a). The result in (a) shows that H' is not primal. Hence H is not primal. We have a contradiction.

(iii) We have shown in (ii) that the column nullity[3] of Γ is 1. Thus rank $(\Gamma) = |T| -$ column nullity $(\Gamma) = |T| - 1$. Also $|P| =$ rank $(\Gamma) +$ row nullity (Γ). It is known that H is conservative, hence row nullity $(\Gamma) \geqq 1$. In other words, $|P| \geqq |T|$. Q.E.D.

6. Some subclasses of primal GPN's. The purpose of this section is to introduce some special types of GPN's and study their termination properties. We distinguish several classes of GPN's by the structure of their graph representations.

In the following definition, transitions and places are nodes in the bipartite directed graph representation of a GPN. *Indegree* (or *outdegree*) of a node is defined to be the number of arcs entering (or leaving) the node.

DEFINITION 11. Let H be a GPN.

(i) H is said to be *forward-conflict-free* if every place has outdegree = 1.

(ii) H is said to be *backward-conflict-free* if every place has indegree = 1.

(iii) H is said to be *forward-concurrent-free* if every transition has indegree = 1.

(iv) H is said to be *backward-concurrent-free* if every transition has outdegree = 1.

The concept in Definition 11 first appeared in [4]. However, no results related directly to these types of Petri nets were reported there.

For convenience, we shall use ff as an abbreviation for forward-conflict-free, bf for backward-conflict-free, fc for forward-concurrent-free, and bc for backward-concurrent free.

DEFINITION 12. A GPN is said to be *conflict-free* if it is both ff and bf. A GPN is *concurrent-free* if it is both fc and bc.

Much of the work on Petri net or its generalized form is concerned with conflict-free or concurrent-free cases. The computation graphs presented in [6] are essentially conflict-free GPN's. Many interesting results for conflict-free Petri nets, which are termed *marked graphs*, were presented in [1], [2] and [4]. Concurrent-free Petri nets were considered in [4] also. Results on other classes of Petri nets can be found in [14].

We first present an elementary result which can be easily derived from our definitions.

THEOREM 7. (i) *The transpose of an* ff GPN *is* fc.

(ii) *The transpose of a* bf GPN *is* bc.

(iii) *The transpose of a conflict-free* GPN *is concurrent-free.*

We can also see that the transpose of an fc GPN is ff, the transpose of a bc GPN is bf, etc. It is straightforward that the transpose of a strongly connected GPN is also strongly connected. In addition, a GPN is conservative if and only if its transpose is strongly repetitive. These simple observations, together with Theorem 7, suggest that many subsequent results for one class of GPN's can be

[3] The column nullity is the maximum number of linearly independent vectors in the set $\{g | \Gamma \cdot g = 0\}$. The column rank of Γ is the maximum number of linearly independent column vectors of Γ. The row nullity and row rank are defined in the same way. It is known that column rank (Γ) = row rank (Γ). Hence the number is called the rank of Γ.

presented in a dual form for their transposes. In fact, a later theorem is found easier to prove in its dual form than in the original form. In the rest of this section, the main stream of presentation will be on ff GPN's, although occasionally we may mention other classes.

The first theorem on ff GPN's is concerned with the structure of the reachability sets.

THEOREM 8. *Let* $G=\langle P, T, x\rangle$ *be an* ff MGPN. *If* $x=\boldsymbol{\sigma}\Rightarrow y$ *and* $x=\boldsymbol{\tau}\Rightarrow z$, *then there must exist a state* w *such that* $y=\boldsymbol{\alpha}\Rightarrow w$, $z=\boldsymbol{\beta}\Rightarrow w$ *for some firing sequences* $\boldsymbol{\alpha}$ *and* $\boldsymbol{\beta}$ *with* $F_j(\boldsymbol{\alpha})=\max(F_j(\boldsymbol{\sigma}), F_j(\boldsymbol{\tau}))-F_j(\boldsymbol{\sigma})$ *and* $F_j(\boldsymbol{\beta})=\max(F_j(\boldsymbol{\sigma}), F_j(\boldsymbol{\tau}))-F_j(\boldsymbol{\tau})$ *for every transition* t_j *in* T.

Proof. Since G is ff, a transition once enabled remains enabled as long as the firing of the transition does not take place. In particular, if t and t' are in $E(x)$, then both tt' and $t't$ are valid firing sequences in x. Suppose $\boldsymbol{\sigma}=\boldsymbol{\sigma}_1\boldsymbol{\sigma}_2\cdots\boldsymbol{\sigma}_l$, where $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, \cdots, \boldsymbol{\sigma}_l$ are transitions in T. We use ${}_1\boldsymbol{\sigma}_k$ to denote the sequence $\boldsymbol{\sigma}_1\boldsymbol{\sigma}_2\cdots\boldsymbol{\sigma}_k$. Assume that q is the greatest integer such that $F({}_1\boldsymbol{\sigma}_q)\leqq F(\boldsymbol{\tau})$ but there exists a transition t_k such that $F_k({}_1\boldsymbol{\sigma}_{q+1})>F_k(\boldsymbol{\tau})$. We can inductively show that $\boldsymbol{\sigma}_{q+1}\in E(z)$. Let $z=\boldsymbol{\sigma}_{q+1}\Rightarrow z_1$. Similarly, one can find a transition $\boldsymbol{\sigma}_{p+1}$ such that $\boldsymbol{\sigma}_{p+1}\in E(z_1)$ and $F({}_1\boldsymbol{\sigma}_p)\leqq F(\boldsymbol{\tau}\boldsymbol{\sigma}_{q+1})$ but $F_{k'}({}_1\boldsymbol{\sigma}_{p+1})>F_{k'}(\boldsymbol{\tau}\boldsymbol{\sigma}_{q+1})$ for some k'. Iteratively, a firing sequence $\boldsymbol{\beta}=\boldsymbol{\sigma}_{q+1}\boldsymbol{\sigma}_{p+1}\cdots$ can be constructed such that $z=\boldsymbol{\beta}\Rightarrow w$ for some state w and $F_j(\boldsymbol{\beta})=\max(F_j(\boldsymbol{\sigma}), F_j(\boldsymbol{\tau}))-F_j(\boldsymbol{\tau})$ for $1\leqq j\leqq|T|$. The firing sequence $\boldsymbol{\alpha}$ can be established in the same manner. Q.E.D.

COROLLARY 1. *Let* $G=\langle P, T, x\rangle$ *be an* ff MGPN. *If the initial state* x *is repetitive, then any state in the reachability set is repetitive.*

COROLLARY 2. *If the initial state* x *of an* ff MGPN *is strongly repetitive, then any state* y *in the reachability set is strongly repetitive and* $y=*\Rightarrow x$.

If no transition is fireable in a state, we say the state is a *dead* state.

COROLLARY 3. *If the reachability set of an* ff MGPN *contains a dead state* y, *then every state* z *in the reachability set must be terminating and* $z=*\Rightarrow y$.

Since every conflict-free MGPN is also ff, the previous result also applies to the conflict-free case. The specialized version of the result in the case of marked graphs was obtained in [1] and [4].

The second theorem on ff GPN's shows that any strongly connected ff GPN automatically satisfies condition (iii) of Definition 10.

THEOREM 9. *If* $H=\langle P, T\rangle$ *is a strongly connected* ff GPN, *then there exists no other strongly connected* GPN $H_0=\langle P_0, T_0\rangle$ *with* $P_0\subseteq P$ *and* $T_0\subseteq T$.

Proof. Assume that $P_0\subseteq P$, $T_0\subseteq T$ and $\langle P_0, T_0\rangle$ is a strongly connected ff GPN. Let t_1 be a transition in T_0. The proof proceeds by cases.

Case 1. Every output place of t_1 is also its input place. T must contain only one transition or else there is no directed path from t_1 to other transitions. Therefore $\langle P_0, T_0\rangle$ is $\langle P, T\rangle$.

Case 2. At least one output place A_1 of t_1 is not its input place. Let A_1 be an input place of t_2. Since H is ff, t_2 must be in T_0. If each output place of t_2 is also its input place, H can not be strongly connected. By induction, we can show that for any transition t_j if there is a directed path from t_1 to t_j, t_j must be included in T_0. This implies that $T=T_0$. Hence $\langle P_0, T_0\rangle=\langle P, T\rangle$. Q.E.D.

COROLLARY. *Let* H *be strongly connected and* ff. *If* x *is a repetitive state in* H, *then* x *is strongly repetitive. If* $x=\boldsymbol{\sigma}_1\Rightarrow x$ *and* $x=\boldsymbol{\sigma}_2\Rightarrow x$ *for some nonempty sequences* $\boldsymbol{\sigma}_1$ *and* $\boldsymbol{\sigma}_2$, *then* $F(\boldsymbol{\sigma}_1)$ *and* $F(\boldsymbol{\sigma}_2)$ *must be linearly dependent.*

The third theorem on ff GPN's is concerned with their conservation property.

THEOREM 10. *Let H be a strongly connected and* ff GPN. *If H is (strongly) repetitive, then it is conservative.*

We shall prove Theorem 10 by showing a dual theorem in fc GPN's, namely, every strongly connected, conservative and fc GPN must be strongly repetitive. Two immediate lemmas are required.

LEMMA 3. *If $H=\langle P, T\rangle$ is a primal and* fc GPN, *then H must be strongly repetitive.*

Proof. Let $\mathbf{f}$ be a weight function of H. Since H is fc, every transition has only one input place. Define a state $D=\sum_{i=1}^{n} d_i A_i$ as follows. For every place A_i, $d_i = \min\{a_{ij} | \exists t_j, \mathrm{LHS}(t_j)=a_{ij}A_i\}-1$. Hence D is the "maximum" dead state in the sense that $D_1 \leqq D$ for any dead state D_1. Take a sufficiently large integer k such that $k>1$ and define a state x as $\sum_{i=1}^{n}(k\cdot d_i+1)A_i$. State x must be nonterminating, since $\mathbf{f}(x)>\mathbf{f}(D)$ and H is conservative. Furthermore, when H is conservative and nonterminating, it must be repetitive. Since H is also primal, it must be strongly repetitive. Q.E.D.

COROLLARY. *If $H=\langle P, T\rangle$ is primal and* fc, *then it is also* ff.

Proof. Since H is strongly connected and fc, we have $|T|\geqq|P|$. But H is primal and repetitive, we also have $|T|\leqq|P|$, by Theorem 6. Hence $|P|=|T|$. No two transitions can share a common input place. Q.E.D.

The next lemma gives a sufficient condition for an fc GPN to be primal.

LEMMA 4. *If $H=\langle P, T\rangle$ is a conservative, strongly connected and* fc GPN *with $|P|=|T|$, then H is primal.*

Proof. Since $|P|=|T|$ and every place is an input place of some transition, H must be ff. By Theorem 9, H is primal. Q.E.D.

Now we are ready to prove the main theorem in this section. The following theorem is the dual version of Theorem 10.

THEOREM 11. *Let $H=\langle P, T\rangle$ be a strongly connected* fc GPN. *If H is conservative, then it is strongly repetitive.*

Proof. We prove that H is strongly repetitive by showing that every transition in H is nonterminating. We will show that every transition t is in a primal and fc GPN $H_0=\langle P_0, T_0\rangle$ such that $P_0\subseteq P$ and $T_0\subseteq T$. Consider the graph representation of H. If $\mathrm{LHS}(t)=\mathrm{RHS}(t)$, the proof is easy. Otherwise, repeat the following steps.

Step 1. Find a directed circuit which contains at least two places and t.

Step 2. Let P_0 be the set of all places in the circuit and T_0 be the transitions in the circuit.

Step 3. If all places appearing in transitions in T_0 are already in P_0, terminate the process. Otherwise, go to the next step.

Step 4. Let A be an output place of a transition in T_0 but not an input place of any transition in T_0. Since H is strongly connected and fc, there must exist a directed path from A to some place A' in P_0 such that all intermediate nodes, places or transitions, are not in P_0 or T_0. Add all places in the path, including A, to V_0 and all transitions in the path to T_0. Go to Step 3.

When the process terminates, $\langle P_0, T_0\rangle$ is strongly connected. Step 4 also guarantees that $|P_0|=|T_0|$. By Lemmas 3 and 4, $\langle P_0, T_0\rangle$ is primal and strongly repetitive. Therefore, H must be strongly repetitive. Q.E.D.

COROLLARY. *Let* $H=\langle P, T\rangle$ *be a strongly connected* bc GPN. *If* H *is conservative, then it is strongly repetitive.*

Proof. Reversing the direction of all arcs in the graph representation of H, we obtain an fc GPN. Q.E.D.

Theorems 10 and 11 characterize the well-behaved ff, bf, fc and bc GPN's. In the remainder of this section, we shall investigate the interrelationship of these four classes of GPN's.

If a strongly connected GPN is ff, bf, fc and bc, its graph representation must correspond to a directed circuit. Let C be a directed circuit of $2k$ nodes in a graph representation of a GPN. The gain q_C is defined in [6] to be the number $b_1 \cdot b_2 \cdot \cdots \cdot b_k / a_1 \cdot a_2 \cdot \cdots \cdot a_k$ where the a_i's are associated with arcs from places to transitions in C and the b_i's are associated with those from transitions to places. The following result can be easily established.

LEMMA 5. *Let* C *be a connected, conflict-free and concurrent-free* GPN.

(i) C *is conservative if and only if* $q_c = 1$.

(ii) C *is strongly repetitive if and only if* $q_c = 1$.

(iii) *If* $q_c > 1$ (*or* $q_c < 1$), *there exists a state* x *such that* $x = \boldsymbol{\sigma} \Rightarrow y$ *for some sequence* $\boldsymbol{\sigma}$ *and* $x \lneqq y$ (*or* $x \gneqq y$ *respectively*).[4]

When we consider a directed circuit in any GPN graph representation, we also say that the circuit is ff (or bf) if the outdegree (or indegree) of every place in the circuit with respect to the entire graph is 1. Similarly, we define fc and bc circuits.

THEOREM 12. *Let* H *be a well-behaved* GPN. *Let* h *be a directed circuit in the graph representation of* H.

(i) *If* h *is* ff *or* fc, *then* $q_h \leqq 1$.

(ii) *If* h *is* bf *or* bc, *then* $q_h \geqq 1$.

(iii) *If* $q_h = 1$, *then* h *is both* ff *and* bf *or neither, and* h *is both* fc *and* bc *or neither.*

(iv) *If* $q_h \neq 1$, *then* h *is neither conflict-free nor concurrent-free.*

Proof. (i) Suppose h is ff and $q_h > 1$. By Lemma 5, the GPN H_0 formed by the places and transitions in h is not conservative nor repetitive. Moreover, we can find a state x_0 for H_0 such that $x_0 = \boldsymbol{\sigma} \Rightarrow y_0$ with $x_0 \lneqq y_0$. Let x_1 be a strongly repetitive state of H such that $x_1 = \boldsymbol{\tau} \Rightarrow x_1$. By Lemma 1, we can find a state x_2, a sequence $\boldsymbol{\alpha}$ in x_2 such that $F(\boldsymbol{\alpha}) = F(\boldsymbol{\sigma}) + F(\boldsymbol{\tau})$. Suppose $x_2 = \boldsymbol{\alpha} \Rightarrow y_2$; we must have $x_2 \lneqq y_2$. This contradicts the conservation property. Since the transpose of an fc GPN is ff, the proof for fc case can be reduced to ff case.

(ii) This case is similar to (i).

(iii) Suppose $q_h = 1$ and h is ff but not bf. The new tokens added to the circuit by firing transitions not in h will increase the total token weight in the places of h. Since the circuit h forms a conservative GPN, the firing of transitions in h can not reduce this total weight. In a long run, the total weight of tokens in h increases. H can not be repetitive. The rest of the proof is similar.

(iv) If h is conflict-free, then q_h must be 1; otherwise H can not be strongly repetitive. The concurrent-free case can be reduced to conflict-free case. Q.E.D.

---

[4] $x \lneqq y$ if and only if $x \leqq y$ and $x \neq y$.

COROLLARY. *If H is a well-behaved* ff *Petri net, H is a marked graph.*

Proof. Every directed circuit h in a Petri net has $q_h = 1$. Q.E.D.

The implication of the previous theorem on conflict-free or concurrent-free GPN's is interesting. Consider a well-behaved conflict-free GPN H. By Theorem 12, every directed circuit h in H must have $q_h = 1$. By Lemmas 1(b) and 5, we can also show that a strongly connected conflict-free GPN in which every directed circuit h has $q_h = 1$ must be strongly repetitive and hence conservative (Theorem 10). Therefore the following result, which was implicit in [6], can be derived from our previous analysis.

THEOREM 13. *Let H be a strongly connected conflict-free (or concurrent-free)* GPN. *H is well-behaved if and only if every circuit h in H has* $q_h = 1$.

Since any directed circuit in h in a Petri net has $q_h = 1$, the following result in [1], [2] and [4] becomes a corollary of Theorem 13.

COROLLARY. *A connected marked graph is well-behaved if and only if it is strongly connected.*

Finally, we establish the following result for Petri nets.

THEOREM 14. *The class of all primal and* ff *Petri nets is exactly the class of all strongly connected marked graphs.*

Proof. Every strongly connected marked graph is conservative. By Theorem 9, we can see that it is primal. If a Petri net is strongly connected and ff, we put one token in each place and then fire all transitions once. Since every place is an output place for some transition, at least one token is received by every place. If any place receives more than one token, the Petri net can not be conservative. If every place receives exactly one token, then no place has indegree greater than one. Therefore the Petri net is also bf. Q.E.D.

7. Conclusion. In § 2 we have presented a generalized form of Petri nets called GPN and MGPN. Two formalisms for the generalized Petri nets were given in § 3. In § 4 we have investigated several levels of termination properties. These properties can be different for general classes of MGPN's, but they coincide in the case of GPN's.

In §§ 5 and 6 we define the conditions for a GPN to be (i) strongly connected, (ii) strongly repetitive, (iii) conservative, (iv) well-behaved, (v) primal, (vi) ff, (vii) bf, (viii) fc and (ix) bc. These properties were summarized in Table 1. The class of ff

TABLE 1
Summary of main concepts

GPN properties	Definition
(i) strongly connected	The graph representation is strongly connected
(ii) strongly repetitive	$\mathbf{\Gamma} \cdot \mathbf{g} = 0$ has a position solution
(iii) conservative	$\mathbf{f} \cdot \mathbf{\Gamma} = 0$ has a positive solution
(iv) well-behaved	Both (ii) and (iii)
(v) primal	(iii) and "minimally" strongly connected
(vi) ff	For every place A, outdegree$(A) = 1$
(vii) bf	For every place A, indegree$(A) = 1$
(viii) fc	For every transition t, indegree$(t) = 1$
(ix) bc	For every transition t, outdegree$(t) = 1$

or bf GPN's contains properly all the marked graphs [1] and all the structures which were presented in a modified form called computation graphs in [6]. The main result in the last two sections is to characterize the property of being well-behaved or primal for the four subclasses of GPN's. The result is summarized in Table 2. The roman numerals in the table refer to the properties in Table 1.

TABLE 2
Summary of main results

GPN subclass	if	then
ff	(i)∧(ii)	(iii)∧(iv)∧(v)
bf	(i)∧(ii)	(iii)∧(iv)∧(v)
fc	(i)∧(iii)	(ii)∧(iv)
fc	(ii)	(v) iff ff
bc	(i)∧(iii)	(ii)∧(iv)
bc	(ii)	(v) iff bf

Acknowledgment. The author is indebted to the referees for pointing out an error in the original manuscript. Their other invaluable suggestions are also appreciated.

REFERENCES

[1] F. COMMONER, A. W. HOLT, S. EVEN AND A. PNUELI, *Marked directed graphs*, J. Comput. System Sci., 5 (1971), pp. 511–523.
[2] H. J. GENRICH, *Einfache Nicht-sequentiella Prozesse*, Gesellshaft für Mathematik und Datenverarbeitung, Birlinghoven, West Germany, 1970.
[3] A. W. HOLT, *Final report of the Information System Theory Project*, Tech. Rep. RADC-TR-68-305, Rome Air Development Center, Griffiss Air Force Base, N.Y., 1968.
[4] A. W. HOLT AND F. COMMONER, *Events and Conditions*, Applied Data Research, New York, 1970.
[5] R. M. KARP AND R. E. MILLER, *Parallel program schemata*, J. Comput. System Sci., 3 (1969), pp. 147–195.
[6] ———, *Properties of a model for parallel computations: Determinacy, termination, queuing*, SIAM J. Appl. Math., 14 (1966), pp. 1390–1411.
[7] R. M. KELLER, *Vector replacement systems: A formalism for modeling asynchronous systems*, Tech. Rep. 117, Dept. of Electr. Engng., Princeton Univ., Princeton, N.J., 1972.
[8] Y. E. LIEN, *Study of theoretical and practical aspects of transition systems*, Ph.D. thesis, Dept. of Electr. Engrg. and Comput. Sci., Univ. of Calif., Berkely, Calif., 1972.
[9] ———, *A note on transition systems*, J. Information Sci., to appear.
[10] S. S. PATIL, *Coordination of asynchronous events*, Rep. MAC-TR-72, Project MAC, Mass. Inst. of Tech., Cambridge, Mass., 1970.
[11] C. A. PETRI, *Communication with automata*, Suppl. 1 to Tech. Rep. RADC-TR-65-377, vol. 1, Griffiss Air Force Base, New York, 1966 = Kommunikation mit Automaten, Univ. of Bonn, West Germany, 1962.
[12] J. D. NOE, *A Petri net model of CDC 6400*, Proc. ACM/SIGOPS Workshop on Systems Performance Evaluation (1971), Association for Computing Machinery, New York, pp. 362–378.
[13] R. M. SHAPIRO AND H. SAINT, *A new approach to optimization of sequencing decisions*, Annual Review in Automatic Programming, vol. 6, Pergamon Press, Oxford, England, 1970, Part 5.
[14] *Project MAC Progress Report*, vol. VIII, Project MAC, Mass. Inst. of Tech., Cambridge, Mass., 1970–71, pp. 13–51.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

ALGORITHMIC ASPECTS OF VERTEX ELIMINATION ON GRAPHS*

DONALD J. ROSE,† R. ENDRE TARJAN‡ AND GEORGE S. LUEKER§

Abstract. We consider a graph-theoretic elimination process which is related to performing Gaussian elimination on sparse symmetric positive definite systems of linear equations. We give a new linear-time algorithm to calculate the fill-in produced by any elimination ordering, and we give two new related algorithms for finding orderings with special properties. One algorithm, based on breadth-first search, finds a perfect elimination ordering, if any exists, in $O(n+e)$ time, if the problem graph has n vertices and e edges. An extension of this algorithm finds a minimal (but not necessarily minimum) ordering in $O(ne)$ time. We conjecture that the problem of finding a minimum ordering is NP-complete

Key words. graph, breadth-first search, lexicographic search, Gaussian elimination, sparse linear equations, perfect elimination, triangulated graph, chordal graph

1. Introduction and notation. A *graph* is a pair $G=(V, E)$, where V is a finite set of $n=|V|$ elements called *vertices* and

$$E \subseteq \{\{v, w\} | v, w \in V, v \neq w\}$$

is a set of $e=|E|$ unordered vertex pairs called *edges*. Given $v \in V$, the set

$$\text{adj}\,(v)=\{w \in V | \{v, w\} \in E\}$$

is the set of vertices *adjacent* to v. The notation v—w means $w \in \text{adj}\,(v)$; $v \not\!-\!w$ means $w \notin \text{adj}\,(v)$. If $A \subseteq V$, the *induced subgraph* $G(A)$ of G is the subgraph $G(A)=(A, E(A))$, where $E(A)=\{\{x, y\} \in E | x, y \in A\}$.

For distinct vertices $v, w \in V$, a v, w *chain* of length k is a sequence of distinct vertices $\mu=[v=v_1, v_2, \cdots, v_{k+1}=w]$ such that v_{i+1}—v_i for $i=1, 2, \cdots, k$. Similarly, a *cycle* of length k is a sequence of distinct vertices $\mu=[v_1, v_2, \cdots, v_k]$ such that v_{i+1}—v_i for $i=1, 2, \cdots, k-1$ and v_k—v_1. We will always assume that G is *connected*; that is, for each pair of distinct vertices $v, w \in V$, there is a v, w chain.

For a graph $G=(V, E)$ with $|V|=n$, an *ordering* of V is a bijection $\alpha : \{1, 2, \cdots, n\} \leftrightarrow V$. Sometimes we denote an ordering by $V=\{x_i\}_{i=1}^{n}$. $G_\alpha=(V, E, \alpha)$ is an *ordered graph*. In G_α, the set of vertices *monotonely adjacent* to a vertex v is defined by

$$\text{madj}\,(v)=\text{adj}\,(v) \cap \{w \in V | \alpha^{-1}(v) < \alpha^{-1}(w)\}.$$

The notation $v \rightarrow w$ means $w \in \text{madj}\,(v)$.

---

* Received by the editors December 9, 1975, and in revised form July 15, 1975. This research was partially sponsored by the Office of Naval Research under contract N00014-67-A-0298-0034 at Harvard University, by the National Science Foundation under Grant MCS 72-03752 A03 at Stanford University, by Miller Research Fellowship at the University of California, Berkeley, and by an NSF Graduate Fellowship at Princeton University. Part of this work was completed while the first author was visiting the Computer Science Department at the University of Colorado and the second author was visiting the Weizmann Institute of Science, Rehovot, Israel.

† Applied Mathematics, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts 02138.

‡ Computer Science Department, Stanford University, Stanford, California 94305.

§ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.

For a vertex v, the *deficiency* $D(v)$ is the set of edges defined by

$$D(v) = \{\{x, y\} | v—x,\ v—y,\ x \not— y,\ x \neq y\}.$$

The graph G_v obtained from G by

(i) adding edges so that all vertices in adj (v) are pairwise adjacent, and

(ii) deleting v and its incident edges

is the *v-elimination graph* of G. That is,

$$G_v = (V - \{v\}, E(V - \{v\}) \cup D(v)).$$

For an ordered graph $G_\alpha = (V, E, \alpha)$, the *elimination process*

$$P(G_\alpha) = [G = G_0, G_1, G_2, \cdots, G_{n-1}]$$

is the sequence of elimination graphs defined recursively by $G_0 = G$, $G_i = (G_{i-1})_{x_i}$ for $i = 1, 2, \cdots, n-1$. If $G_i = (V_i, E_i)$ for $i = 0, 1, \cdots, n-1$, the *fill-in* $F(G_\alpha)$ is defined by

$$F(G_\alpha) = \bigcup_{i=1}^{n-1} \tau_i,$$

where $\tau_i = D(x_i)$ in G_{i-1}, and the *elimination graph* G_α^* is defined by

$$G_\alpha^* = (V, E \cup F(G_\alpha)).$$

The notion of vertex elimination arises in the solution of sparse symmetric positive definite systems of linear equations by Gaussian elimination. Given any symmetric $n \times n$ matrix $M = (m_{ij})$ which represents the coefficients of a set of linear equations, we can construct an ordered graph $G_\alpha = (V, E, \alpha)$, where vertex x_i corresponds to row i (and variable i), and $\{x_i, x_j\} \in E$ if and only if $m_{ij} \neq 0$ and $i \neq j$. The unordered graph $G = (V, E)$ corresponds to the equivalence class of matrices PMP^T, where P is any permutation matrix.

Suppose we solve the system with coefficients M using Gaussian elimination, eliminating variables in the order $1, 2, \cdots, n$. Assuming no lucky cancellation of nonzero elements, the edges τ_i correspond exactly to the new nonzero elements created when variable i is eliminated. For further discussion of this correspondence, see [19], [21]. In order to make the elimination process efficient, we would like to create no more nonzero elements than necessary; that is, we would like to find an elimination ordering which minimizes the fill-in.

Given a graph $G = (V, E)$, an ordering α of V is a *perfect elimination ordering* of G if $F(G_\alpha) = \varnothing$. Thus α is a perfect elimination ordering if $v \to w$ and $v \to x$ in G_α imply $w—x$ or $w = x$. A graph which has a perfect elimination ordering is a *perfect elimination graph.* An ordering α is a *minimal elimination ordering* of G if no other ordering β satisfies $F(G_\beta) \subset F(G_\alpha)$, where the containment is proper. An ordering α is a *minimum elimination ordering* of G if no other ordering β satisfies $|F(G_\beta)| < |F(G_\alpha)|$.

Any elimination graph G_α^* is a perfect elimination graph, since α is a perfect ordering of this graph. Any perfect ordering of a graph is minimum, and any minimum ordering of a graph is minimal. If a graph is a perfect elimination graph, any minimal ordering is perfect.

The problem we would like to solve is that of finding a minimum elimination ordering for any graph G. However, this seems to be a very difficult task in general; we conjecture that the problem of finding a minimum elimination order is NP-complete[1], based on our proof that the corresponding problem for directed graphs is NP-complete [22]. Thus we restrict our attention to finding a minimal ordering for any graph and finding a perfect ordering for any perfect elimination graph.

Perfect elimination graphs arise in contexts other than Gaussian elimination. Rose [19], [21] has given several characterizations of perfect elimination graphs, including the one below.

A graph G is called *triangulated* if for every cycle $\mu = [v_1, v_2, \cdots, v_k]$ of length $k > 3$, there is an edge of G joining two nonconsecutive vertices of μ. Such an edge is called a *chord* of the cycle. For an arbitrary graph $G = (V, E)$, a set of edges F is a *triangulation* if $G' = (V, E \cup F)$ is triangulated. F is a *minimal triangulation* if $G_0 = (V, E \cup F_0)$ is not triangulated for any $F_0 \subset F$.

THEOREM A (Rose [19], [21], Dirac [4]). *A graph G is a perfect elimination graph if and only if it is triangulated.*

Triangulated or perfect elimination graphs have also been called *chordal* [7], *monotone transitive* [21], and *rigid circuit graphs* [4]. Gavril [6] has presented efficient algorithms for finding all maximal cliques, maximum cliques, minimum colorings, maximum independent sets, and minimum clique coverings in triangulated graphs (for arbitrary graphs, these problems are NP-complete). These algorithms depend upon exploiting the necessary perfect elimination ordering. Assuming that such an ordering is given, it is easy to implement Gavril's algorithms to run in $O(n+e)$ time. ($O(n+e)$ is optimum to within a constant factor; the time bounds in [6] are not as tight as possible.) Several important classes of graphs, such as trees, k-trees [20], and interval graphs [5], [9], are triangulated, and a recognition algorithm for triangulated graphs can be used to recognize interval graphs efficiently [5], [9], [16].

The recognition problem for perfect elimination graphs bears a superficial resemblance to the problem of testing a directed graph for transitivity. It is easy to construct an $O(ne)$ algorithm to find a perfect ordering of a graph, if one exists. Gavril [7] has developed a way to find a perfect ordering of a graph G in $O(t(n, e)+n+e)$ time, where $t(n, e)$ is the time required to square the adjacency matrix of G. If Strassen's method of matrix multiplication [25] is used, this algorithm has an $O(n^{2.81})$ time bound. Below we present an $O(n+e)$ algorithm for finding perfect orderings. The algorithm uses a lexicographic search (or ordering scheme) which is a special type of breadth-first search. Surprisingly, a similar ordering scheme is useful in solving certain scheduling problems [2], [12], [24].

---

[1] The NP-complete problems, roughly speaking, are the hardest problems solvable using *nondeterministic* polynomial-time algorithms. Either *all* the NP-complete problems have deterministic polynomial-time algorithms or none of them do. Many people have tried and failed to find polynomial-time algorithms for problems in this class, but no one has proved that such algorithms do not exist. The tautology problem of propositional calculus, the traveling salesman problem, the maximum clique problem, and many other well-known problems are NP-complete. See [3] and [13] for further information.

Ohtsuki, Cheung, and Fujisawa [17] have extended Rose's results in order to develop algorithms for finding minimal orderings. Given a graph $G=(V, E)$, let V^* be the set of vertices such that $v \in V^*$ if and only if, for each $\{x, y\} \in D(v)$, there is a chain from x to y which contains no vertices in $\{v\} \cup \text{adj}\,(v) - \{x, y\}$.

THEOREM B [17]. *An ordering α on G is a minimal elimination ordering if and only if $\alpha(i) \in V_{i-1}^*$, where $P(G_\alpha)=[G_0, G_1, G_2, \cdots, G_{n-1}]$ and $G_i=(V_i, E_i)$ for $i=0, 1, \cdots, n-1$.*

This theorem leads to an algorithm for finding minimal orderings. Ohtsuki [18] has refined this method to get an $O(ne)$ algorithm for finding minimal orderings. The lexicographic ordering we consider here gives a different $O(ne)$ algorithm for finding minimal orderings, and our analysis of the properties of a lexicographic search leads to a characterization of minimal triangulations in terms of the cycle structure of the minimal triangulated graph.

We shall first study the more general problem of finding minimal orderings and then streamline our algorithm to solve the easier problem of finding perfect orderings. In § 2 we derive some properties of minimal and perfect orderings. In § 3 we motivate the idea of a lexicographic search by considering the relationship between breadth-first searches and perfect orderings. In §§ 4 and 5 we consider in detail the analysis and implementation of lexicographic orderings, and in § 6 we present some additional remarks. Although our results deal mainly with the application of lexicographic search to produce minimal and perfect elimination orderings, we feel the notion of a lexicographic search is algorithmically interesting and may have wider application.

2. Properties of elimination orderings and fill-in. We begin by developing some properties of elimination orderings and of the fill-in they produce.

LEMMA 1. *Let α be a perfect elimination ordering of a triangulated graph $G=(V, E)$, and let $x \in V$. Then α is also a perfect ordering of $G'=(V, E \cup D(x))$.*

Proof. We must show that for any $\{w, y\}, \{z, y\} \in E \cup D(x)$ with $\alpha^{-1}(y) < \min(\alpha^{-1}(w), \alpha^{-1}(z))$ and $w \neq z$, we have $\{w, z\} \in E \cup D(x)$. There are three cases. If $\{w, y\}, \{z, y\} \in E$, then $\{w, z\} \in E$ since α is perfect. If $\{w, y\}, \{z, y\} \in D(x)$, then $w, z \in \text{adj}\,(x)$ and $\{w, z\} \in E \cup D(x)$. The last case is $\{w, y\} \in E$, $\{z, y\} \in D(x)$ (or equivalently $\{w, y\} \in D(x)$, $\{z, y\} \in E$). This means $y, z \in \text{adj}\,(x)$ and $\{y, z\} \notin E$. If $w=x$, $z \in \text{adj}\,(x)$ means $\{w, z\} \in E$. Otherwise (i.e., if $w \neq x$), $\alpha^{-1}(x) > \alpha^{-1}(y)$ since α is perfect, and $\{x, w\} \in E$ since $x, w \in \text{adj}\,(y)$ and α is perfect. But $w, z \in \text{adj}\,(x)$ imply $\{w, z\} \in E \cup D(x)$. □

COROLLARY 1. *If $G=(V, E)$ is triangulated and x is any vertex, the elimination graph $G_x=(V-\{x\}, E(V-\{x\}) \cup D(x))$ is triangulated.* (This corollary is also proved in [21].)

COROLLARY 2. *If $G=(V, E)$ is triangulated and x is any vertex with $D(x)=\varnothing$, there is a perfect elimination ordering α with $\alpha(1)=x$.* (This corollary appears in [4], [5] and [21].)

LEMMA 2. *Let $G=(V, E)$ be a triangulated graph. Suppose $G'=(V, E \cup F)$ with $F \neq \varnothing$, $E \cap F=\varnothing$ is also triangulated. Then there exists some $f \in F$ such that $G'-f=(V, E \cup F-\{f\})$ is triangulated.*

Proof. We prove the theorem by induction on $n=|V|$. If $n \leqq 3$, the result is obvious since any graph with three or fewer vertices is triangulated. Suppose the

result is true for $n \leqq n_0$ and let $n = n_0 + 1$. Let $R = \{x | D(x) = \varnothing\}$, where $D(x)$ is the deficiency in G. Let $S = \{x | D'(x) = \varnothing\}$, where $D'(x)$ is the deficiency in G'. We know $R \neq \varnothing$ and $S \neq \varnothing$. There are two cases.

(i) For some $x \in S$ there exists an edge $f = \{u, x\} \in F$. By Corollary 2 there is a perfect elimination order β for G' with $\beta(1) = x$. Then β is also a perfect elimination order for $G' - f$.

(ii) Case (i) does not hold. We prove that there exists some $x \in S$ with $F \not\subseteq D(x)$. Pick any $z \in S$. If $F \not\subseteq D(z)$, let $x = z$. Otherwise, since $D(z) \subseteq F$, $F = D(z)$. In this case, let x be any vertex such that $x \in R$. By Corollary 2, there is a perfect ordering α of G such that $\alpha(1) = x$, and by Lemma 1, since $F = D(z)$, α is a perfect ordering of G'. Thus $x \in S$. Since $D(x) = \varnothing$, $F \not\subseteq D(x)$.

Now $G_x = (V - \{x\},\ E(V - \{x\}) \cup D(x))$ and $G'_x = (V - \{x\}, E(V - \{x\}) \cup F \cup D(x))$ are triangulated by Corollary 1. By the induction hypothesis, there exists $f \in F - D(x)$ such that $G'_x - f$ is triangulated. But then $G' - f$ is triangulated since $f \notin D(x)$. □

THEOREM 1. *Let $G = (V, E)$ be a graph, and let $G' = (V, E \cup F)$ be triangulated with $E \cap F = \varnothing$. F is a minimal triangulation if and only if for each $f \in F$, $G' - f = (V, E \cup F - \{f\})$ is not triangulated.*

Proof. One direction is immediate from the definition of minimal triangulation; Lemma 2 gives the other direction. □

LEMMA 3. *Let $G = (V, E)$ be triangulated and $f \in E$. Then either $G - f$ is triangulated or $G - f$ has an unchorded cycle of length* 4.

Proof. If $G - f$ is not triangulated and has an unchorded cycle of length $\geqq 5$, then adding f to $G - f$ will not suffice to make $G - f$ triangulated; i.e., G is not triangulated. □

THEOREM 2. *Let $G = (V, E)$ be a graph and $G' = (V, E \cup F)$ be triangulated. Then F is a minimal triangulation if and only if each $f \in F$ is a unique chord of a* 4*-cycle in G'.*

Proof. If F is minimal and $f \in F$, $G' - f$ is not triangulated and hence contains an unchorded cycle μ of length 4 by Lemma 3. If each $f \in F$ is a unique chord of a 4-cycle in G', then $f \in F$ implies $G' - f$ has an unchorded cycle of length 4, and Theorem 1 implies F is minimal. □

Theorems 1 and 2 provide two useful characterizations of minimal triangulations (and of minimal orderings, since α is a minimal ordering if and only if $F(G_\alpha)$ is a minimal triangulation). In § 4 we prove that the lexicographic ordering scheme defined there produces an ordering whose fill-in satisfies the unique chord property of Theorem 2. Thus any lexicographic ordering is minimal.

LEMMA 4. *Let $G_\alpha = (V, E, \alpha)$ be an ordered graph. Then $\{v, w\}$ is an edge of $G^*_\alpha = (V, E \cup F(G_\alpha))$ if and only if there exists a chain $\mu = [v = v_1, v, \cdots, v_{k+1} = w]$ in G_α such that*

$$\alpha^{-1}(v_i) < \min(\alpha^{-1}(v), \alpha^{-1}(w)), \qquad 2 \leqq i \leqq k.^2 \tag{1}$$

Proof. We show by induction on $l = \min(\alpha^{-1}(v), \alpha^{-1}(w))$ that, given any edge $\{v, w\}$ in G^*_α, there exists a v, w chain with the required property (1). Suppose

[2] For convenience, although at the risk of possible confusion, we adopt the following convention: in the vacuous case where v—w and hence $\{v_2, \cdots, v_k\} = \varnothing$, the condition is regarded as satisfied. Similar conventions are followed below.

$l = 1$. Then v—w in G_α, hence in G_α^*, and (1) holds vacuously. Suppose the result holds when $l \leqq l_0$, and consider the case $l = l_0 + 1$. If v—w in G_α, then again (1) holds vacuously. Otherwise $\{v, w\} \in F(G_\alpha)$, and we have by the definition of $F(G_\alpha)$ an $x \in V$ with $\alpha^{-1}(x) < \min(\alpha^{-1}(v), \alpha^{-1}(w))$ and x—v, x—w in G_α^*. The induction hypothesis implies the existence of x, v and x, w chains in G_α satisfying (1), and combining these chains gives the required v, w chain.

The converse is established by induction on k, the length of μ. If $k = 1$, v—w in G_α^* trivially. Suppose the converse holds for $k \leqq k_0$, and consider the case $k = k_0 + 1$. Let $\mu = [v = v_1, v_2, \cdots, v_{k+1} = w]$ and choose $x = v_i$, where $\alpha^{-1}(v_i) = \max\{\alpha^{-1}(v_j) | 2 \leqq j \leqq k\}$. The induction hypothesis implies that v—x and x—w in G_α^*; hence $v - w$ in G_α^*. □

This lemma provides a characterization of the fill-in produced by any elimination ordering. It is useful to have an efficient algorithm for calculating the fill-in. We derive another characterization for this purpose.

LEMMA 5. *Let $G_\alpha = (V, E, \alpha)$ be an ordered graph. Then $E \cup F(G_\alpha)$ is the smallest set E^* of edges such that $E \subseteq E^*$ and*

$$(2) \qquad (v \to w \quad in \quad E^*) \Rightarrow (m(v) = w \quad or \quad m(v) \to w \quad in \quad E^*),$$

where $m(v)$ is the vertex u with minimum $\alpha^{-1}(u)$ such that $v \to u$ in E^*.

Proof. $E^* = E \cup F(G_\alpha)$ clearly satisfies $E \subseteq E^*$ and (2). We must prove that any set E^* satisfying $E \subseteq E^*$ and (2) contains $E \cup F(G_\alpha)$. Suppose $\{v, w\} \in E \cup F(G_\alpha)$. Without loss of generality, assume $\alpha^{-1}(v) < \alpha^{-1}(w)$. We prove by induction on $i = \alpha^{-1}(v)$ that $\{v, w\} \in E^*$. Suppose this result holds for $i \leqq i_0$, and consider the case $i = i_0 + 1$. If $\{v, w\} \in E$, then $\{v, w\} \in E^*$. If $\{v, w\} \in F(G_\alpha)$, there is some u with $u \to v$, $u \to w$ in G_α^*. By the induction hypothesis, $u \to v$, $u \to w$ in E^*. Pick the u such that $u \to v$, $u \to w$ in E^* and $\alpha^{-1}(u)$ is maximum. Then $v = m(u)$ (otherwise $m(u) \to v$, $m(u) \to w$ in E^* by (2) and $\alpha^{-1}(u)$ is not maximum). But then $m(u) \to w$ in E^* by (2); i.e., $\{u, w\} \in E^*$. □

The following algorithm uses Lemma 5 to compute the edges in G_α^* for any ordered graph $G_\alpha = (V, E, \alpha)$. If $A(v) = \{w | v \to w \text{ in } G_\alpha\}$ for all vertices v when the algorithm starts, then $A(v) = \{w | v \to w \text{ in } G_\alpha^*\}$ for all v when the algorithm finishes.

```
ALGORITHM FILL: begin
    loop: for i := 1 until n − 1 do begin
            v := α(i)
            m(v) := α(min{α⁻¹(w) | w ∈ A(v)});
        add: for w ∈ A(v) do if w ≠ m(v) then
                add w to A(m(v));
end end FILL;
```

It is immediate from Lemma 5 that FILL correctly computes the edges of G_α^*. Efficient implementation of FILL is discussed in § 5.

3. Breadth-first search and perfect orderings. Given a graph G, a *breadth-first search* of G starting at a vertex s is a systematic examination of the edges of G using the following algorithm.

ALGORITHM BFS: **begin**
initialize *queue* to contain s;
$level(s) := 0$;
mark s explored;
while *queue* is nonempty **do begin**
remove first vertex v on *queue*;
explore: **for** $w \in \text{adj}(v)$ **do if** w is unexplored **then begin**
add w to end of *queue*;
$level(w) := level(v) + 1$;
mark $\{v, w\}$ as a tree edge;
mark w explored;
end end end BFS;

At each step, this algorithm picks an edge incident to the oldest reached vertex and finds out where the edge leads. The edge may lead either to a vertex already reached or to a new vertex, which is now considered reached.

During execution of this algorithm, statement *explore* processes each edge exactly twice; once for $w \in \text{adj}(v)$ and once for $v \in \text{adj}(w)$. The effects of BFS are (i) to generate a spanning tree of G, given by the edges $\{v, w\}$ such that w is unreached when statement *explore* is executed with $w \in \text{adj}(v)$, and (ii) to partition the vertices of G into *levels*: if v is a vertex, $level(v) = i$ if the shortest chain from s to v has length i. Figure 1 illustrates BFS applied to a graph.

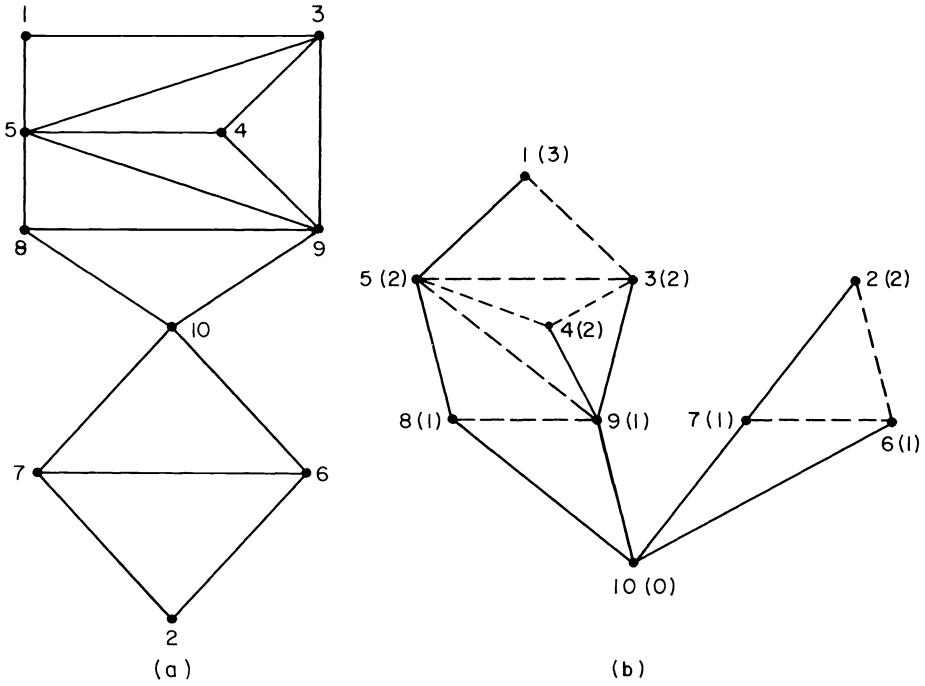


FIG. 1. *Breadth-first search of a graph starting at vertex* 10. (a) *Graph.* (b) *Tree produced by* BFS, *with level numbers in parentheses, and nontree edges dashed. Vertex numbers give a perfect elimination order which is consistent with level order.*

Each edge joins two vertices on the same level or on adjacent levels. If α is a perfect elimination order for a graph G, and a vertex v is joined to a vertex w with $level(w) = level(v)+1$ and to a vertex u with $level(u) = level(v)-1$, then $\alpha^{-1}(v) > \min(\alpha^{-1}(u), \alpha^{-1}(w))$ since $\{u, w\}$ is not an edge of G. This suggests that any perfect elimination graph has a perfect ordering which is consistent with the partial ordering by levels. This conjecture is true. The numbering in Fig. 1 shows a perfect ordering with this property.

Thus the levels given by BFS convey some information about perfect orderings. But we must break ties within the levels. We can use a breadth-first search within each level to accomplish this, if the new searches are guided by the inter-level edges. This idea leads to a highly complicated way of generating perfect orderings which uses BFS applied recursively. Fortunately, there is a simpler way to look at this method. We present it in the next section. In its full generality, the resulting algorithm gives minimal orderings, not just perfect orderings, and it is very efficient.

4. Lexicographic search for minimal and perfect orderings. To find minimal orderings and perfect orderings, we use a lexicographic ordering scheme which is a special type of breadth-first search. The vertices of the graph are numbered from n to 1. During the search, each vertex v has an associated *label* consisting of a set of numbers selected from $\{1, 2, \cdots, n\}$, ordered in *decreasing* order. Given two labels $L_1 = [p_1, p_2, \cdots, p_k]$ and $L_2 = [q_1, q_2, \cdots, q_l]$, we define $L_1 < L_2$ if, for some j, $p_i = q_i$ for $i = 1, 2, \cdots, j-1$ and $p_j < q_j$, or if $p_i = q_i$ for $i = 1, 2, \cdots, k$ and $k < l$. $L_1 = L_2$ if $k = l$ and $p_i = q_i$, $1 \leqq i \leqq k$.

4.1. Minimal orderings. Consider the following ordering scheme.

ALGORITHM LEX M: **begin**
assign the label $\varnothing$ to all vertices;
for $i := n$ **step** -1 **until** 1 **do begin**
select: pick an unnumbered vertex v with largest *label*;
comment assign v the number i;
$\alpha(i) := v$;
update: **for** each unnumbered vertex w such that there is a chain $[v = v_1, v_2, \cdots, v_{k+1} = w]$ with v_j unnumbered and $label(v_j) < label(w)$ for $j = 2, 3, \cdots, k$ **do** add i to $label(w)$;
end end LEX M;

This algorithm constructs an ordering α for an initially unordered graph $G = (V, E)$ and constructs a label $L(v)$ given by the final value of $label(v)$ for each $v \in V$. We call any ordering which can be generated by LEX M a *lexicographic ordering*. Figure 2 shows the application of this algorithm to an example graph. The complicated condition in statement *update* for updating labels is necessary because there may be fill-in edges; we are trying to find minimal orderings, not just perfect ones. Label updating can be simplified if the object is to find perfect orderings, as we shall see.

To establish the fact that algorithm LEX M produces a minimal ordering, we will prove that the fill-in produced by a lexicographic ordering has the unique chord property of Theorem 2. We need two lemmas which characterize the labels

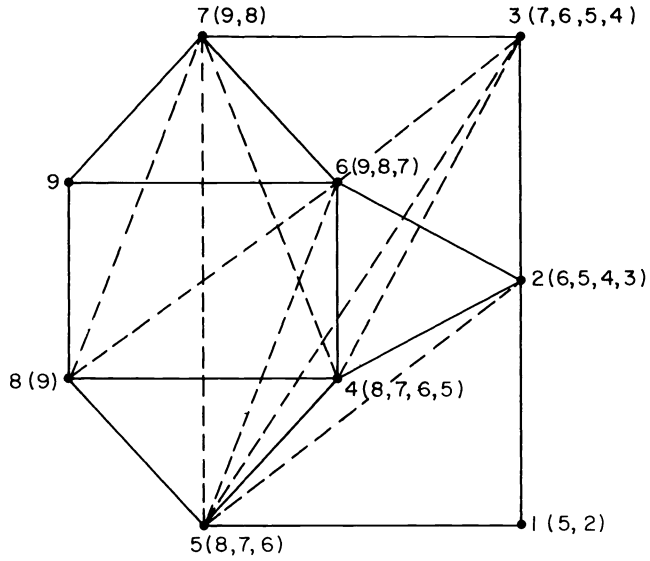


FIG. 2. *Minimal ordering of a graph generated by* LEX M. *Final labels are in parentheses, nine fill-in edges are dotted. Ordering is not minimum since there is another ordering with only five fill-in edges.*

$L(v)$. For any vertex w with label $L(w)$, let

$$L_i(w) = L(w) \cap \{n, n-1, \cdots, i+1\};$$

that is, $L_i(w)$ is the value of *label*(w) in LEX M just before the number i is assigned to a vertex. (Note: $L_n(w) = \varnothing$ for all w.)

LEMMA 6. *Let $G = (V, E)$ be a graph with lexicographic ordering α and labels $L(v)$, $v \in V$.*

(i) *If $L_i(v) < L_i(w)$, then $L_j(v) < L_j(w)$ for all $1 \leqq j \leqq i$.*

(ii) *$L_i(w) \leqq L_j(w)$ for all $j \leqq i$.*

(iii) *If $\alpha^{-1}(w) = j < \alpha^{-1}(v) = i$, then either $L_i(w) < L_i(v)$ and $L(w) < L(v)$, or $L_i(w) = L_i(v)$ and $L(v) \leqq L(w)$.*

(iv) *If $L(w) < L(v)$ with $\alpha^{-1}(w) = j$ and $\alpha^{-1}(v) = i$, then either $j < i$ with $L_i(w) < L_i(v)$, or $i < j$ with $L_j(w) = L_j(v)$.*

(v) *$j \in L(w)$ if and only if $\alpha^{-1}(w) < j$ and there exists a $v = \alpha(j)$, w chain $[v = v_1, v_2, \cdots, v_{k+1} = w]$ such that $L_j(v_i) < L_j(w)$ and $\alpha^{-1}(v_i) < j$, $2 \leqq i \leqq k$.*

The proofs of (i)–(v) are straightforward. Properties (i) and (v) follow from the definitions of the labels and the order relation; (iii) and (iv) summarize statement *select* of LEX M. Property (v) follows from (i), (ii) and statement *update* of LEX M; (v) means that the updated labels produced by one execution of statement *update* depend only on the old labels and not on the order of updating.

LEMMA 7. *If α is a lexicographic ordering of $G = (V, E)$, then in G^*_α,*

$$L(w) = \{\alpha^{-1}(v) | w \to v\}$$

for $w \in V$.

Proof. Suppose $i = \alpha^{-1}(v) \in L(w)$. Then, by (v) of Lemma 5, $\alpha^{-1}(w) < i$ and there exists a chain $[v = v_1, v_2, \cdots, v_{k+1} = w]$ with $L_i(v_j) < L_i(w)$ for $2 \leqq j \leqq k$. Thus, for each such j, $L_m(v_j) < L_m(w)$ for all $m \leqq i$ and hence $\alpha^{-1}(v_j) < \alpha^{-1}(w)$. Then $w \to v$ follows from Lemma 4.

The proof of the converse is somewhat less direct. Suppose that $w \to v$, and let $i = \alpha^{-1}(v)$. By Lemma 4 there exists a chain $[v = v_1, v_2, \cdots, v_{k+1} = w]$ with $\alpha^{-1}(v_j) < \alpha^{-1}(w)$, $2 \leqq j \leqq k$. Since $\alpha^{-1}(v_j) < \alpha^{-1}(w) < i$, we have $L_i(v_j) \leqq L_i(w)$, $2 \leqq j \leqq k$. If $L_i(v_j) < L_i(w)$ for all j such that $2 \leqq j \leqq k$, then $i \in L(w)$ directly from Lemma 6, part (v). Otherwise, suppose j_0 is the least such j with $L_i(v_{j_0}) = L_i(w)$. Since $\alpha^{-1}(v_{j_0}) < \alpha^{-1}(w) = m$ (say) we have, in addition (Lemma 6, parts (i) and (iii)), $L_p(v_{j_0}) \leqq L_p(w)$ for $m \leqq p \leqq i$. The chain $[v = v_1, v_2, \cdots, v_{j_0}]$ is such that $L_i(v_p) < L_i(v_{j_0})$ for $2 \leqq p \leqq j_0 - 1$; hence $i \in L_{i-1}(v_{j_0})$. But $L_{i-1}(v_{j_0}) \leqq L_{i-1}(w)$, $L_i(v_{j_0}) = L_i(w)$, and $i \in L_{i-1}(v_{j_0})$ imply $i \in L_{i-1}(w)$ by the lexicographic ordering of the labels. □

THEOREM 3. *Let $G = (V, E)$ be a graph with lexicographic ordering α and labels $L(v)$, $v \in V$. Then any edge $\{v, w\} \in F(G_\alpha)$ is the unique chord of some 4-cycle $\mu = [p, v, q, w]$ in G_α^*.*

Proof. Without loss of generality, we may assume $\alpha^{-1}(w) < \alpha^{-1}(v)$. Since $w \to v$ in G_α^*, from Lemma 7 we know that $\alpha^{-1}(v) \in L(w)$. Thus by Lemma 6, part (v), there exists a v, w chain $[v = v_1, v_2, \cdots, v_{k+1} = w]$ in G_α such that $L_j(v_i) < L_j(w)$ and $\alpha^{-1}(v_i) < j$ for $2 \leqq i \leqq k$, where $j = \alpha^{-1}(v)$. Let $l = \max\{\alpha^{-1}(v_i) | 2 \leqq i \leqq k\}$ and let $p = \alpha(l)$. Then $p \to v$ and $p \to w$ in G_α^* by Lemma 4.

Now $p \to w$ in G_α^* implies madj $(p) - \{w\} \subseteq$ adj (w) in G_α^*. Thus $L_j(p) \subseteq L_j(w)$ by Lemma 7. Since $L_j(p) < L_j(w)$, there is some $q \in$ madj $(w) -$ madj (p) with $\alpha^{-1}(q) > j$. Then $p \not\to q$ in G_α^*, $w \to q$ in G_α^*, and v—q in G_α^* since $w \to v$ and α is a perfect elimination ordering for G_α^*. Hence $\mu = [p, v, q, w]$ satisfies the theorem. □

Theorems 2 and 3 now immediately imply the following.

THEOREM 4. *Let $G = (V, E)$ be a graph with lexicographic ordering α. Then α is a minimal ordering.*

Proof. $F(G_\alpha)$ is a minimal triangulation by Theorems 2 and 3. □

4.2. Perfect orderings. Since any minimal ordering of a perfect elimination graph is perfect, we can test a graph G to see if it is perfect by generating a minimal ordering α using LEX M and testing whether $F(G_\alpha) = \varnothing$ using FILL. There is a better way, however. If G is perfect and α is a lexicographic ordering, then $F(G_\alpha) = \varnothing$; and by Lemma 7, $L(w) = \{\alpha^{-1}(v) | w \to v\}$ in G. Suppose we modify LEX M by simplifying statement *update*.

ALGORITHM LEX P: **begin**
assign the label $\varnothing$ to all vertices;
for $i := n$ **step** -1 **until** 1 **do begin**
select: pick an unnumbered vertex v with largest *label*;
comment assign v the number i, $\alpha(i) := v$;
*update*2: **for** each unnumbered vertex $w \in$ adj (v) **do** add i to *label*(w);
end end LEX P;

Algorithm LEX P will generate an ordering α which, by the observation above, must be perfect if G has any perfect orderings, although if G has no perfect

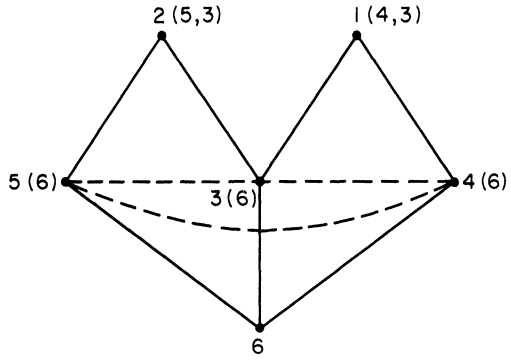


FIG. 3. *Order generated by* LEX P *for a nontriangulated graph. Fill-in edges are dotted. Ordering is not minimal since another ordering has fill-in* {{3, 4}, {3, 5}}.

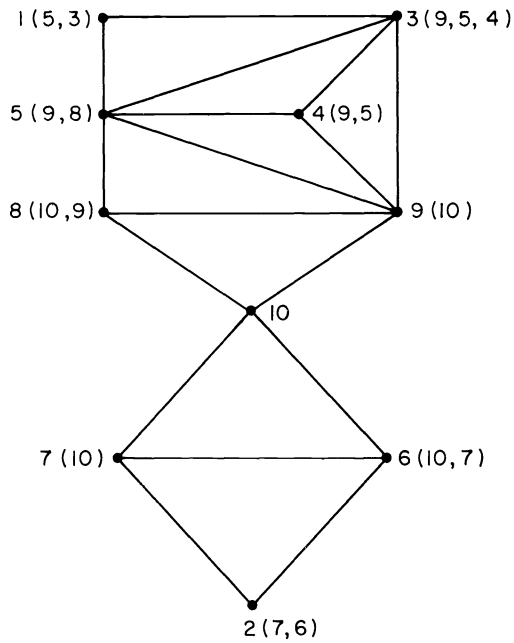


FIG. 4. *Final labels and perfect elimination order generated when* LEX P *is applied to graph in Fig.* 1.

orderings α may not be minimal (see Fig. 3). Figure 4 shows the application of LEX P to the graph in Fig. 1. The relationship between LEX M, LEX P, and BFS is as follows: if α is a LEX M ordering of a graph G, then α is a LEX P ordering of G_α^*; if β is a LEX P ordering of a graph G, then β is a BFS ordering of G.

5. Implementation and complexity. In this section we give linear-time implementations of FILL and LEX P, and an $O(ne)$ implementation of LEX M.

5.1. Computation of fill-in. To get algorithm FILL to run fast, we must make sure that the adjacency lists $A(v)$ do not contain too many redundant

elements. During the ith iteration of statement loop in FILL, we read through the elements in $A(\alpha(i))$ and eliminate duplicates. We use a Boolean array *test*(j), setting *test*(j) to true if and only if $\alpha(j) \in A(\alpha(i))$. The implementation appears below in ALGOL-like notation.

```
ALGORITHM FILL: begin
    for j := 1 until n do test (j) := false;
loop: for i := 1 until n − 1 do begin
        k := n;
        v := α(i);
        comment eliminate duplicates in A(v) and compute m(v);
    dup: for w ∈ A(v) do
            if test(α^{-1}(w)) then delete w from A(v)
            else begin
                test(α^{-1}(w)) := true;
                k := min (k, α^{-1}(w));
            end;
        m(v) := α(k)
        comment add required fill-in edges and reset test;
    add: for w ∈ A(v) do begin
    test(α^{-1}(w)) := false;
    if w ≠ m(v) then add w to A(m(v));
        end end end FILL;
```

Suppose this algorithm is applied to an ordered graph $G_\alpha = (V, E, \alpha)$ whose elimination graph G_α^* has e' edges. Each time statement *add* is executed, $A(v)$ is free of redundancies since *dup* eliminates such redundancies. The number of entries made to adjacency lists by one execution of *add* is thus bounded by $|A(v)|$ (defined in G_α^*), and the total number of additions to adjacency lists made by *add* over all iterations of *loop* is bounded by e'. The total time spent in *dup* over all iterations of *loop* is bounded by the total number of additions made to adjacency lists and is thus $O(n+e')$. The total running time of *add* over all iterations of *loop* is also bounded by the total number of additions to adjacency lists and is $O(n+e')$. Thus the total running time of FILL is $O(n+e')$.

5.2. Perfect orderings. The programming of LEX P is an interesting exercise in list processing, since the search requires that verticles be kept in a particular order depending on their labels. To make the implementation efficient, we do not actually calculate the labels of the vertices. For each label value, we keep a set of all vertices which have that label. We keep the sets in a queue ordered lexicographically by label (highest to lowest). When a new vertex v is numbered, we create a new set S' for each old set S containing a vertex w such that v—w. We delete from S all such vertices w and add them to the new set S', which is then inserted in the queue of sets just in front of S. It is easy to see that this method maintains the proper lexicographic ordering without actually calculating the labels. This method is similar to that used by Sethi [24] to implement the Coffman–Graham two-processor scheduling algorithm [2].

An implementation of this method is given below in an ALGOL-like notation. To maintain the queue and the sets, we use cells, each containing four items: *flag*,

head, *next*, and *back*. Certain cells are used as set headers. These cells are doubly linked using *head* and *back* in a queue which is ordered lexicographically by set label. A single cell is used as a header for this queue (it does not head a set of vertices).

Other cells are used to contain the vertices in the sets. Cells representing a set are doubly linked using *next* and *back*. If c is the header cell of a set, $next(c)$ points to the list of elements in the set. If c is an element cell, $head(c)$ contains the name of the vertex in the cell and $flag(c)$ points to the header of the set containing the cell. Figure 5 gives an example of this data structure.

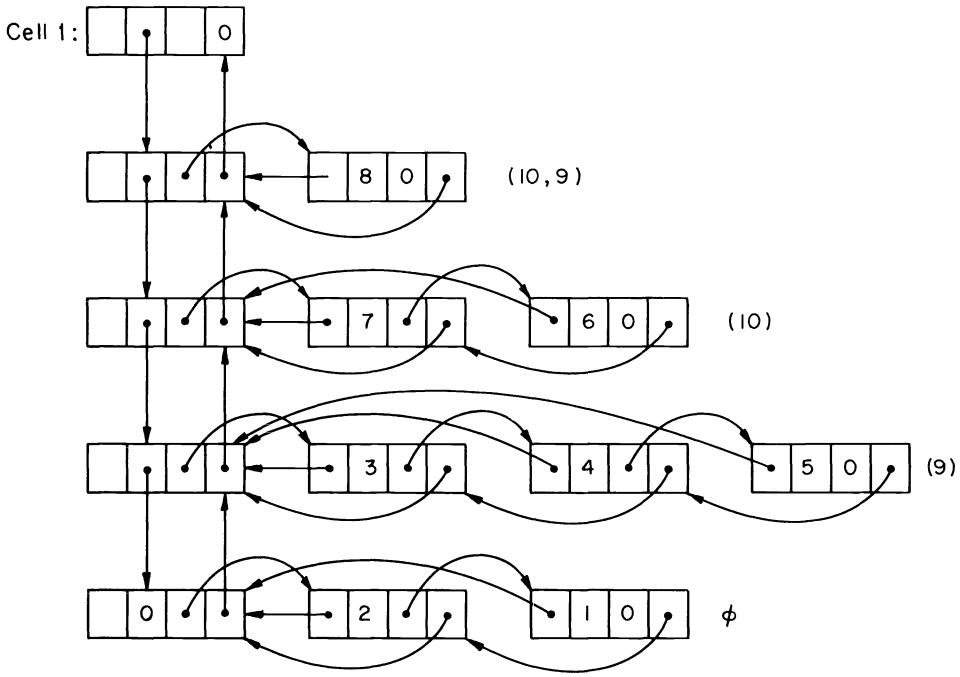


FIG. 5. *Data structure for Fig. 4 example after vertices 10 and 9 have been numbered. For convenience, vertices are assumed to be identified by their elimination number. Implicit labels of the sets are in parentheses.*

The program uses certain other variables. If v is a vertex, $cell(v)$ points to the cell containing v. The list *fixlist* contains pointers to the headers of the new sets created after a vertex v is numbered. Each such header h has $flag(h) = 1$ until after all the new sets are constructed. The algorithm then empties *fixlist*, resetting all the flags of the corresponding headers to zero.

ALGORITHM LEX P: **begin**

comment (implicitly) assign label $\varnothing$ to all vertices;

$head(1) := 2$;

$back(2) := 1$;

$head(2) := back(1) := next(1) := flag(1) := flag(2) := 0$;

$c := 3$;

```
comment c is the number of the first empty cell;
for v ∈ V do begin
    head(c) := v;
    cell(v) := next(c − 1) := c;
    flag(c) := 2;
    back(c) := c − 1;
    c := c + 1;
    α^{-1}(v) := 0;
end;
next(c − 1) := 0;
for i := n step −1 until 1 do begin
    comment skip empty sets;
    while next(head(1)) = 0 do begin head(1) := head(head(1));
        back(head(1)) := 1 end;
    comment pick next vertex to number;
select: p := next(head(1));
    comment delete cell of vertex from set;
    next(head(1)) := next(p);
    next(back(cell(w))) := next(cell(w));
    if next(cell(w)) ≠ 0 then back(next(cell(w))) :=
    comment assign v the number i;
    α(i) := v;
    α^{-1}(v) := i;
    fixlist := ∅;
update2: for w ∈ adj (v) do if α^{-1}(w) = 0 then begin
            comment delete cell of w from set;
            next(back(cell(w))) := next(cell(w));
            if next(cell(w)) ≠ 0 then back(next(cell(w))) :=
        back(cell(w));
            h := back(flag(cell(w)));
            comment if h is an old set then create a new set;
            if flag(h) = 0 then begin
                head(c) := head(h);
                head(h) := c;
                back(head(c)) := c;
                back(c) := h;
                flag(c) := 1;
                next(c) := 0;
                add c to fixlist;
                h := c;
                c := c + 1;
            end;
            comment add cell of w to new set;
            next(cell(w)) := next(h);
            if next(h) ≠ 0 then back(next(h)) := cell(w);
            flag(cell(w)) := back(cell(w)) := h;
            next(h) := cell(w);
        end;
```

```
            for h ∈ fixlist do flag(h):= v;
        end end LEX P;
```

It is routine to verify that algorithm LEX P, as implemented above, operates correctly and requires $O(n+e)$ time and space to order a graph.

We can use LEX P to generate an ordering and then use FILL to calculate its fill-in. If the fill-in is empty, the graph is triangulated, and the ordering is perfect. If the fill-in is nonempty, the graph is not triangulated. Thus we have an $O(n+e)$ algorithm to test whether a graph is a perfect elimination graph and to generate a perfect elimination ordering if there is one.

5.3. Minimal orderings. Algorithm LEX M apparently requires more execution time than LEX P, because statement *update* in LEX M requires more graph searching than statement *update*2 in LEX P. Here is an implementation of LEX M which runs in $O(ne)$ time.

To keep track of the labels, we use a less complicated scheme than used in the implementation of LEX P. Each unnumbered vertex w has an associated label number $l(w)$, such that $l(y)=l(z)$ if and only if y and z have the same label, and $l(y)<l(z)$ if and only if the label of y is less than the label of z. These label numbers are integers between 1 and k, where k is the number of distinct labels. When a new vertex v is numbered, each vertex w connected to v by a chain of the type defined in statement *update* is assigned a new label number $l'(w)=l(w)+\frac{1}{2}$. Label numbers of other vertices are not changed. The resultant label numbers are then sorted (using a radix sort) and new label numbers assigned so that all label numbers are integers between 1 and the new value of k.

To find chains of the type defined in statement *update*, we conduct a search starting from the newly numbered vertex v. First the search passes only through vertices of lowest label. Then the search is extended through vertices of second lowest label, and so on. In this way all appropriate chains can be found efficiently. The program appears below in ALGOL-like notation.

```
ALGORITHM LEX M: begin
    for v ∈ V do begin l(v) := 1; α^{-1}(v) := 0 end;
    k := 1;
loop: for i := n step −1 until 1 do begin
select: pick an unnumbered vertex v with l(v) = k;
        comment assign v the number i;
        mark v reached;
        α(i) := v;
        α^{-1}(v) := i;
        for j := 1 until k do reach(j) := ∅;
        mark all unnumbered vertices unreached;
        for w ∈ adj (v) and α^{-1}(w) = 0 do begin
            add w to reach(l(w));
            mark w reached;
            l(w) := l(w)+1/2;
            mark {v, w} as an edge of G*_α;
        end;
```

```
search: for j := 1 until k do
          while reach(j) ≠ ∅ do begin
              delete a vertex w from reach(j);
              for z ∈ adj (w) and z unreached do begin
                  mark z reached;
                  if l(z) > j then begin
                      add z to reach(l(z));
                      l(z) := l(z) + 1/2;
                      mark {v, z} as an edge of G*_α;
                  end else add z to reach(j);
          end end;
sort: sort unnumbered vertices by l(w) value;
      reassign l(w) values to be integers from 1 to k, redefining k appro-
      priately;
      end end LEX M;
```

It is an easy exercise to show that this program correctly implements algorithm LEX M to compute a minimal ordering. The time required per execution of statement *search* is $O(e)$ since each vertex can only be marked "reached" once and thus each edge can only be examined once. Statement *sort* requires $O(n)$ time when implemented as a radix sort [15]. The running time of the program is thus $O(e)$ per execution of statement *loop*, or $O(ne)$ time altogether. LEX M requires $O(n+e)$ storage space.

6. Remarks. This paper has given an $O(n+e)$ algorithm for finding a perfect elimination order on a graph if one exists and a related $O(ne)$ algorithm for finding minimal elimination orderings. The algorithm for finding perfect orderings is optimum to within a constant factor and is asymptotically faster than anything previously published. The minimal ordering algorithm, as implemented here, has the same asymptotic time bound as Ohtsuki's algorithm [18], but his algorithm does not calculate the fill-in produced by the ordering. The approach here solves both the perfect ordering and minimal ordering problems efficiently, reveals certain properties of lexicographic search, and provides a new characterization of minimal triangulations. It is not known whether there is a better algorithm for finding minimal orderings, or whether the problem of finding a minimum ordering is NP-complete.

It is possible to extend the notions of perfect, minimal, and minimum elimination orderings to directed graphs; such orderings are related to trying to minimize the fill-in when performing Gaussian elimination on sparse *asymmetric* matrices [10], [14]. Lemmas 1, 2 and 4, Corollaries 1 and 2, and Theorem 1 all generalize to directed graphs. However, lexicographic search does not seem to help in finding good orderings on directed graphs. We have constructed $O(ne)$ algorithms to compute the fill-in of any ordering and to find a perfect ordering if one exists. We have devised an $O(n^4)$ algorithm for finding a minimal ordering, using the proof of Lemma 2. We can show that testing whether a directed graph has a perfect ordering or computing any ordering's fill-in requires as much time as testing a directed graph for transitivity, and that finding a minimum ordering on a directed graph is an NP-complete problem [22].

Minimum orderings are desired in practice, but finding them is time-consuming. Minimal orderings are not necessarily close to minimum; for instance, if the lexicographic ordering scheme described here is applied to a graph representing an $n \times n$ square grid, the fill-in is $O(n^3)$, while the nested dissection method [8], [23] gives an ordering with $O(n^2 \log n)$ fill-in, which is minimum to within a constant factor [11]. The development of good ordering schemes for special cases (such as grid graphs) and the theoretical study of heuristics seem to be fruitful areas for future research.

In particular, two heuristics which seem to work well in practice are the minimum-degree heuristic and the minimum-fill-in heuristic [21]. It seems possible that the minimum degree heuristic produces minimum fill-in to within a constant factor, at least on grid graphs. A proof of such a statement would be extremely interesting.

When performing Gaussian elimination in practice, it might be important to minimize something other than the fill-in, such as the total operation count [1], [21]. The problem of finding an ordering which minimizes the operation count or some other criterion can be formulated graph-theoretically; only the fill-in criterion has been studied extensively.

REFERENCES

[1] U. BERTELE AND F. BRIOSCHI, *Non-serial Dynamic Programming*, Academic Press, New York, 1971.

[2] E. G. COFFMAN, JR. AND R. L. GRAHAM, *Optimal scheduling for two processor systems*, Acta Informat., 1 (1972), pp. 200–213.

[3] S. COOK, *The complexity of theorem-proving procedures*, Proc. 3rd Ann. ACM Symp. on Theory of Computing, 1971, pp. 151–158.

[4] G. A. DIRAC, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961), pp. 71–76.

[5] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[6] F. GAVRIL, *Algorithms for minimum coloring, maximum clique, minimum coloring by cliques and maximum independent set of a chordal graph*, this Journal, 1 (1972), pp. 180–187.

[7] ———, *An algorithm for testing chordality of graphs*, Information Processing Letters, 3 (1974), pp. 110–112.

[8] J. A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[9] P. C. GILMORE AND A. J. HOFFMAN, *A characterization of comparability graphs and of interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.

[10] L. HASKINS AND D. ROSE, *Toward characterization of perfect elimination digraphs*, this Journal, 2 (1973), pp. 217–224.

[11] A. J. HOFFMAN, M. S. MARTIN AND D. J. ROSE, *Complexity bounds for regular finite difference and finite element grids*, SIAM J. Numer. Anal., 10 (1973), pp. 364–369.

[12] T. C. HU, *Parallel sequencing and assembly line problems*, Operations Res., 9 (1961), pp. 841–848.

[13] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[14] D. J. KLEITMAN, *A note on perfect elimination digraphs*, this Journal, 3 (1974), pp. 280–282.

[15] D. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973, pp. 170–180.

[16] G. S. LUEKER AND K. S. BOOTH, *Linear algorithms to recognize interval graphs and test for the consecutive ones property*, Proc. 7th ACM Symp. on Theory of Computing, 1975, pp. 255–265.

[17] T. OHTSUKI, L. K. CHEUNG AND T. FUJISAWA, *On minimal triangulation of a graph*, J. Math. Anal. Appl., to appear.
[18] T. OHTSUKI, *A fast algorithm for finding an optimal ordering for vertex elimination on a graph*, this Journal, 5 (1976), pp. 133–145.
[19] D. J. ROSE, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.
[20] ———, *On simple characterizations of k-trees*, Discrete Math., 7 (1974), pp. 317–322.
[21] ———, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, Graph Theory and Computing, R. Read, ed., Academic Press, New York, 1973, pp. 183–217.
[22] D. ROSE AND R. TARJAN, *Algorithmic aspects of vertex elimination on directed graphs*, to appear.
[23] D. J. ROSE AND G. F. WHITTEN, *Automatic nested dissection*, Proc. ACM Ann. Conf., 1974, pp. 82–88.
[24] R. SETHI, *Scheduling graphs on two processes*, this Journal, 5 (1976), pp. 73–82.
[25] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1964), pp. 354–356.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

DECOMPOSITION THEOREMS FOR VARIOUS KINDS OF LANGUAGES PARALLEL IN NATURE*

SVEN SKYUM†

Abstract. In this paper we give a method for decomposing subclasses of different families of languages, parallel in nature, into other families. These decomposition theorems can be used to produce languages not in a family by using examples of languages not belonging to some "smaller" family.

Key words. formal languages, parallelism, L systems, definable sets, level grammars, decompositions

Introduction. Within the last few years there has been a growing interest in various forms of parallelism in rewriting systems. This is mainly due to the large amount of work done in the area of L systems or developmental languages.

In this paper we will examine the ability of different systems to generate languages in which the words of the language are composed of words from languages belonging to other families. These decomposition theorems can be used for examining the relation between various families of languages.

On this basis we can give examples of languages not belonging to a certain family by giving examples of languages not belonging to some smaller family.

Ehrenfeucht, Rozenberg and Skyum [3] employ this technique to show that the family of ET0L languages is properly included in the family of INDEX languages.

It is assumed that the reader is familiar with the basic notions concerning formal language theory. For unexplained notions, we refer to Salomaa [12].

The following notations are used in this paper:

I	denotes the set of nonnegative integers.
$\|\Sigma\|$	denotes the cardinality of Σ.
$\|x\|$	denotes the length of x.
$\|x\|_r$	denotes the number of occurrences in $x \in \Sigma^*$ of symbols belonging to some subalphabet $\Sigma_r \subseteq \Sigma$.
$\min(x)$	denotes the set of symbols occurring in x.

1. L systems. For a general introduction to L systems, we refer to [7] and [11].

DEFINITION 1. An E0L *system* is a 4-tuple $G=(V, P, w, \Sigma)$ where V (the alphabet) is a finite set of symbols, P (the productions) is a finite subset of $V \times V^*$, such that for each $A \in V$ there exists an $x \in V^*$ such that (A, x) is in P, w (the axiom) is a word in $V+$, and Σ (the target alphabet) is a subset of V.

DEFINITION 2. The E0L *language* $L(G)$ of an E0L system $G=(V, P, w, \Sigma)$ is

$$L(G)=\{x \in \Sigma^* | w \overset{*}{\underset{G}{\Rightarrow}} x\},$$

* Received by the editors November 6, 1974.

† Department of Computer Science, Institute of Mathematics, University of Aarhus, 8000 Aarhus, Denmark.

where $\overset{*}{\underset{G}{\Rightarrow}}$ is the transitive and reflexive closure of $\underset{G}{\Rightarrow}$ defined by $z \underset{G}{\Rightarrow} y$ iff $z = y = \lambda$ or there exist $a_1, a_2, \cdots, a_k \in V$ and $v_1, v_2, \cdots, v_k \in V^*$ such that $z = a_1 a_2 \cdots a_k$, $y = v_1 v_2 \cdots v_k$, and $(a_i, v_i) \in P$ for each $1 \leqq i \leqq k$.

DEFINITION 3. An E0L *system* $G = (V, P, w, \Sigma)$ *is deterministic* (abbreviated ED0L) if for each $A \in V$ there exists exactly one $x \in V^*$ such that $(A, x) \in P$.

DEFINITION 4. An ET0L *system* is a 4-tuple $G = (V, \mathcal{P}, w, \Sigma)$ where V, w, and Σ are as in the definition of an E0L system and $\mathcal{P}$ is a finite set (whose elements are called tables) such that for every $P \in \mathcal{P}$, (V, P, w, Σ) is an E0L system.

DEFINITION 5. The ET0L *language* $L(G)$ of an ET0L system $G = (V, \mathcal{P}, w, \Sigma)$ is

$$L(G) = \{x \in \Sigma^* | w \overset{*}{\underset{G}{\Rightarrow}} x\}$$

where $\underset{G}{\Rightarrow}$ is defined by $z \underset{G}{\Rightarrow} y$ iff $z = y = \lambda$ or there exist $P \in \mathcal{P}$, $a_1, a_2, \cdots, a_k \in V$, and $v_1, v_2, \cdots, v_k \in V^*$ such that $z = a_1 a_2 \cdots a_k$, $y = v_1 v_2 \cdots v_k$, and $(a_i, v_i) \in P$ for each $1 \leqq i \leqq k$. (We will then write $z \underset{P}{\Rightarrow} y$.)

DEFINITION 6. An ET0L *system is deterministic* iff each of the underlying E0L systems is deterministic.

DEFINITION 7. An EF0L (ETF0L) *system* is defined as above, but here we allow a finite set Ω of axioms instead of a single axiom w. The language generated by such a system consists of the union of the languages generated by the system obtained by choosing in turn each element $w \in \Omega$ to be the axiom.

DEFINITION 8. A 0L (F0L, T0L, TF0L) *system* is an E0L, (EF0L, ET0L, ETF0L) system $G = (V, P, w, \Sigma)$, $(G = (V, \mathcal{P}, w, \Sigma))$ where $\Sigma = V$.

For any class of systems, we will use the same notation for the family of languages generated by these systems.

DEFINITION 9. The *prefix* H attached to the name of a language family indicates that we are considering homomorphic images of the languages in the family, e.g., $L \subseteq \Sigma^*$ belongs to HDF0L iff there exists a DF0L system $G = (V, P, \Omega, V)$ and a homomorphism $h : V^* \to \Sigma^*$ such that $L = h(L(G))$.

Some of the relations between different L families are shown in Fig. 1. If two nodes labeled X and Y are connected by an oriented edge, then $X \subsetneq Y$, and if two

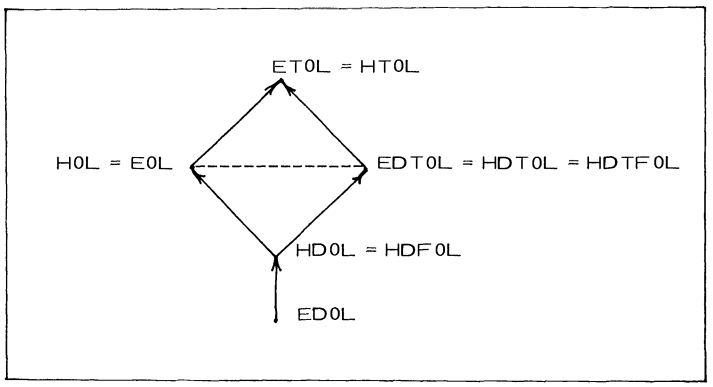


FIG. 1

nodes labeled X and Y are connected by a broken edge, then X and Y are mutually incomparable. The proofs of the relations can be found in [1] and [9].

2. Definable and extended definable sets. Simple and extended recurrence languages. The following five definitions define notions introduced by Rose [10].

DEFINITION 10. A *(n-ary) format* is any triple $(\Sigma; \xi; F)$, where Σ (the alphabet) is a finite set of symbols, ξ is a n-tuple $(\xi_1, \cdots, \xi_n)$ of symbols (called variables) not in Σ, and F is an n-tuple $(F_1, \cdots, F_n)$ of finite subsets of $(\Sigma \cup \{\xi_1, \cdots, \xi_n\})^*$.

DEFINITION 11. The *generating function* $g_{\Sigma;\xi;F}$ for a given (n-ary) format $(\Sigma; \xi; F)$ is defined thus: for each n-tuple $W = (W_1, \cdots, W_n)$ of finite subsets of $(\Sigma \cup \{\xi_1, \cdots, \xi_n\})^*$,

$$g_{\Sigma;\xi;F}(W) = \Big(\bigcup_{\sigma \in R_{\Sigma;\xi(W)}} \sigma(F_1), \cdots, \bigcup_{\sigma \in R_{\Sigma;\xi(W)}} \sigma(F_n) \Big),$$

where $R_{\Sigma;\xi(W)}$ is the set of all substitutions σ such that, for each $x \in \Sigma$, $\sigma(x) = \{x\}$ and $\sigma(\xi_i)$ is a subset of W_i with at most one element $(1 \leqq i \leqq n)$.

DEFINITION 12. The *approximating sequence* $E(k) = (E_1(k), \cdots, E_n(k))$ $(k \in I)$ for a given (n-ary) format $(\Sigma; \xi; F)$ is defined thus: $E_i(0) = \varnothing$ $(1 \leqq i \leqq n)$, and for all $k > 0$, $E(k) = g_{\Sigma;\xi;F}(E(k-1))$. Then n-tuple $E = (\bigcup_{k \geqq 0} E_1(k), \cdots, \bigcup_{k \geqq 0} E_n(k))$ is said to be generated by $(\Sigma; \xi; F)$.

DEFINITION 13. A language $L \subseteq \Sigma^*$ is said to be *extended definable* if it is the nth coordinate of the n-tuple generated by some (n-ary) format. We will denote the family of extended definable sets as ED.

DEFINITION 14. The *polynomial function* $p_{\Sigma;\xi;F}$ for a given (n-ary) format $(\Sigma; \xi; F)$ is defined thus: for each n-tuple $W = (W_1, \cdots, W_n)$ of finite subsets of $(\Sigma \cup \{\xi_1, \cdots, \xi_n\})^*$,

$$p_{\Sigma;\xi;F}(W) = (S_\xi^W(F_1), \cdots, S_\xi^W(F_n)),$$

where S_ξ^W is the substitution σ such that, for each $x \in \Sigma$, $\sigma(x) = \{x\}$ and $\sigma(\xi_i) = W_i$.

The following lemma is due to Rose [10].

LEMMA 1. *A language $L \subseteq \Sigma^*$ is definable (defined by Ginsburg and Rice [4]) if and only if it is the n-th coordinate for the minimal fixpoint (mfp) of the polynomial function $p_{\Sigma;\xi;F}$ for some (n-ary) format $(\Sigma; \xi; F)$.*

The mfp for $p_{\Sigma;\xi;F}$ is $D = (D_1, \cdots, D_n) = (\bigcup_{k \geqq 0} D_1(k), \cdots, \bigcup_{k \geqq 0} D_n(k))$, where $D_i(0) = \varnothing$ $(1 \leqq i \leqq n)$ and, for all $k \leqq 1$,

$$D(k) = p_{\Sigma;\xi;F}(D(k-1)).$$

We will denote the family of definable sets by D.

If we use the notion from Definitions 10 and 14, we can give the following definition of the simple recurrence languages introduced by Herman [6].

DEFINITION 15. A *recurrence system* is a 4-tuple $R = (\Sigma; \xi; F; \alpha)$, where $(\Sigma; \xi; F)$ is a (n-ary) format and $\alpha = (\alpha_1, \cdots, \alpha_n)$ is an n-tuple of finite subsets of Σ^*.

We define the simple recurrence language $L(R)$ of R by

$$L(R) = \bigcup_{k \geqq 0} D_n'(k),$$

where $D'(k)=(D'_1(k), \cdots, D'_n(k))$ is defined inductively by $D'(0)=(\alpha_1, \cdots, \alpha_n)$, and for $k \geqq 1$, $D'(k)=p_{\Sigma,\xi;F}(D'(k-1))$. The family of simple recurrence languages is denoted by SR.

DEFINITION 16. Let $R=(\Sigma; \xi; F; \alpha)$ be a recurrence system. The *extended recurrence language* $L_E(R)$ of R is defined by

$$L_E(R)=\bigcup_{k\geqq 0} E'_n(k),$$

where $E'(k)=(E'_1(k), \cdots, E'_n(k))$ is defined inductively by $E'(0)=(\alpha_1, \cdots, \alpha_n)$, and for $k \geqq 1$, $E'(k)=g_{\Sigma;\xi;F}(E'(k-1))$.

The family of extended recurrence languages is denoted by ER.

PROPOSITION 1. *For every recurrence system* $R=(\Sigma; \xi; F; \alpha)$, *there exists a recurrence system* $R'=(\Sigma; \xi'; F'; \alpha')$ *such that* $F'=(F'_1, F'_2, \cdots, F'_n)$ *is an* n*-tuple of finite subsets of* $\{\xi'_1, \xi'_2, \cdots, \xi'_n\}^*$, *and for* $1 \leqq i \leqq n$, α'_i *is either empty or consists of a single element in* Σ, $L(R)=L(R')$, *and* $L_E(R)=L_E(R')$. (The proof can be found in [6] and [15].)

DEFINITION 17. A *level grammar* is a 4-tuple $G=(V, P, S, \Sigma)$, where

V is the alphabet,

P (the productions) is a finite subset of $V \times V^*$,

$S \in V$ is the start symbol, and

$\Sigma \subseteq V$ is the terminal alphabet.

DEFINITION 18. We say that $w(A, n)w'$ *directly yields* $w(A_1, n+1) \cdots (A_k, n+1)w'$ *in* G ($w(A, n)w' \underset{G}{\Rightarrow} w(A_1, n+1) \cdots (A_k, n+1)w'$) if $w, w' \in (V, I)^*$ and $(A, A_1 \cdots A_k) \in P$. $\overset{*}{\underset{G}{\Rightarrow}}$ is the transitive and reflexive closure of $\underset{G}{\Rightarrow}$. As usual, we will write $\Rightarrow$ and $\overset{*}{\Rightarrow}$ if it is clear which grammar G is involved in.

DEFINITION 19. The *level language* $L(G)$ *is generated by a level grammar* $G=(V, P, S, \Sigma)$ if

$$L(G)=h(\{w \in (V, I)^* | (S, 0) \overset{*}{\Rightarrow} w\}) \cap \Sigma^*,$$

where $h: (V, I)^* \to V^*$ is a partial function only defined on strings, where all variables are associated with the same level number $n \in I$.

More specifically, h is defined as follows:

(i) $h(\lambda)=\lambda$;

(ii) for all $A_1, \cdots, A_k \in V$ and $n \in I$, $h((A_1, n) \cdots (A_k, n))=A_1 \cdots A_k$;

(iii) for all other strings in $(V, I)^+$, h is undefined.

We have that

$$L(G)=\bigcup_{n\geqq 0} [h(\{w \in (V, n)^* | (S, 0) \overset{*}{\Rightarrow} w\}) \cap \Sigma^*] = \bigcup_{n\geqq 0} L(G, n).$$

We say that $L(G, n)$ is the language of level n generated by G.

Example 1. Let $G=(\{S, a, b\}, \{(S, ab), (a, aa), (b, b), (b, bb)\}, S, \{a, b\})$.

Then

$$
\begin{aligned}
&L(G, 0) = \varnothing,\\
&L(G, 1) = \{ab\},\\
&L(G, 2) = \{aab, aabb\},\\
&\quad\vdots\\
&L(G, n) = \{a^{2^{n-1}} b^i \mid 1 \leqq i \leqq 2^{n-1}\},\\
&\quad\vdots\\
&L(G) \quad = \bigcup_{n \geqq 0} L(G, n) = \{a^{2^n} b^i \mid n \geqq 0,\ 1 \leqq i \leqq 2^n\}.
\end{aligned}
$$

The family of level languages will be denoted by LL.

DEFINITION 20. Let $G = (V, P, S, \Sigma)$ be a level grammar. We write $w_1(A, n)w_2(A, n) \cdots w_{k-1}(A, n)w_k \Rightarrow_P w_1 w w_2 w \cdots w_{k-1} w w_k$ if $w_i \in ((V, I) \setminus (A, n))^*$ $(1 \leqq i \leqq k)$ and $(A, n) \Rightarrow w$. $\overset{*}{\Rightarrow}_P$ is again the transitive and reflexive closure of $\Rightarrow_P$. We say that *w derives w' in parallel* if $w \overset{*}{\Rightarrow}_P w'$.

DEFINITION 21. The *parallel level language* $L_P(G)$ *generated by a level grammar* $G = (V, P, S, \Sigma)$ is

$$L_P(G) = h(\{w \in (V, I)^* \mid (S, 0) \overset{*}{\Rightarrow}_P w\}) \cap \Sigma^*.$$

Again $L_P(G) = \bigcup_{n \geqq 0} L_P(G, n)$, where

$$L_P(G, n) = h(\{w \in (V, n)^* \mid (S, 0) \overset{*}{\Rightarrow}_P w\}) \cap \Sigma^*.$$

Example 2. Let G be the level grammar from Example 1. Then

$$
\begin{aligned}
&L_P(G, 0) = \varnothing,\\
&L_P(G, 1) = \{ab\},\\
&L_P(G, 2) = \{aab, aabb\},\\
&L_P(G, 3) = \{aaaab, aaaabb, aaaabbbb\},\\
&\quad\vdots\\
&L_P(G, n) = \{a^{2^{n-1}} b^{2^{i-1}} \mid 1 \leqq i \leqq n\},\\
&\quad\vdots\\
&L_P(G) \quad = \bigcup_{n \geqq 0} L_P(G, n) = \{a^{2^n} b^{2^i} \mid 0 \leqq i \leqq n\}.
\end{aligned}
$$

The family of parallel level languages is denoted by PLL.

DEFINITION 22. A *restricted level grammar* $G = (V, P, S, \Sigma)$ is a level grammar with the restriction that for each $A \in \Sigma$, (A, A) is a production in P.

The corresponding families of restricted (parallel) level languages are denoted by RLL (RPLL).

Figure 2 shows the relations between the families defined in this section. The proofs of the relations can be found in [6], [10] and [15].

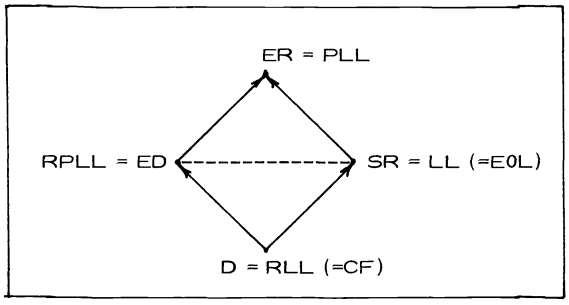


FIG. 2

3. Russian and Indian parallelism. For definitions and discussion, see [8], [13], [14] and [16].

DEFINITION 23. A *Russian parallel context-free grammar* is a 5-tuple $G=(V, \Sigma, P_u, P_o, S)$, where the only difference to an ordinary context-free grammar is that the set of productions is divided into two sets of productions, P_u (the universal productions) and P_o (the ordinary productions).

DEFINITION 24. The *language generated by a Russian parallel context-free grammar* is

$$L(G)=\{x \in \Sigma^* | S \overset{*}{\Rightarrow}_G x\}$$

where $\overset{*}{\Rightarrow}_G$ is the transitive and reflexive closure of $\Rightarrow_G$ defined by $z \Rightarrow_G y$ iff either

(i) $z=z_1Az_2$ and $y=z_1vz_2$ for some v, z_1, $z_2 \in (V \cup \Sigma)^*$, and $A \in V$ such that (A, v) is in P_o, or

(ii) $z=z_1Az_2A \cdots Az_k$ and $y=z_1vz_2v \cdots vz_k$ for some $v \in (V \cup \Sigma)^*$, $A \in V$, and $z_i \in ((V \cup \Sigma) \setminus \{A\})^*$, $1 \leqq i \leqq k$, such that $(A, v) \in P_u$.

We will denote the family of Russian parallel languages by RP.

DEFINITION 25. An *Indian parallel context-free grammar* is a 4-tuple $G=(V, \Sigma, P, S)$ like a context-free grammar, but P consists of universal productions only. That means that $(V, \Sigma, P, \varnothing, S)$ is a Russian parallel context-free grammar.

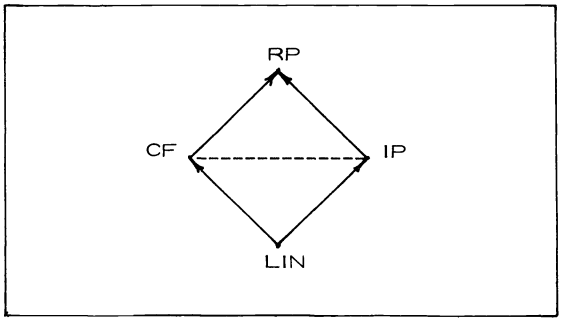


FIG. 3

We will denote the family of Indian parallel languages by IP.

The relations between RP, IP, CF and LIN (linear languages) are shown in Fig. 3. The proofs of the nontrivial relations can be found in [12] and [16].

In Fig. 4 we summarize the known inclusions between the families defined in §§ 1–3.

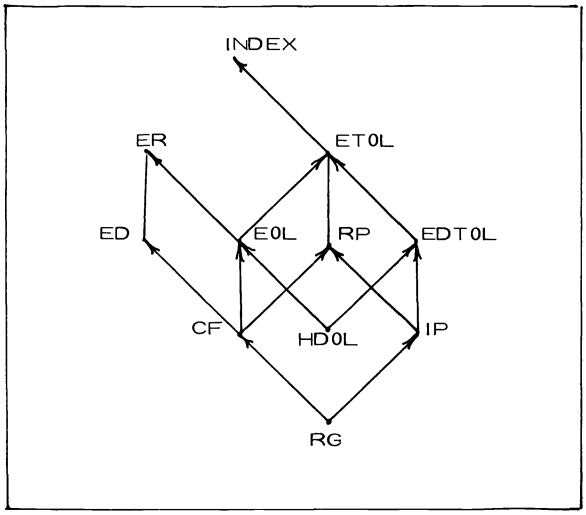


FIG. 4

4. Results. In this section we will examine the possibility of decomposing some languages from a certain family into languages from some smaller family.

THEOREM 1. *Let* Σ *be some alphabet and* $K \subseteq \Sigma^*$. *Let* $\# \notin \Sigma$ *and* $c_{\#}^n(K) = \{(w\#)^{n-1}w \mid w \in K\}$. *The families* HD0L, EDT0L, ED, ER, *and* IP *are closed under the operators* $c_{\#}^n$.

Proof. The proof is straightforward and is omitted.

DEFINITION 26. Let $L = h(L(G))$, where $G = (V, \mathcal{P}, w, V)$ is a T0L system and $h : V^* \to \Sigma^*$ is a homomorphism. $a \in V$ is called *essentially nondeterministic with respect to h* and G iff

(i) there exist words $x_1, x_2, x_3 \in V^*$, such that $x_1 a x_2 a x_3 \in L(G)$;

(ii) there exists a sequence of tables $P_{i_1}, P_{i_2}, \cdots, P_{i_n}$ and words $w_1, w_2 \in V^*$ such that for $j = 1, 2$,

$$a \underset{P_{i_1}}{\Rightarrow} w_{j1} \underset{P_{i_2}}{\Rightarrow} w_{j2} \underset{P_{i_3}}{\Rightarrow} \cdots \underset{P_{i_n}}{\Rightarrow} w_{jn} = w_j$$

and $h(w_1) \neq h(w_2)$.

LEMMA 2. *Let* $L = h(L(G))$, *where* $G = (V, \mathcal{P}, w, V)$ *is a* T0L *system and* $h : V^* \to \Sigma^*$ *a homomorphism. If there are no essentially nondeterministic symbols in V with respect to h and G, then* $L \in$ HDT0L.

Proof. Let $\bar{G} = (V, \bar{\mathcal{P}}, w, V)$ be the DT0L system where $\bar{\mathcal{P}}$ is defined by $P \in \bar{\mathcal{P}}$ iff $P \subseteq P'$ for some $P' \in \mathcal{P}$ and (V, P, w, V) is a D0L system. Now it is obvious that $L(\bar{G}) \subseteq L(G)$.

Let $x = h(y)$ where $y \in L(G)$. Let

$$w = w_0 \underset{P_{i_1}}{\Rightarrow} w_1 \underset{P_{i_2}}{\Rightarrow} \cdots \underset{P_{i_n}}{\Rightarrow} w_n = y$$

be a derivation of y in G. For $1 \leqq j \leqq n$, let P'_{i_j} be a table in $\bar{\mathscr{P}}$ such that $P'_{i_j} \subseteq P_{i_j}$, and if $a \in V$ only occurs once in w_{j-1} and is rewritten as z in the derivation, then $(a, z) \in P'_{i_j}$. If a occurs several times in w_{j-1}, then just choose one production from P_{i_j} to be in P'_{i_j}. Note that because a is not essentially nondeterministic, it does not matter which production one chooses. Then

$$w = w_0 \underset{P'_{i_1}}{\Rightarrow} w'_1 \underset{P'_{i_2}}{\Rightarrow} \cdots \underset{P'_{i_n}}{\Rightarrow} w'_n = y'$$

is a derivation in $\bar{G}$, and $h(y) = h(y')$.

THEOREM 2. *Let Σ be an alphabet and let $K \subseteq \Sigma^*$. Let $\# \notin \Sigma$.*

(a) *If $c^2_{\#}(K) \in$ ET0L, then $K \in$ EDT0L.*

(b) *If $c^3_{\#}(K) \in$ E0L, then $K \in$ HD0L.*

(c) *If $c^2_{\#}(K) \in$ RP, then $K \in$ IP.*

(d) *If $c^2_{\#}(K) \in$ ED, then $K \in$ ED.*

Proof. (a) Let $c^2_{\#}(K) = h(L(G)) \in$ HT0L = ET0L for some T0L system G and homomorphism h. Because of the form of $c^2_{\#}(K)$, it immediately follows that there is no essentially nondeterministic symbol in G. Therefore by Lemma 2 it follows that $c^2_{\#}(K) \in$ HDT0L = EDT0L.

Let then $c^2_{\#}(K) = h(L(G))$, where $G = (V, \mathscr{P}, w, V)$ is a DT0L system and $h : V^* \to (\Sigma \cup \{\#\})^*$ is a homomorphism.

Define $V_{\#} \subseteq V$ to be the set satisfying $a \in V_{\#}$ iff there exists an x such that $a \overset{*}{\Rightarrow} x$ and $\# \in \min(h(x))$. Note that every symbol in $V_{\#}$ can occur at most once in every word in $L(G)$.

Define a DT0L system $H = (\bar{V}, \bar{\mathscr{P}}, \bar{w}, \bar{V})$ as follows: $\bar{V} = V \times 2^{V_{\#}}$.

If $(a_0, a_1 a_2 \cdots a_{k_1} b_1 a_{k_1+1} \cdots a_{k_2} b_2 \cdots b_n a_{k_n+1} \cdots a_{k_{n+1}}) \in P$ $(n \geqq 0)$, where $a_i \in V \setminus V_{\#}$, $1 \leqq i \leqq k_{n+1}$, $b_i \in V_{\#}$, $1 \leqq i \leqq n$, and $P \in \mathscr{P}$, then for all $N \subseteq V_{\#}$, $([a_0, N], [a_1, M][a_2, M] \cdots [a_{k_1}, M][b_1, M \cup \{b_1\}][a_{k_1+1}, M \cup \{b_1\}] \cdots [a_{k_2}, M \cup \{b_1\}] \cdot [b_2, M \cup \{b_1, b_2\}] \cdots [b_n, M \cup \{b_1, b_2, \cdots, b_n\}][a_{k_n+1}, M \cup \{b_1, b_2, \cdots, b_n\}] \cdots [a_{k_{n+1}}, M \cup \{b_1, b_2, \cdots b_n\}])$ is in a corresponding table $\bar{P}$ in $\bar{\mathscr{P}}$. $M = \min(y) \cap V_{\#}$ for some y such that there exists an x, where $N \setminus \{a_0\} = \min(x)$ and $x \Rightarrow_P y$. Note that M is uniquely determined by N.

If $w = a_1 \cdots a_{k_1} b_1 a_{k_1+1} \cdots a_{k_2} b_2 a_{k_2+1} \cdots a_{k_n} b_n a_{k_n+1} \cdots a_{k_{n+1}}$, where $a_i \in V \setminus V_{\#}$, $1 \leqq i \leqq k_{n+1}$, and $b_i \in V_{\#}$, $1 \leqq i \leqq n$, then $\bar{w} = [a_1, \varnothing] \cdots [a_{k_1}, \varnothing][b_1, \{b_1\}][a_{k_1+1}, \{b_1\}] \cdots [a_{k_2}, \{b_1\}][b_2, \{b_1, b_2\}][a_{k_2+1}, \{b_1, b_2\}] \cdots [a_{k_n}, \{b_1, \cdots, b_{n-1}\}][b_n, \{b_1, \cdots, b_n\}][a_{k_n+1}, \{b_1, \cdots, b_n\}] \cdots [a_{k_{n+1}}, \{b_1, \cdots, b_n\}]$.

If we now define a homomorphism $g : \bar{V}^* \to \Sigma^*$ by

$$g([a, N]) = \begin{cases} h(a) & \text{if } \# \notin \min(h(b)) \text{ for all } b \in N, \\ v & \text{if } a \in N \text{ and } h(a) = v \# u \text{ for some } u \in \Sigma^*, \\ \lambda & \text{otherwise,} \end{cases}$$

then it follows that $K = g(L(H)) \in$ HDT0L = EDT0L.

(b) Let $c^3_{\#}(K) = h(L(G))$, where $G = (V, P, w, V)$ is an 0L system and $h : V^* \to (\Sigma \cup \{\#\})^*$ is a length preserving homomorphism. This is no restriction (see, e.g., Ehrenfeucht and Rozenberg, [3]). Let $n = |V|$, m be an integer such that $|w| \leqq m$ and if $(a, x) \in P$, then $|x| \leqq m$. Let $V_m \subseteq V$ be the set of mortal symbols in V; that means that $a \in V_m$ iff $a \Rightarrow^n x$ implies that $x = \lambda$. Let $V_v = V \setminus V_m$ be the set of vital symbols. Let $V_s \subseteq V$ be defined by $a \in V_s$ iff there exists an $i > 0$ such that if $x \in L(G)$, $|x|_v > i$ then $a \notin \min(x)$. Let $V_b = V \setminus V_s$.

Observations. 1. If $a \in V_b$ and $a \overset{*}{\Rightarrow} x$, then $\min(x) \subseteq V_b$.
2. There exists a $k > 0$ such that if $x \in L(G)$ and $|x|_v > k$, then $\min(x) \subseteq V_b$.
3. If $x \in L(G)$ and $|x| > m^n \cdot k$, then $\min(x) \subseteq V_b$.

To see the correctness of (3), let $w \overset{*}{\Rightarrow} w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = x$ be a derivation of $x \in L(G)$, where $|x| > m^n \cdot k$. Assume that $a \in \min(x) \cap V_s$. Then because of Observation 1 there exists a symbol $b \in \min(w_1) \cap V_s$ which implies that $|w_1|_v \leqq k$ and $|x| \leqq k \cdot m^n$, so our assumption cannot be true.

Let now $x \in L(G)$, $|x| > m^n \cdot k$, and $a \in \min(x)$. We will prove that if $a \Rightarrow^i w_1$ and $a \Rightarrow^i w_2$ for some $i > 0$ and $w_1, w_2 \in V^*$, then $h(w_1) = h(w_2)$. From that it will follow that $c^3_{\#}(K) \in \mathrm{HDF0L} = \mathrm{HD0L}$ because we can just choose one production for every symbol, and choose $\{x \in L(G) \| x| \leqq m^n \cdot k\}$ to be the set of axioms. Now back to the statement.

Let $t = \max\{|w_1|, |w_2|\}$. Because of Observation 3 above, $a \in V_b$ and therefore there exists a word $y \in L(G)$ such that $a \in \min(y)$ and $|y|_v > 3t + 2$. Let $y = z_1 a z_2$ and $z_j \Rightarrow^i v_j$ for $j = 1, 2$. Then $|v_1 w_j v_2| > 3t + 2$ for $j = 1, 2$, and since $|w_j| \leqq t$ for $j = 1, 2$ $h(w_1)$ and $h(w_2)$ must be equal. To prove that $K \in \mathrm{HD0L}$ if $c^3_{\#}(K) \in \mathrm{HD0L}$, we can use exactly the same technique as in the proof of $K \in \mathrm{HDT0L}$ if $c^2_{\#}(K) \in \mathrm{HDT0L}$.

(c) Let $c^2_{\#}(K) = L(G)$, where $G = (V, \Sigma \cup \{\#\}, P_u, P_o, S)$ is a Russian parallel context-free grammar. As for ordinary context-free grammars, we can assume that all nonterminals are useful. That means that for all $A \in V$ there exist words $x, y \in (V \cup \Sigma \cup \{\#\})^*$ and $v \in (\Sigma \cup \{\#\})^*$ such that $S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} v$.

Assume that $S \overset{*}{\Rightarrow} x_1 A x_2 A x_3 \overset{*}{\Rightarrow} v_1 A v_2 A \cdots A v_k \overset{*}{\Rightarrow} v_1 w_1 v_2 w_2 \cdots w_{k-1} v_k$ for some words $x_i \in (V \cup \Sigma \cup \{\#\})^*$, $1 \leqq i \leqq 3$, $v_j \in (\Sigma \cup \{\#\})^*$, $1 \leqq j \leqq k$, $w_i \in (\Sigma \cup \{\#\})^*$, $1 \leqq i \leqq k-1$, and some $A \in V$ such that A is not rewritten anywhere in the subderivation $x_1 A x_2 A x_3 \overset{*}{\Rightarrow} v_1 A v_2 A \cdots A v_k$.

Then $w_1 = w_2 = \cdots = w_{k-1}$ because $v_1 w_{i_1} v_2 w_{i_2} \cdots w_{i_{k-1}} v_k \in L(G)$ for all $1 \leqq i_1, i_2, \cdots, i_{k-1} \leqq k-1$. It then follows that $c^2_{\#}(K) = L(G')$, where $G' = (V, \Sigma \cup \{\#\}, P_u \cup P_o, \varnothing, S)$, which means that $c^2_{\#}(K) \in \mathrm{IP}$.

Define $V_{\#} \subseteq V \cup \{\#\}$ to be the set satisfying $A \in V_{\#}$ iff $A \overset{*}{\Rightarrow} x_1 \# x_2$ for some $x_1, x_2 \in \Sigma^*$.

Let $\bar{G} = (V, \Sigma, P', \varnothing, S)$, where P' is defined as follows: if $(A, A_1 A_2 \cdots A_k B A_{k+1} A_{k+2} \cdots A_n) \in P_u \cup P_o$, where $A_i \in V \cup \Sigma$, $1 \leqq i \leqq n$, and $B \in V_{\#}$, then $(A, A_1 A_2 \cdots A_k B) \in P'$ if $B \neq \#$, and otherwise $(A, A_1 A_2 \cdots A_k) \in P'$. (Note that in every sentential-form in G' there is exactly one occurrence of a letter in $V_{\#}$.)

If $(A, x) \in P_u \cup P_o$ and $V_\# \cap \min(x) = \varnothing$, then $(A, x) \in P'$. No other productions are in P'. It is easy to check that $K = L(\bar{G})$.

(d) The proof of part (d) is very similar to the last half of the proof of part (c) if we use the fact that ED = RPLL.

Remark 1. Note that it is an open question whether $c^2_\#(K) \in$ E0L implies $K \in$ HD0L or not.

Remark 2. If $c^2_\#(K) \in$ CF, then $K \in$ RG. This is in fact a special case of Theorem 2.3.2 in [5].

Conjecture. If $c^2_\#(K) \in$ ER, then $K \in$ ER.

Theorem 2 is visualized in Fig. 5. If two nodes labeled X and Y are connected by an oriented edge, then $c^3_\#(L) \in X$ implies that $L \in Y$. We can give a more general theorem than Theorem 2, namely the following.

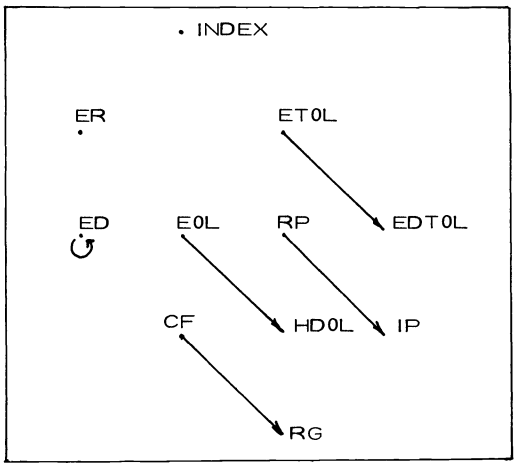


FIG. 5

THEOREM 3. *Let* Σ *be an alphabet and let* $K_1, K_2 \subseteq \Sigma^*$. *Let* $\# \notin \Sigma$ *and let* $f : K_1 \to K_2$ *be a bijective function from* K_1 *onto* K_2. *Let* $K = \{w \# f(w) | w \in K_1\}$.

(a) *If* $K \in$ ETOL, *then* $K, K_1, K_2 \in$ EDTOL.

(b) *If* $K \in$ RP, *then* $K, K_1, K_2 \in$ IP.

(c) *If* $K \in$ ED, *then* $K, K_1, K_2 \in$ ED.

Proof. The proof is analogous to the proof of Theorem 2.

Remark. Note that it is not true that $K \in$ E0L implies that $K_2 \in$ HD0L. Let $\Sigma = \{a, b\}$, $K_1 = K_2 = \Sigma^*$, and $f : K_1 \to K_2$ be defined by $w \in K_1 : f(w) = \text{mir}(w)$. (mir($w$) denotes the mirror-image of w.)

$K = \{w \# f(w) | w \in K_1\} \in$ E0L because it is generated by the following E0L system.

$$(\{S, a, b, \#\}, \{S \to aSa, S \to bSb, S \to \#, a \to a, b \to b, \# \to \#\}, S, \{a, b, \#\}).$$

But Σ^* is not a HD0L language.

Instead of having a special marker # which divides the words into two parts, we could have disjoint alphabets such that the words are concatenations of words in the alphabets.

THEOREM 4. *Let* $\Sigma_1, \Sigma_2, \cdots, \Sigma_n$ *be n alphabets, not necessarily disjoint, and let* $f_i : \Sigma_1^* \to \Sigma_i^*$, $2 \leqq i \leqq n$, *be homomorphisms. Let* $K \subseteq \Sigma_1^*$ *and* $c_n'(K) = \{wf_2(w)f_3(w) \cdots f_n(w) | w \in K\}$. *Then* EDT0L *and* HD0L *are closed under the operator* c_n'.

Proof. The conclusion is easy to check.

If f_i is bijective for $2 \leqq i \leqq n$, then we will denote the operator by c_n. Note that ED, ER and IP are not closed under c_n'.

LEMMA 3. *Let* Σ_1 *and* Σ_2 *be two disjoint alphabets and let* $K_1 \subseteq \Sigma_1^*, K_2 \subseteq \Sigma_2^*$. *Let f be a bijective function from* K_1 *onto* K_2. *Let* $K = \{wf(w) | w \in K_1\}$. *If* $K \in$ ER, *then* $K \in$ SR (= E0L).

Proof. Let $K = L_E(R)$, where $R = (\Sigma; \xi_1, \xi_2, \cdots, \xi_n; F_1, F_2, \cdots, F_n; \alpha_1, \alpha_2, \cdots, \alpha_n)$ is a recurrence system satisfying the properties of Proposition 1. Assume that there exist integers $1 \leqq i, i_1, \cdots, i_m \leqq n$, $l, q > 2$, words $w_k \in \{\xi_1, \cdots, \xi_{i-1}, \xi_{i+1}, \cdots, \xi_n\}^*$ for $1 \leqq k \leqq l$, and words $w_1^{(k)}, w_2^{(k)} \in \{\xi_1, \cdots, \xi_n\}^*$ for $1 \leqq k \leqq m$ such that

(i) $w_1\xi_1 w_2 \xi_i w_3 \cdots \xi_i w_l \in F_{i_1}$;
(ii) $i_m = n$;
(iii) $w_1^{(k)} \xi_{i_k} w_2^{(k)} \in F_{i_{k+1}}$ for $1 \leqq k \leqq m$;
(iv) if $\xi_r \in \min(w_1 w_2 \cdots w_l)$, then $E_r(q) \neq \varnothing$;
(v) if $\xi_r \in \min(w_1^{(k)} w_2^{(k)})$, then $E_r(q+k) \neq \varnothing$ for $1 \leqq k < m$;
(vi) there exist words $v_1, v_2 \in E_i(q)$.

Then there exist words $x_k \in \Sigma^*$, $1 \leqq k \leqq l$, $w, \bar{w} \in \Sigma_1^*$ such that $x_1 v_1 x_2 v_1 \cdots v_1 x_l = wf(w)$ and $x_1 v_2 x_2 v_2 \cdots v_2 x_l = \bar{w} f(\bar{w})$. Since $w, \bar{w} \in \Sigma_1^*$ and $f(w), f(\bar{w}) \in \Sigma_2^*$ and f is a bijection, we must have that $v_1 = v_2$. Since the conclusion of our assumption is that $v_i = v_2$, it does not matter whether we substitute in parallel or not. Therefore $K = L(R)$ and $K \in$ SR (= E0L).

THEOREM 5. *Let* Σ_1, Σ_2 *and* Σ_3 *be disjoint alphabets and let* $K_i \subseteq \Sigma_i^*$, $1 \leqq i \leqq 3$. *Let* $f : K_1 \to K_2$ *and* $g : K_1 \to K_3$ *be bijective functions. Let* $K = \{wf(w)g(w) | w \in K_1\}$.

(a) *If* $K \in$ ET0L, *then* $K_1, K_2, K_3, K \in$ EDT0L.
(b) *If* $K \in$ E0L, *then* $K_1, K_2, K_3, K \in$ HD0L.
(c) *If* $K \in$ ER, *then* $K_1, K_2, K_3, K \in$ HD0L.

Proof. (a) In [3] it is shown that if $\{wf(w) | w \in K_1\} \in$ ET0L, then K_1, K_2, $\{wf(w) | w \in K_1\} \in$ EDT0L. This statement is stronger than the one in this theorem.

(b) The proof is quite similar to the proof of Theorem 2 (b).

(c) This follows from Lemma 3 and part (b) of this theorem.

THEOREM 6. *Let* Σ_1, Σ_2 *be disjoint alphabets and let* $K_1 \subseteq \Sigma_1^*$, $K_2 \subseteq \Sigma_2^*$. *Let* $f : K_1 \to K_2$ *be a monotone bijective function, monotone in the sense that* $|x| \leqq |y|$ *implies that* $|f(x)| \leqq |f(y)|$. *Let* $K = \{wf(w) | w \in K_1\}$.

(a) *If* $K \in$ RP, *then* $K \in$ LIN *and* $K_1, K_2 \in$ RG.
(b) *If* $K \in$ ED, *then* $K \in$ LIN *and* $K_1, K_2 \in$ RG.

Proof. (a) Let $\Sigma = \Sigma_1 \cup \Sigma_2$. Let $K = L(G)$, where $G = (V, \Sigma, P_u, P_o, S)$ is a Russian parallel context-free grammar. We will assume that there are no useless letters in V.

Assume $A \in V$ and $A \overset{*}{\Rightarrow} w_1$, $A \overset{*}{\Rightarrow} w_2$ for some $w_1, w_2 \in \Sigma^*$. We have then $S \overset{*}{\Rightarrow} x_1 A x_2 \overset{*}{\Rightarrow} v_1 A v_2 A \cdots A v_k$ for some $x_1, x_2 \in (V \cup \Sigma)^*$ and $v_i \in \Sigma^*$, $1 \leqq i \leqq k$, and therefore $v_1 w_1 v_2 w_1 \cdots w_1 v_k \in L(G)$ and $v_1 w_2 v_2 w_2 \cdots w_2 v_k \in L(G)$. Since f

is a monotone bijection, we have that if $w_1 \neq w_2$, then A can occur at most once in a sentential-form and w_1, w_2 cannot both be words in Σ_1^* or Σ_2^*.

Now construct $H = (V, \Sigma, P, S)$ as follows. If $(A, x_1Bx_2) \in P_u \cup P_o$ for some $x_1, x_2 \in (V \cup \Sigma)^*$ and $B \in V$ such that B can generate more than one word, then $(A, v_1Bv_2) \in P$, where $x_1 \overset{*}{\Rightarrow} v_1 \in \Sigma_1^*$ and $x_2 \overset{*}{\Rightarrow} v_2 \in \Sigma_2^*$. Note that x_1, x_2, B, v_1, v_2 are unique.

If $(A, x) \in P_u \cup P_o$ and no symbol in x can generate more than one word, then $(A, v) \in P$ where $x \overset{*}{\Rightarrow} v$. Note again that v is unique.

It is now clear that $L(G) = L(H)$ and that implies that $K \in \text{LIN}$ and K_1, $K_2 \in \text{RG}$.

(b) This is similar to (a) when we observe that ED = RPLL.

Let Σ_1, Σ_2 and Σ_3 be three disjoint alphabets. Let $K_i \subseteq \Sigma_i^*$, $1 \leqq i \leqq 3$, and let $f_i : K_1 \to K_i$, $i = 2, 3$, be length preserving isomorphisms from K_1 onto K_i. Let $K = \{wf_2(w)f_3(w) | w \in K_1\}$.

Using Theorems 5 and 6, we can get Fig. 6.

If two nodes labeled X and Y are connected by an oriented edge, then $K \in X$ implies that $K_1 \in Y$.

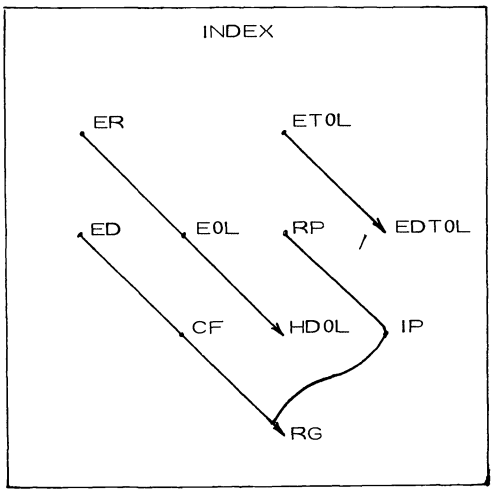


FIG. 6

5. Applications. In this section we will show how the theorems in § 4 can be used to solve some of the open relations between families occurring in Fig. 4.

It is known from the literature that mutual incomparability holds between families X and Y if there is no path from X to Y or from Y to X in Fig. 4 with the following exceptions:

(ED, EDTOL), (ED, ETOL), (ER, EDTOL), (ER, ETOL), (ED, IP), (ED, RP), (ER, IP), and (ER, RP).

Mutual incomparability between

(ED, EDTOL), (ED, ETOL), (ER, EDTOL), and (ER, ETOL)

follows by the following theorem.

THEOREM 7. ED $\not\subseteq$ ET0L *and* EDT0L $\not\subseteq$ ER.

Proof. In [2] it is proved that there exist context-free languages which are not EDT0L. Let L be such a language. Then by using Theorems 1 and 2, we get that $c^2_{\#}(L)$ belongs to ED but not to ET0L. Hence ED $\not\subseteq$ ET0L. Let $K \subseteq \Sigma_1^*$ be a language in EDT0L\HD0L (e.g., $\{a, b\}^*$) and let $f_i : \Sigma_1^* \to \Sigma_i^*$, $i = 2, 3$, be isomorphisms, where Σ_1, Σ_2 and Σ_3 are disjoint alphabets. Then by Theorems 4 and 5, we get $\{wf_1(w)f_2(w) | w \in K_1\} \in$ EDT0L\ER.

COROLLARY. ED $\not\subseteq$ RP.

The remaining open problem is now whether or not IP is contained in ER.

Acknowledgment. I wish to thank Joost Engelfriet and Arto Salomaa for many useful discussions and comments.

REFERENCES

[1] A. EHRENFEUCHT AND G. ROZENBERG, *The equality of E0L languages and codings of 0L languages*, Internat. J. Comput. Math., 4 (1974), pp. 95–104.

[2] ———, *On some context-free languages that are not deterministic ET0L languages*, Dept. of Computer Sci. Tech. Rep. CU-CS-047-74, Univ. of Colo., Boulder, 1974.

[3] A. EHRENFEUCHT, G. ROZENBERG AND S. SKYUM, *A relationship between ET0L and EDT0L languages*, Dept. of Computer Sci. DAIMI-PB40, Univ. of Aarhus, Aarhus, Denmark, 1974.

[4] S. GINSBURG AND H. G. RICE, *Two families of languages related to ALGOL*, J. Assoc. Comput. Mach., 9 (1962), pp. 350–371.

[5] S. GINSBURG, *The Mathematical Theory of Context-free Languages*, McGraw-Hill, New York, 1966.

[6] G. T. HERMAN, *A biologically motivated extension of ALGOL-like languages*, Information and Control, 22 (1973), pp. 487–502.

[7] G. T. HERMAN AND G. ROZENBERG, *Developmental systems and languages*, North-Holland, Amsterdam, to appear.

[8] M. LEVITINA, *O nekotoryh grammatikah s pravilami globalnoi podstanovki*, Akad. Nauk SSSR Nauchno-Tekhn. Inform. Ser. 2, (1973), pp. 32–36.

[9] M. NIELSEN, G. ROZENBERG, A. SALOMAA AND S. SKYUM, *Nonterminals, homomorphisms and codings in different variations of 0L systems. I: Deterministic systems, II: Nondeterministic systems*, Acta Informat., 3 (1974), pp. 357–364 and 4 (1974), pp. 87–106.

[10] G. F. ROSE, *An extension of ALGOL-like languages*, Comm. ACM, 7 (1964), pp. 52–61.

[11] G. ROZENBERG AND A. SALOMAA, *L systems*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1974.

[12] A. SALOMAA, *Formal Languages*, Academic Press, New York, 1973.

[13] ———, *Parallelism in rewriting systems*, Automata, Languages and Programming, Lecture Notes in Computer Science, No. 14, 1974, pp. 523–533.

[14] R. SIROMONEY AND K. KRITHIVASAN, *Parallel context-free languages*, Information and Control, 24 (1974), pp. 155–161.

[15] S. SKYUM, *On extensions of ALGOL-like languages*, Ibid., 26 (1974), pp. 82–93.

[16] ———, *Parallel context-free languages*, Ibid., 26 (1974), pp. 280–285.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

RANKING ALGORITHMS: THE SYMMETRIES AND COLORATIONS OF THE n-CUBE*

JAY P. FILLMORE AND S. G. WILLIAMSON†

Abstract. This paper discusses determination of the ranks, in lexicographic lists, of the symmetries and proper symmetries of the n-cube and of its colorations by r colors, up to these symmetries.

Key words. combinatorial algorithms, ranking algorithms, serial numbering schemes, n-cube

1. Introduction. Classically, "enumerative combinatorics" has been concerned with counting rather than listing sets of combinatorial structures. A systematic discussion of basic questions on enumeration from the latter point of view was initiated by D. H. Lehmer in *The machine tools of combinatorics* [1]: a set of combinatorial structures should be analyzed by first giving an algorithm for its generation, and hence linearly ordering it, and second giving a ranking or "serial numbering" function, an easily computable function giving the rank or position of each structure within the linear order. Carrying out such an analysis is generally more difficult than merely determining the cardinality of a set of structures; its importance is amply indicated by Lehmer. A knowledge of the ranking function is essential for the effective use of an algorithm, as the exponential size of the outputs of many algorithms prevents their entire execution.

This paper deals with a classical problem, coloring the n-cube up to symmetries, from Lehmer's point of view. We present the ranking functions for four algorithms concerning the n-cube. A listing is:

1. the group of all symmetries,
2. the group of proper symmetries,
3. the colorations by r colors up to all symmetries,
4. the coloration by r colors up to proper symmetries.

In each case, the algorithm is the listing of the structures in a suitable lexicographic order.

2. The groups of the n-cube. Let G and H be groups acting on the sets A and B, respectively. G acts on the Cartesian product $A \times B$ via the first factor, $g \cdot (a, b) = (g \cdot a, b)$, a in A, etc.; the set H^A of all functions f from A to H is a group under pointwise operations and it acts on $A \times B$ by $f \cdot (a, b) = (a, f(a) \cdot b)$. Together G and H^A generate the wreath product, the semidirect product $G \cdot H^A$, in which H^A is normal, and which acts on $A \times B$.

The set of $2n$ faces of the n-cube is the Cartesian product of the set $A = \{1, 2, \cdots, n\}$ of n axes and a set $B = \{1, -1\}$ of two elements which distinguish faces associated with an axis; that is, the centers of the faces are $\pm e_i$, $e_1 = (1, 0, \cdots, 0)$, etc. The group of all $n!2^n$ symmetries of the n-cube is the wreath product $G \cdot H^A$, where G is the group of all permutations of A and H that of B. Denoting by σ an element of G and by $(\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n)$, $\varepsilon_i = \pm 1$, a function in H^A, $(\sigma, (\varepsilon_1, \cdots, \varepsilon_n))$ is the symmetry $e_i \to \varepsilon_{\sigma^{-1}(i)} e_{\sigma^{-1}(i)}$.

* Received by the editors August 22, 1974, and in revised form June 1, 1975.

† Department of Mathematics, University of California, San Diego, La Jolla, California 92037.

The research of the second author supported by the National Science Foundation under Grant GJ43333.

List the elements of G and H^A in lexicographic order, and those of $G \cdot H^A$ in the product lexicographic order.

The lexicographic rank g, $0 \leqq g < n!$, of the permutation

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix} = a_1 a_2 \cdots a_n$$

is

$$g = a_1' \cdot (n-1)! + a_2' \cdot (n-2)! + \cdots + a_{n-1}' \cdot 1!$$

where a_i', $0 \leqq a_i \leqq n-i$, is the rank of a_i in $a_i a_{i+1} \cdots a_n$ [1, p. 20].

The rank h, $0 \leqq h < 2^n$, of the function $(\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n)$ is

$$h = \varepsilon_1' \cdot 2^{n-1} + \varepsilon_2' \cdot 2^{n-2} + \cdots + \varepsilon_{n-1}' \cdot 2 + \varepsilon_n',$$

where $\varepsilon_i' = 0$ or 1 as $\varepsilon_i = 1$ or -1.

The rank N, $0 \leqq N < n!2^n$, of the symmetry $(a_1 \cdots a_n, (\varepsilon_1, \cdots, \varepsilon_n))$ is

$$N = g \cdot 2^n + h.$$

For example: the 118th symmetry of the 4-cube has rank $N = 117$.

$$117 = 7 \cdot 2^4 + 5,$$

so $g = 7$, $h = 5$;

$$7 = 1 \cdot 3! + 0 \cdot 2! + 1 \cdot 1!,$$

so $a_1' a_2' a_3' = 101$, $a_1 a_2 a_3 a_4 = 2143$;

$$5 = 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1 \cdot 1,$$

so $(\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4) = (1, -1, 1, -1)$.

The 118th symmetry is

$$(2143, (1, -1, 1, -1)) \quad \text{or} \quad \begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

The group of all $n!2^{n-1}$ proper symmetries of the n-cube consists of those symmetries $(\sigma, (\varepsilon_1, \cdots, \varepsilon_n))$ for which σ and $(\varepsilon_1, \cdots, \varepsilon_n)$ have like parity $\varepsilon_1 \cdot \cdots \cdot \varepsilon_n$.

List the elements of G and H^A as before and pair those of like parity.

The parity of the element of G of rank g is $(-1)^{[(g+1)/2]}$. If we observe that G consists of n lists of permutations on $n-1$ symbols, each list prefixed by $1, 2, \cdots,$ or n, a proof is obtained by induction.

The parity of the element of H^A of rank h is $p(h) = \varepsilon_1 \cdot \cdots \cdot \varepsilon_n$, where $h = \varepsilon_1' \cdot 2^{n-1} + \varepsilon_2' \cdot 2^{n-2} + \cdots + \varepsilon_{n-1}' \cdot 2 + \varepsilon_n'$, with $\varepsilon_i' = 0$ as $\varepsilon_i = 1$ or -1. $p(h)$ may be defined recursively by:

$$p(0) = 1, \quad p(1) = -1 \quad \text{and} \quad p(h) = -p(h'),$$

where

$$h' = h - (\text{largest power of 2 not exceeding } h).$$

The pairs of ranks 01, 23, 45, $\cdots$ have one rank of each parity. It follows that the kth element of H^A of a given parity appears in the kth pair. Of the elements of parity 1, the one of rank k appears for $h = 2k + (1 - p(k))/2$; of the elements of parity -1, the one of rank k appears for $h = 2k + (1 + p(k))/2$.

The rank N, $0 \leqq N < n!2^{n-1}$, of the symmetry $(\sigma, (\varepsilon_1, \cdots, \varepsilon_n))$ in the lexicographic list of proper symmetries is

$$N = g \cdot 2^{n-1} + k,$$

where g is the rank of σ, $0 \leqq g < n!$ and k is the rank of $(\varepsilon_1, \cdots, \varepsilon_n)$ in the list of functions of the same parity as that of σ, $0 \leqq k < 2^{n-1}$.

For example, the 118th proper symmetry of the 4-cube has rank $N = 117$.

$$117 = 14 \cdot 2^3 + 5,$$

so $g = 14$, $k = 5$;

$$14 = 2 \cdot 3! + 1 \cdot 2! + 0 \cdot 1!,$$

so $a_1'a_2'a_3' = 210$, $a_1a_2a_3a_4 = 3214$.

3214 has parity $(-1)^{[(14+1)/2]} = -1$.

$$h = 2 \cdot 5 + \frac{1 + p(5)}{2} = 11,$$

$$11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2 + 1 \cdot 1,$$

so $(\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4) = (-1, 1, -1, -1)$.

The 118th proper symmetry is

$$(3214, (-1, 1, -1, -1)) \quad \text{or} \quad \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

Other algorithms for listing the elements of G and H^A lead similarly to the listing of the symmetries and proper symmetries of the n-cube. The method of adjacent marks [1, p. 22] lists the permutations in G with alternating parity. The subsets of A may be listed first by cardinality and then lexicographically within sets of equal cardinality. A ranking function for the latter may be based on the combinatorial representation of integers [1, p. 28].

3. The colorations of the n-cube. Let S be a set. H acts on the sets S^B of all functions f from B to S via the action on B, $(h \cdot f)(b) = f(h^{-1} \cdot b)$; its orbit space we denote S^B/H. Likewise, G acts on $(S^B/H)^A$ via the action on A with orbit space $(S^B/H)^A/G$, and $G \cdot H^A$ acts on $S^{A \times B}$ via the action on $A \times B$ with orbit space $S^{A \times B}/G \cdot H^A$. There is a well-known canonical bijection between these latter two orbit spaces, as made evident in the geometric examples below. (For a

proof, note that the canonical bijection $S^{A\times B}/G\cdot H^A=(S^B/H)^A/G$ is given by

$$f \bmod G\cdot H^A \to (\text{the function } a'\to f(a',\cdot) \bmod H) \bmod G.$$

Functions of two variables can be viewed as functions of the first variable with values in the set of functions of the second variable; $S^{A\times B}/G\cdot H^A=(S^B)^A/G\cdot H^A$. H^A is normal in $G\cdot H^A$, so its action may be factored out first and the action of $G\cdot H^A/H^A$ second; $(S^B)^A/G\cdot H^A=((S^B)^A/H^A)/(G\cdot H^A/H^A)$. The action of H^A on functions from A to S^B is one copy of the action of H on S^B for each element of A; $(S^B)^A/H^A=(S^B/H)^A$. Finally, $G\cdot H^A/H^A=G$ is clear.)

The set of all distinct (under the symmetries in $G\cdot H^A$) labelings of the cube $A\times B$ by elements of a set S is the orbit space $S^{A\times B}/G\cdot H^A=(S^B/H)^A/G$. That is, place all distinct (under H) labeled axes onto the cube in all distinct (under G) ways.

For example, when $n=3$ and $S=\{0,1\}$, the distinct labeled axes are 00, 01, and 11. The distinct colorations of the cube, ten in number, are

00	00	00	00	11	11
00	00	01	01	01	01
00	00	11	01	01	11
00	01	01	01	11	11
00	01	11	11	11	11

Note that these are the lexicographically nondecreasing sequences of length three on the symbols 00, 01, and 11.

In general, when $S=\{0,1,\cdots,r-1\}$ and $G\cdot H^A$ is the group of all symmetries of the n-cube, the distinct labeled axes S^B/H are described by symbols ij, $0\leqq i\leqq j<r$; since G consists of all permutations of A, the colorations of the n-cube up to all symmetries, $S^{A\times B}/G\cdot H^A=(S^B/H)^A/G$, are described by lexicographically nondecreasing sequences of length n on the symbols ij. There are $l=\binom{r}{2}+r$ symbols ij and $\binom{l+n-1}{n}$ nondecreasing sequences of length n on these l symbols; hence there are

$$\binom{\binom{r}{2}+r+n-1}{n}$$

colorations of the n-cube by r colors, up to all symmetries.

There is a bijection between nondecreasing sequences $x_1x_2\cdots x_n$, $0\leqq x_1\leqq x_2\leqq\cdots\leqq x_n<l$, and $(l-1)$- or n-element subsets of an $(n+l-1)$-element set:

$$x_1x_2\cdots x_n \to \overbrace{0\ 0\ 0\ \cdots\ 0}^{d_1}\ 1\ \overbrace{0\ 0\ \cdots\ 0}^{d_2}\ 1\ \cdots\ 1\ \overbrace{0\ 0\ \cdots\ 0}^{d_l}$$

$$0\ 1\ 2 \qquad\qquad n+l-1$$

where $d_x \geqq 0$ is the number of x's in $x_1x_2 \cdots x_n$. The subsets are specified by the strictly increasing sequence of the ranks of the $l-1$ 1's or the n 0's. When the sequences $x_1x_2 \cdots x_n$ are listed in lexicographic increasing order:

(i) the sequences of length n of the ranks of the 0's, read left to right, appear in lexicographic increasing order.

(ii) the sequences of length $l-1$ of the ranks of the 1's, read right to left, appear in colexicographic increasing order.

Every integer N, $0 \leqq N < \binom{c}{k}$, has a unique combinatorial representation

$$N = \binom{b_1}{1} + \binom{b_2}{2} + \cdots + \binom{b_k}{k},$$

with $0 \leqq b_1 < b_2 < \cdots < b_k < c$ [1, p. 28]; as N increases, all such strictly increasing sequences $b_1b_2 \cdots b_k$ are produced in colexicographic order (the last unequal entry is strictly larger). Similarly,

$$\binom{c}{k} - 1 - N = \binom{c-1-a_k}{1} + \binom{c-1-a_{k-1}}{2} + \cdots + \binom{c-1-a_1}{k}$$

produces all strictly increasing sequences $a_1a_2 \cdots a_k$ in lexicographic order [1, p. 28].

For example, the $l = \binom{r}{2} + r$ symbols ij, $0 \leqq i \leqq j < r$, are nondecreasing sequences of length 2 on $0, 1, 2, \cdots, r-1$. To ij corresponds

$$\overbrace{\underset{0}{1}\ \underset{1}{1}\ \underset{2}{1}\ \cdots\ 1}^{i}\ 0\ \overbrace{1\ 1\ 1\ \cdots\ 1}^{j-i}\ 0\ \overbrace{1\ 1\ 1\ \cdots\ \underset{r-1}{1}}^{r-j-i}.$$

The rank N of the strictly increasing sequence $a_1, a_2 = i, j+i$ is given by

$$\binom{r+1}{2} - 1 - N = \binom{r-j-1}{1} + \binom{r-i}{2}.$$

The colorations of the n-cube by r colors are ranked by ranking the nondecreasing sequences of length n on the l symbols ij, $0 \leqq i \leqq j < r$, lexicographically ordered.

For example, there are

$$\binom{\binom{10}{2} + 10 + 5 - 1}{5} = \binom{59}{5} = 5006386$$

colorations of the 5-cube by 10 colors. The 1000000th coloration has rank $N = 999999$.

$$\binom{59}{5} - 1 - 999999 = \binom{58-a_5}{1} + \binom{58-a_4}{2} + \binom{58-a_3}{3} + \binom{58-a_2}{4} + \binom{58-a_1}{5},$$

$$4006386 = \binom{0}{1} + \binom{30}{2} + \binom{37}{3} + \binom{47}{4} + \binom{56}{5},$$

so $a_1a_2a_3a_4a_5 = 2\ \ 11\ \ 21\ \ 28\ \ 58$.

To the strictly increasing sequence described by

$$1^2 \quad 0 \quad 1^8 \quad 0 \quad 1^9 \quad 0 \quad 1^6 \quad 0 \quad 1^{29} \quad 0$$

corresponds the nondecreasing sequence:

$$2 \quad 10 \quad 19 \quad 25 \quad 54.$$

These are the ranks of the symbols

$$02 \quad 11 \quad 22 \quad 28 \quad 99$$

in the list of 55 symbols ij, $0 \leqq i \leqq j < 10$. This is the 1000000th coloration of the 5-cube by 10 colors.

In the opposite direction, the ranks of

$$01 \quad 23 \quad 45 \quad 67 \quad 89$$

in the list of 55 symbols ij are

$$1 \quad 20 \quad 35 \quad 46 \quad 53.$$

To this corresponds the strictly increasing sequence described by

$$1 \quad 0 \quad 1^{19} \quad 0 \quad 1^{15} \quad 0 \quad 1^{11} \quad 0 \quad 1^{7} \quad 0 \quad 1,$$

so $a_1a_2a_3a_4a_5 = 1 \quad 21 \quad 37 \quad 49 \quad 57$;

$$\binom{59}{5} - 1 - N = \binom{58-57}{1} + \binom{58-49}{2} + \binom{58-37}{3} + \binom{58-21}{4} + \binom{58-1}{5},$$

$$5006386 - 1 - N = \binom{1}{1} + \binom{9}{2} + \binom{21}{3} + \binom{37}{4} + \binom{57}{5} = 4254518,$$

so $N = 751867$.

This coloration of the 5-cube by 10 colors is the 751868th.

Under the group of all symmetries of the n-cube, every coloration by r colors $0, 1, \cdots, r-1$ is described by a nondecreasing sequence of length n on the symbols ij, $0 \leqq i \leqq j < r$. Under the group of *proper* symmetries of the n-cube, every coloration is equivalent to one of the above *or* to a coloration obtained by interchanging the first i and j in a strictly increasing sequence of symbols ij, $0 \leqq i \leqq j < r$. For whenever a symbol ij with $i = j$ appears or a symbol ij is repeated, a reflection perpendicular to an axis or an interchange of two axes leaves the coloration unchanged. There are

$$\binom{\binom{r}{2}}{n}$$

strictly increasing sequences of length n on the symbols ij, $0 \leqq i < j < r$, lexicographically ordered; hence there are

$$\binom{\binom{r}{2} + r + n - 1}{n} + \binom{\binom{r}{2}}{n}$$

colorations of the n-cube by r colors, up to proper symmetries.

For the purpose of ranking these colorations, the lexicographic list of nondecreasing sequences on the symbols ij, $0 \leqq i \leqq j < r$, is followed by the lexicographic list of strictly increasing sequences on the symbols ij, $0 \leqq i < j < r$.

For example, there are

$$\left(\begin{matrix}\binom{10}{2}+10+4-1\\ 5\end{matrix}\right)+\left(\begin{matrix}\binom{10}{2}\\ 5\end{matrix}\right)=\binom{59}{5}+\binom{45}{5}$$

$$=5006386+1221759=6228145$$

colorations of the 5-cube by 10 colors. The 6000000th coloration has rank

$$N=5999999-5006386=993613$$

in the list of strictly increasing sequences of length 5 on the symbols ij, $0 \leqq i < j < 10$.

$$\binom{45}{5}-1-993613=\binom{44-a_5}{1}+\binom{44-a_4}{2}+\binom{44-a_3}{3}+\binom{44-a_2}{4}+\binom{44-a_1}{5},$$

$$228145=\binom{2}{1}+\binom{14}{2}+\binom{27}{3}+\binom{29}{4}+\binom{32}{5},$$

so $a_1a_2a_3a_4a_5=12\quad 15\quad 17\quad 30\quad 42$.

These are the ranks of the symbols

$$15\quad 18\quad 23\quad 45\quad 78$$

in the list of 45 symbols ij, $0 \leqq i < j < 10$. Hence the 6000000th coloration of the 5-cube by 10 colors is

$$51\quad 18\quad 23\quad 45\quad 78.$$

In the opposite direction, the ranks of the symbols

$$01\quad 23\quad 45\quad 67\quad 89$$

in the list of 45 symbols ij are

$$0\quad 17\quad 30\quad 39\quad 44,$$

so $a_1a_2a_3a_4a_5=0\quad 17\quad 30\quad 39\quad 44$;

$$\binom{45}{5}-1-N=\binom{44-44}{1}+\binom{44-39}{2}+\binom{44-30}{3}+\binom{44-17}{4}+\binom{44-0}{5},$$

$$1221759-1-N=\binom{0}{1}+\binom{5}{2}+\binom{14}{3}+\binom{27}{4}+\binom{44}{5}=1103932,$$

so $N=117826$.

Hence

$$10\quad 23\quad 45\quad 67\quad 89$$

has rank

$$5006386+117826=5124212$$

and is the 5124213th coloration of the 5-cube by 10 colors.

The number of colorations of the n-cube up to these symmetries may also be obtained from an extension of Pólya's theory of counting; such a counting neither produces the structures nor ranks them.

REFERENCE

[1] D. H. LEHMER, *The machine tools of combinatorics*, Applied Combinatorial Mathematics, E. F. Beckenbach, ed., John Wiley, New York, 1964, pp. 5–31.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

LOCAL ADJUNCT LANGUAGES AND REGULAR SETS*

LEON S. LEVY †

Abstract. The class of regular sets is shown to be contained in the class of local adjunct languages. The essential idea in the proof is an intercalation theorem for the Myhill congruence classes.

Key words. congruence, finite state machine, grammar, intercalation, language

1. Introduction. String adjunct grammars were introduced by Joshi, Kosaraju and Yamada [3] as a formalization of the base component of a transformational grammar for natural language along the lines suggested by Harris [2]. Local adjunct grammars are a special case of string adjunct grammars, defined as follows.

DEFINITION (Local adjunct grammar). Let Σ be a finite alphabet, and Σ^* the set of finite strings over Σ. A local adjunct grammar is $G=(\Sigma_c, J)$, where Σ_c is a finite set of *center strings*, and J is a finite set of *rules*. Each rule in J is of the form (x, y, p), where x and y are finite strings over Σ called the *host* and *adjunct*, respectively, and p is a *point of adjunction*. Each p is l_k or r_k, $1 \leqq k \leqq \text{length}(x)$.

In the grammar, a *syntactic class* of strings, $\Sigma(x)$, is generated by repeatedly adjoining to a string, x, any strings of syntactic class y for which there is a rule (x, y, p). The adjunction is performed to the *left* of the kth symbol of x if p is l_k, and to the *right* of the kth symbol of x if p is r_k. (As long as we are adjoining to x, the adjunction rule refers to the symbol in the host string.)

Example. $J=\{(aab, ac, r_1), (aab, aab, l_2)\}$. Using the first rule, ac may be adjoined to aab, yielding $\underline{aacab}$, where we have underlined the host to identify its symbols. Then, using the second rule, $aacab$, which is of syntactic class aab, may be adjoined to $\underline{aacab}$ to yield $\underline{aacaacabab}$.

DEFINITION (Local adjunct language). The language, $L(G)$, generated by a local adjunct grammar, G, is the union of the syntactic classes of all center strings. (For a more formal definition, see Levy [5]).

Example. $\Sigma_c=\{ab\}$, $J=\{(ab, ab, r_1), (ab, ab, r_2)\}$ yields the Dyck language over matching symbols a and b.

In Joshi, Kosaraju and Yamada [3], it is shown that the class of local adjunct languages is properly contained in the class of context-free languages. The relationship of the class of local adjunct languages and the class of regular sets was left open. (An attempt to answer this question led to a conjecture about null symbols, which was shown in Levy [5] to be incorrect.)

In § 2, the basic ideas of congruences are presented, and a new intercalation theorem is presented. This theorem is used, in § 3, to show that every regular set is a local adjunct language.

2. Congruences. A finite state machine (see Ginzburg [1]) is a system $M=(S, \Sigma, \delta, s_0, F)$ where S is a finite state set, Σ is a finite alphabet, δ is a mapping, $\delta : S \times \Sigma \to S$, s_0 is a distinguished state in S, the start state, and $F \subseteq S$ is the set of

* Received by the editors April 21, 1975, and in revised form September 3, 1975.

† Department of Statistics and Computer Science, University of Delaware, Newark, Delaware 19711.

final states. We can extend δ to a mapping $\hat{\delta}$. $\hat{\delta} : S \times \Sigma^* \to S$ by $\hat{\delta}(s, \lambda) = s$, where λ is the null string, and $\hat{\delta}(s, xy) = \delta(\hat{\delta}(s, x), y)$.

Alternatively, the mappings can be parametrized, to obtain a set of mappings $\{(x)^M | x \text{ in } \Sigma^*\}$ where each $(x)^M : S \to S$ and where $(x)^M : s_1 \to s_2$ iff $\hat{\delta}(s_1, x) = s_2$. Since there are at most $|S|^{|S|}$ distinct mappings associated with a given finite state machine, the mappings set is finite, and of cardinality $\leqq |S|^{|S|}$. Two strings x, y are said to be *congruent*, $x \equiv y$, iff $(x)^M = (y)^M$. Analogous to the well-known intercalation theorems, we have the following.

THEOREM 1. *Let M be a finite state machine with k states and input alphabet Σ, and let $n_0 = k^k$. If w is a string over Σ such that* length$(w) \geqq n_0$, *then $w = uvx$, $v \neq \lambda$,* length$(ux) < n_0$ *and $u(v)^r x \equiv w$, $r \geqq 0$.*

Proof. There are at most n_0 different mappings associated with different strings. If $w = a_1 a_2 \cdots a_N$, each a_i in Σ, $w/i \triangleq a_1 a_2 \cdots a_{N-i}$, then at least two of $\{w/0, w/1, \cdots, w/n_0\}$ must have the same associated state mapping. Suppose i, j are the smallest i and j, $j > i$, such that w/i and w/j have the same associated state mapping. Then $a_1 a_2 \cdots a_i a_{j+1} \cdots a_N \equiv w$, and the theorem is satisfied with $u = a_1 a_2 \cdots a_i$, $v = a_{i+1} \cdots a_j$ and $x = a_{j+1} \cdots a_N$.

Remarks. 1. This theorem is just the usual intercalation theorem for finite state machines applied to the semigroup machine.

2. By analogy, the canonical system of Buchi for regular sets can be applied to the semigroup machine to obtain a set of congruence reduction rules. (See Rounds [6].)

3. The statement of the intercalaction theorem follows the usual form. Strictly speaking, in the local adjunct language proof we should use the following: if length$(w) > n_0$, then $w = uvx$, $u \neq \lambda$, $v \neq \lambda$, length$(ux) \leqq n_0$ and $u(v)^r x \equiv w$, $r \geqq 0$.

3. Application to the local adjunct language problem. It remains to be shown that the string reduction rule implicit in the intercalation theorem given above can be incorporated into a local adjunct grammar. The main definition required is the following.

DEFINITION. Let x, y be strings over Σ, and $(x)^M$, $(y)^M$ be the corresponding associated state mappings in a finite state machine, M. Also, let $x = x_1 x_2$ and length $(x_1) = k$. Then (x, y, k) is said to be *free*, or "y is free in x at k" if $(x_1)^M = (x_1 y)^M$.

Theorem 1 can now be rephrased as follows: let M be a finite state machine with k states, and input alphabet Σ, and let $n_0 = k^k$. If w is a string over Σ such that length$(w) > n_0$, then $w = uvx$, $u \neq \lambda$, $v \neq \lambda$, length$(ux) \leqq n_0$ and *v is free in w at* length(u).

Note that given a finite state machine, M, strings x, y, and index, k, one can readily decide if y is free in x at k by computing the appropriate mappings.

PROPOSITION 1. *If y_1 is free in x at k and y_2 is free in x at k, then $z \in \{y_1, y_2\}^*$ is free in x at k.*

PROPOSITION 2. *If y_1 is free in x at k, y_2 is free in y_1 at l, and $S(y_1, y_2, l)$ denotes the result of adjoining y_2 to y_1 at l, then $S(y_1, y_2, l)$ is free in x at k.*

Remarks. The distinction between adjunction to the left of the $(l+1)$st symbol and adjunction to the right of the lth symbol is not essential here as it is

elsewhere. (See Levy [4].) Accordingly, we have only noted the point of adjunction. However, the ability to adjoin between any pair of symbols, to the left of the initial symbol, and to the right of the final symbol, is needed to obtain the full power of local adjunct languages.

Propositions 1 and 2 assure us that freeness of adjunction allows arbitrary sequences of adjunctions because in every case the state mapping associated with the resultant string is the same. Hence a string adjunct grammar may have a rule (x, y, p). Providing that y is free in x at p will assure that all strings generated by this rule will have the same associated state mapping as x.

Now the construction of a string adjunct grammar for a given finite state machine is straightforward. Let M have S states, and let $k = S^S$. Each string of length $\leqq k$ which is accepted by M is a center string. For each pair of strings $1 \leqq x$, $y \leqq k$, the adjunction rule (x, y, r_p) is in J if y is free in x at p.

We have the following lemma.

LEMMA 1. *Let G be the grammar constructed above. Then for each string w, there is a string w', length$(w') \leqq k$, such that w is in the syntactic class of w'.*

Proof. If length $(w) \leqq k$, then choose $w' = w$. Assume that the lemma is true for strings of length less than n. Let w be a string of length n. Then by Theorem 1, $w = a_1 a_2 \cdots a_r a_{r+1} \cdots a_w a_{s+1} \cdots a_n$ and there is a $w_1 = a_1 \cdots a_r a_{s+1} \cdots a_n$, length$(w_1) \leqq k$ and $w_1 \equiv w$. Also, by the inductive assumption, $a_{r+1} \cdots a_s$ is in the syntactic class of some string w_2 and length$(w_2) \leqq k$. By the construction of G there is a rule adjoining w_2 to w_1. Hence w is of syntactic class w_1 and length$(w_1) \leqq k$. Q.E.D.

Clearly, each string generated by the grammar leads from the start state to a final state, since by construction each string generated is congruent to a center string, and each center string leads to a final state.

Conversely, assume that x is accepted by M. If length$(x) \leqq k$, then x is a center string. If length $(x) > k$, then by Theorem 1 it can be generated from a center string by adjoining a string v to a host v' whose length is less than or equal to k. But by Lemma 1, v is in the syntactic class of some v'' such that length$(v'') \leqq k$. And by the construction of the grammar, there is a rule of adjoining v'' to v'. Hence v is generated in the grammar.

Theorem 2 below follows from the above construction and further argument.

THEOREM 2. *Every regular set is a local adjunct language.*

4. Conclusion. The class of regular sets has been shown to be properly contained in the class of local adjunct languages. This answers an open question in the theory of adjunct languages.

Acknowledgment. The referees' careful reading of the paper and their helpful suggestions are appreciated.

REFERENCES

[1] A. GINZBURG (1968), *Algebraic Theory of Automata*, Academic Press, New York.
[2] Z. S. HARRIS (1968), *Mathematical Structure of Language*, Mouton, The Hague.
[3] A. K. JOSHI, S. R. KOSARAJU AND H. YAMADA (1972), *String adjunct grammars*, Information and Control, 21, pp. 93–116.
[4] L. S. LEVY (1972), *Tree automata and adjunct grammars*, Moore School Rep 72–16, Univ. of Pa., Philadelphia.

[5] ——— (1973), *Structural aspects of local adjunct languages*, Information and Control, 23, pp. 260–287.

[6] W. C. ROUNDS (1970), *Tree oriented proofs of some theorems on context-free and indexed languages*, 2nd ACM Symp. on the Theory of Computing, pp. 109–116.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

SYNTHESIZING A RESPONSE FUNCTION WITH A FEEDBACK SCHEDULING ALGORITHM*

ARTHUR BERNSTEIN†

Abstract. An algorithm is described for synthesizing a specified response function within the framework of a feedback scheduler. The scheduler uses a single queue and feeds back jobs to positions which are dependent on their attained service.

Key words. feedback scheduler, response function, synthesis, time-shared system

1. Introduction. A number of authors have analyzed the behavior of algorithms for scheduling jobs in a computer system [1]–[7]. Models of such systems include an arrival process, a server and a system of queues. A primary goal of such work is to describe the performance of an algorithm with a response function, $T(t)$, which relates the service time of a job, t, to the expected total time a job spends in the system.

In most models, the arrival process is assumed to be Poisson. The server represents the central processing unit and, in most cases, a preemptive resume strategy is assumed: a job is given a Q second quantum of service and if it does not complete, it is returned to a particular position in the system of queues so that service can be resumed at a later time. In addition to the simple round robin algorithm (RR), two queuing disciplines of particular interest to this paper have been considered. These are the feedback discipline with multiple queues (FBMQ) [2] and the feedback discipline with a single queue (FBSQ) [4].

As shown in Fig. 1, FBMQ uses an array of queues. At each decision point, the job at the head of the lowest numbered, nonempty queue is selected and receives a quantum of service. If it completes in the allocated time it leaves the system; if not it is returned to the tail of the next higher numbered queue. New arrivals are generally placed at the tail of queue 0. Thus a job which is linked to queue i has visited the server i times. In the most general formulation of the algorithm, the size of the quantum allocated to a job depends upon the queue to which it is linked. In any case, the number of queues required can be computed from the quantum sizes and the maximum service time which may be required by any new arrival.

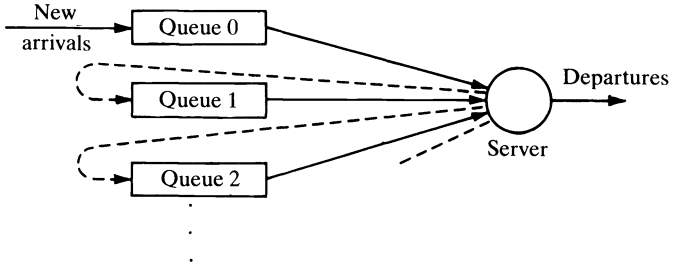


FIG. 1. *FBMQ model*

* Received by the editors November 11, 1974, and in final revised form July 11, 1974.

† Department of Computer Science, State University of New York, Stony Brook, New York, 11790. This work was supported in part by the National Science Foundation under Grant J042562.

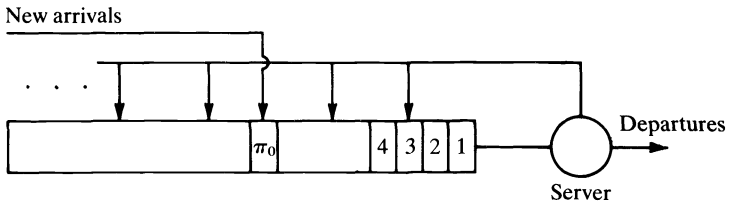


FIG. 2. *FBSQ model*

The FBSQ, as shown in Fig. 2, is related to the FBMQ since in both, the location within the queuing system to which a job is fedback is a function of the job's attained service. In this case, however, there is only a single queue whose positions are numbered 1, 2, 3, $\cdots$ starting at the queue head. The job in position 1, J, is selected for service. If, at the end of the allocated quantum, J does not terminate and if its attained service at that time is iQ, then the following steps are taken to reorder jobs in the queue. J is removed from the queue; all jobs in positions numbered less than or equal to π_i are moved forward one position; and then J is inserted into position π_i. The job in position 1 is then, once again, selected for service. Notice that jobs in positions numbered greater than π_i are not moved as a result of this event. Of course, if the length of the queue is less than π_i, J is simply returned to the end of the queue. New arrivals are inserted in the queue at position π_0. Jobs in positions numbered greater than or equal to π_0 must be moved backward one position to make room for the new arrival. Thus a discipline in which, for all i, $\pi_i = \infty$, is in reality RR while if $\pi_0 = \infty$ and, for $i > 0$, $\pi_i = 1$, we obtain the first come first served discipline (FCFS). By choosing intermediate values of π_i, it is possible to obtain a wide spectrum of disciplines.

Note that in FBMQ the attained service of the job which is selected to be served next is always less than or equal to the attained service of every other job in the system. Since this is not the case in FBSQ, one would expect that FBMQ would, in general, give better service to short running jobs at the expense of long running jobs. On the other hand, in FBMQ, jobs requiring the same amount of service and which coexist in the system tend to terminate at roughly the same time independent of their arrival times. Thus in FBMQ, one would expect that the total time in the system experienced by these jobs would be more widely dispersed than in FBSQ where jobs tend to progress at a more predictable rate independent of the arrival process. Uniform turnaround time is clearly important in systems attempting to give good service to longer running jobs. It is interesting to note that FBMQ was first used in a system whose primary goal was to provide interactive service [8] while the FBSQ model was motivated by an algorithm implemented in a system designed to provide both batch and interactive service [9].

Although the analysis of scheduling algorithms has received a good deal of attention, the synthesis problem has not. In this case, one would like to start with a specification of a desired response function, $T(t)$, and produce a corresponding algorithm. Obviously, if the arrival process and the service time requirements of jobs are given, then a desired response function will generally turn out to be unrealizable. Thus it may be too conservative (i.e., not realizable without hold-back) or too ambitious (i.e., the input load is too heavy to meet the response

specification with existing processing power). To overcome this difficulty, the description of the response function should be left incomplete and the synthesis procedure should attempt to realize that part of the function which has been specified and produce the best performance possible for the unspecified part. Alternatively, the desired performance can be specified as a bound on the response function.

The synthesis procedure is facilitated if one starts with an algorithm schema—i.e., a parameter controlled algorithm—which can be tuned to provide a wide range of response functions depending on the values chosen for the parameters. This is essentially the approach taken by Michel and Coffman [7], where a procedure is presented for generating an FBMQ algorithm to meet a performance goal specified in terms of bounds on the response function. In this case, the parameters are the quantum sizes associated with each queue. Of course, it should be realized that some schema may be better suited than others to achieve a particular performance goal.

In this paper, a synthesis procedure will be developed for the FBSQ model. The procedure is based on a partial specification of the response function and produces the set of feedback points, π_i, which determine the algorithm.

2. The model. We will assume that all jobs are identical with service times chosen from an exponential density function $f(t)$, where

$$f(t) = \mu e^{-\mu t}. \tag{1}$$

The arrival process is Poisson with average arrival rate λ_0. Since holdback is not permitted, it follows from elementary queuing theory [10] that the utilization of the server, ρ, is given by

$$\rho = \frac{\lambda_0}{\mu} \tag{2}$$

and that the steady state probability that there are n jobs in the system, $p(n)$, is given by

$$p(n) = (1-\rho)\rho^n. \tag{3}$$

Jobs will be allocated a Q second quantum of service and, if they do not terminate, will be fed back into the queue. The overhead of job switching will be ignored. We will make the approximation that each time a job reenters the queue it sees the same queue length distribution (3). More detailed analyses in the discrete [13] and continuous cases [5], [14] are consistent with such an approximation when it is assumed that the mean service time of a job is much greater than the time quantum. We will make such an assumption in what follows. Since a job being served may terminate before an entire quantum has elapsed, the average service time actually used by a job, $\bar{Q}$, when it has been allocated a quantum of service is given by [5]

$$\bar{Q} = \frac{1}{\mu}(1 - e^{-\mu Q}). \tag{4}$$

An FBSQ algorithm is specified by choosing a vector with an infinite number of components $\pi = (\pi_0, \pi_1, \pi_2, \cdots)$. π_i is an integer with value greater than zero which is used to determine the position into which a job, which has just received its ith quantum of service without departing, is returned to the queue. If the current queue length is n, then such a job is entered into position $H(i)$ where $H(i)$ is the min $\{\pi_i, n\}$. This mapping of attained service into positions in the queue does not change with time. We will say that a job in the queue with an attained service of iQ seconds is a member of group i [4].

It is convenient to restrict the form of the π-vector in several ways. First of all we assume that

$$\pi_i \geqq \pi_j \quad \text{for } i > j > 0. \tag{5}$$

This restriction is for notational convenience only since the analysis below could be carried through without it. Furthermore, it does not seem to eliminate any realistic situations, since one would normally expect a job with a large attained service to wait longer for additional service than a job with a small attained service. Secondly, we will assume that there are only a finite number, M, of distinct π_i for $i > 0$. This is also not unrealistic since queues must adhere to bounds implicit in the implementation anyway. Let ϕ_i, for $i = 1, 2, \cdots, M$, be the distinct feedback points such that $\phi_i > \phi_j$ for $i > j$. Then as a result of the previous assumption, successive components of π will be equal and it will have the form $\pi = (\pi_0, \phi_1, \cdots, \phi_1, \phi_2, \cdots, \phi_2, \cdots, \phi_M, \cdots)$. Furthermore, for $1 \leqq j \leqq M-1$, let

$$\pi_{k_{j-1}+1} = \pi_{k_{j-1}+2} = \cdots = \pi_{k_j} = \phi_j, \tag{6}$$

where $k_0 = 0$ and $\pi_j = \phi_M$ for all j, $j > k_{M-1}$.

The analysis of the FBSQ algorithm has been carried out [4], and it has been shown that the response function is piecewise linear. A typical function produced by an algorithm with three distinct feedback points and a fourth insertion point for new arrivals is shown in Fig. 3. The function is piecewise linear as a result of (6) since, for a range of attained service, jobs get fedback to the same point and

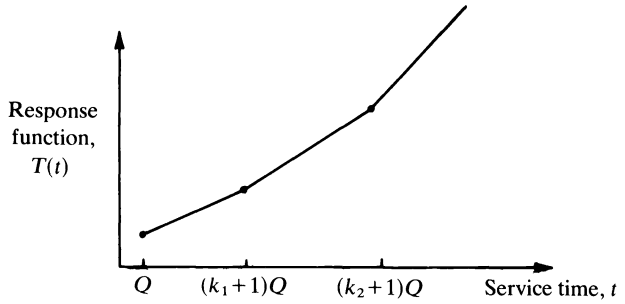


FIG. 3. *Response function of an FBSQ algorithm with 4 distinct feedback points*

therefore, on average, wait the same time for each successive quanta. Thus ϕ_1, ϕ_2 and ϕ_3 determine the slopes of the three line segments. Condition (5) assures that the function is monotonically increasing. The vertical displacement of the line segments is determined by the average time a job must wait to receive its first quantum and is thus a function of π_0.

In this paper, we will develop a synthesis technique for the special class of FBSQ algorithms obtained with the restriction $\pi_0 = \infty$ (note that this does not violate (5)). Since, in addition, we have assumed a finite number of distinct feedback points, the scheduler has the form shown in Fig. 4. The restriction on π_0 simplifies the synthesis procedure since it implies that a job in the queue never

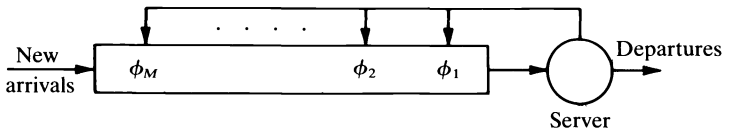


FIG. 4. *An FBSQ scheduler with a finite number of distinct feedback points and* $\pi_0 = \infty$

moves away from the server. As a result, it follows that all jobs in positions numbered greater than ϕ_M have an attained service of zero.

3. Synthesis procedure. In developing a synthesis procedure, it is convenient to decompose the response function, $T(t)$, into the time a job spends actually receiving service from the server and the average time it spends waiting in the queue. Let $W(t)$ be the average time a job requiring t seconds of service spends waiting in the queue. Since $T(t)$ is equal to $W(t)+t$, the specification for a desired response function can easily be translated into a specification for a desired waiting time function.

The synthesis procedure produces a set of M feedback points ϕ_i, $i = 1, 2, \cdots, M$, which will realize the shape of a particular piecewise linear, continuous and monotonically increasing waiting time function which has been specified. The values of k_i are obtained from the abscissas of points at which the line segments intersect. The unspecified feature of the waiting time function is its vertical displacement, $W(Q)$. This can be calculated once the ϕ_i have been determined. A mechanism for modifying $W(Q)$ should its value prove unsatisfactory is described in the next section.

Using Little's law [11], Kleinrock and Coffman [12] have obtained a result which relates the average number of jobs in the queue waiting for their $(j+1)$st quantum of service, $N(jQ)$, to the waiting time function as follows:

$$N(jQ) = \lambda_0(1 - B(jQ))\Delta(jQ), \qquad j \geqq 0, \tag{7}$$

where $B(jQ)$ is the distribution function for the service time and $\Delta(jQ) = W((j+1)Q) - W(jQ)$. Clearly, $N(jQ)$ is also the number of jobs waiting in the queue with an attained service of jQ seconds. The term $\lambda_0(1 - B(jQ))$ is simply the rate at which jobs are fedback to π_j (enter group j) and will be referred to as λ_j. This is so since $1 - B(jQ)$ is the fraction of new arrivals which require more than jQ seconds of service. Using (1) we have

$$\lambda_j = \lambda_0 e^{-\mu jQ}. \tag{8}$$

An alternate expression for $N(jQ)$ can be obtained by direct examination of the queue. Let r_i be the sum of rates at which jobs enter all groups l, for $l = k_{i-1}+1, k_{i-1}+2, \cdots, k_i$. Such jobs will actually be fed back into the queue at position ϕ_i or at the end, if ϕ_i is greater than the total queue length. Similarly, let

R_i be the rate at which jobs are fed back from the server into all groups l for $l > k_i$, plus the rate of arrival of new jobs into the system (λ_0). If the queue has length greater than ϕ_i, this reduces to the rate at which jobs are fed forward in the queue into position ϕ_i. The flow of jobs under this assumption is illustrated in Fig. 5. It follows that

$$r_i = \sum_{l=k_{i-1}+1}^{k_i} \lambda_l,$$

$$R_i = \lambda_0 + \sum_{l=i+1}^{M} r_l.$$

Note that it follows from this and (8) that

$$R_0 = \frac{\lambda_0}{1 - e^{-\mu Q}}. \tag{9}$$

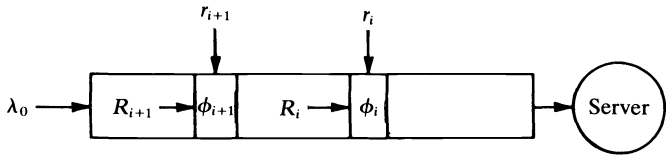


FIG. 5. *Job flow in FBSQ model*

If $\pi_j = \phi_{i+1}$, where $i, j \geqq 1$, then, since jobs never move backward in the queue, jobs with attained service jQ can never occupy positions numbered greater ϕ_{i+1}. The fraction of jobs between any two adjacent feedback points, $\phi_{m-1}+1$ and ϕ_m, where $i+1 \geqq m > 1$, with an attained service of jQ seconds is just λ_j / R_{m-1}, and the average number of these jobs in this region is given by

$$\frac{\lambda_j}{R_{m-1}} \left[\sum_{n=\phi_{m-1}+1}^{\phi_m} (n - \phi_{m-1}) p(n) + (\phi_m - \phi_{m-1}) \sum_{n=\phi_m+1}^{\infty} p(n) \right]. \tag{10}$$

In counting the number of jobs with attained service jQ in the region between ϕ_1 and the server, we must be careful not to include the job currently being served. Thus the number of such jobs in this region is given by

$$\frac{\lambda_j}{R_0} \left[\sum_{n=2}^{\phi_1} (n-1) p(n) + (\phi_1 - 1) \sum_{n=\phi_1+1}^{\infty} p(n) \right]. \tag{11}$$

Since $\pi_j = \phi_{i+1}$, an expression for $N(jQ)$ can be obtained by summing terms of the form (10) for $m = 2, 3, \cdots, i+1$ with (11). Using (3), (4) and (9) and simplifying, we get

$$N(jQ) = \lambda_j \left[\sum_{m=2}^{i+1} \frac{\rho^{\phi_{m-1}}}{R_{m-1}} \left(\frac{\rho - \rho^{s_m+1}}{1-\rho} \right) + \bar{Q} \left(\frac{\rho - \rho^{\phi_1}}{1-\rho} \right) \right], \tag{12}$$

where $s_m = \phi_m - \phi_{m-1}$ and $j > 0$.

Using (7) and (12), we can successively solve for the feedback points starting with ϕ_1. Thus assuming $\pi_1 = \phi_1$ and setting $j = 1$, we get

$$N(Q) = \lambda_1 \bar{Q} \frac{(\rho - \rho^{\phi_1})}{(1-\rho)} = \lambda_1 \Delta(Q), \tag{13}$$

which reduces to

$$\rho^{\phi_1} = \rho - \frac{(1-\rho)}{\bar{Q}} \Delta(Q). \tag{14}$$

Since ϕ_1 must satisfy $1 \leqq \phi_1 \leqq \infty$, (14) has a solution for ϕ_1 as long as

$$0 \leqq \Delta(Q) \leqq \bar{Q} \frac{\rho}{1-\rho}.$$

These bounds correspond to the two extreme cases of the model, FCFS and RR. Note that too large a specification for $\Delta(Q)$ implies holdback and that a response function with performance uniformly better than specified can be obtained.

If $\Delta(2Q) = \Delta(Q)$, then $\pi_2 = \phi_1$ and (12) yields no new information. An equation for ϕ_2 in terms of ϕ_1 can be obtained by setting $j = k_1 + 1$ and, in general, we can solve for ϕ_{i+1}, given that $\phi_1, \phi_2, \cdots, \phi_i$ have already been determined, by setting $j = k_i + 1$ in (7) and (12) and equating the right-hand sides, yielding

$$\rho^{\phi_{i+1}} = \rho^{\phi_i} - \frac{R_i}{\rho}\left[(1-\rho)\Delta((k_i+1)Q) - \bar{Q}(\rho - \rho^{\phi_1}) - \sum_{m=2}^{i} \frac{\rho^{\phi_{m-1}}}{R_{m-1}}(\rho - \rho^{s_m+1})\right]. \tag{15}$$

Once again the fact that $\phi_{i+1} \leqq \infty$ places an upper limit on $\Delta((k_i+1)Q)$. It follows from the assumption that the response function is monotonically increasing (i.e., $\Delta((k_i+1)Q) > \Delta(k_iQ)$) that the right-hand side of (15) has a value less than ρ^{ϕ_i}, thus assuring that the solution for ϕ_{i+1} satisfies $\phi_{i+1} > \phi_i$.

Note that the solutions obtained for the feedback points will not in general be integral. The approximation obtained by taking the nearest integer values for the actual feedback points does not seem to be unwarranted in view of the other simplifying assumptions made in this analysis.

Once the feedback points have been determined, the actual value of $W(Q)$ can be calculated by setting $j = 0$ in (7) and a modified version of (12), yielding

$$\begin{aligned} \lambda_0 W(Q) = \lambda_0 \Bigg[& \sum_{m=2}^{M} \frac{\rho^{\phi_{m-1}}}{R_{m-1}} \left(\frac{\rho - \rho^{s_m+1}}{1-\rho}\right) + \bar{Q}\left(\frac{\rho - \rho^{\phi_1}}{1-\rho}\right)\Bigg] \\ & + \sum_{n=\phi_M+1}^{\infty} (n - \phi_M) p(n). \end{aligned} \tag{16}$$

The last term on the right-hand side of (16) accounts for the jobs in positions greater than ϕ_M, all of which have an attained service of 0 seconds. Simplifying (16) we obtain

$$W(Q) = \sum_{m=2}^{M} \frac{\rho^{\phi_{m-1}}}{R_{m-1}} \left(\frac{\rho - \rho^{s_m+1}}{1-\rho}\right) + \bar{Q}\left(\frac{\rho - \rho^{\phi_1}}{1-\rho}\right) + \frac{\rho^{\phi_M}}{\mu(1-\rho)}. \tag{17}$$

4. Adjustment of the initial waiting time. It has been shown in the previous section that, starting with the slopes of the line segments comprising a waiting time function, we can obtain a corresponding set of feedback points and from this, using (17), calculate the initial waiting time, $W(Q)$. Should this value prove unsatisfactory, it becomes necessary to adjust the feedback points, modifying both $W(Q)$ and the slopes. For example, should $W(Q)$ prove too large, it can be reduced by increasing the value of one or more of the ϕ_i. This can be done in a number of ways. A particularly simple adjustment is to increase the value of all feedback points by the same constant. Consider the effect of increasing each ϕ_i by c. Let $W(Q)$ be the initial waiting time using $\phi_1, \phi_2, \cdots, \phi_M$ and $W'(Q)$ be the initial waiting time using $\phi_1+c, \phi_2+c, \cdots, \phi_M+c$. Then it follows from (17) that

$$W'(Q)=\sum_{m=2}^{M}\frac{\rho^{\phi_{m-1}+c}}{R_{m-1}}\left(\frac{\rho-\rho^{s_m+1}}{1-\rho}\right)+\bar{Q}\left(\frac{\rho-\rho^{\phi_1+c}}{1-\rho}\right)+\frac{\rho^{\phi_M+c}}{\mu(1-\rho)}. \tag{18}$$

Substituting for $W(Q)$ on the right-hand side and simplifying, we get

$$W'(Q)=\rho^c W(Q)+\frac{\bar{Q}}{1-\rho}(\rho-\rho^{c+1}). \tag{19}$$

Note that for an RR discipline,

$$W_{\mathrm{RR}}(Q)=\bar{Q}\frac{\rho}{1-\rho} \tag{20}$$

and that this is the minimum value of the initial waiting time achievable within the restricted FBSQ model studied here, since once a new arrival is placed at the tail of the queue, no insertions are made in front of it. It thus follows from (20) that $W(Q)\geqq\bar{Q}\rho/(1-\rho)$ and that therefore, using (19), $W'(Q)<W(Q)$ as predicted.

Similarly, we can calculate the effect that such a transformation will have on the slopes of the line segments in the waiting time function. Letting $\Delta'(jQ)/Q$ be the slope when $\phi_1+c, \phi_2+c, \cdots, \phi_M+c$ are the feedback points, we get from (15)

$$\frac{\Delta'(jQ)}{Q}=\rho^c\frac{\Delta(jQ)}{Q}+\frac{\bar{Q}}{Q}\left(\frac{\rho-\rho^{c+1}}{1-\rho}\right). \tag{21}$$

Using (19) and (21) it is possible to locally tune the response function if the result obtained from the synthesis procedure of the previous section is not totally satisfactory.

5. Conclusions. A synthesis procedure has been described which can be used to achieve a particular response function using a feedback queuing algorithm. The system is approximated using a simplified model which assumes a single server with an exponential distribution of service times and a Poisson arrival process.

Acknowledgment. The author would like to thank Dr. Y. S. Chua for several helpful discussions.

REFERENCES

[1] L. KLEINROCK, *Time-shared systems: A theoretical treatment*, J. Assoc. Comput. Mach., 14 (1967), pp. 242–261.

[2] E. COFFMAN AND L. KLEINROCK, *Feedback queuing models for time-shared systems*, Ibid., 15 (1968), pp. 549–576.

[3] J. MCKINNEY, *A survey of analytical time-sharing models*, Comput. Surveys, 1 (1969), pp. 105–116.

[4] Y. CHUA AND A. BERNSTEIN, *Analysis of a feedback scheduler*, this Journal, 3 (1974), pp. 159–176.

[5] J. SHEMER, *Some mathematical considerations of time-sharing scheduling algorithms*, J. Assoc. Comput. Mach., 14 (1967), pp. 262–272.

[6] L. KLEINROCK, *A continuum of time-sharing scheduling algorithms*, Proc. AFIPS 1970 SJCC, vol. 36, AFIPS Press, Montvale, N.J., pp. 453–458.

[7] J. MICHEL AND E. COFFMAN, *Synthesis of a feedback queuing discipline for computer operation*, J. Assoc. Comput. Mach., 21 (1974), pp. 329–339.

[8] F. CORBATO, M. DAGGETT AND R. DALEY, *An experimental time-sharing system*, Proc. AFIPS 1962 SJCC, vol. 21, Spartan Books, New York, pp. 335–344.

[9] A. BERNSTEIN AND J. SHARP, *A policy driven scheduler for a time-sharing system*, Comm. ACM, 14 (1971), pp. 74–78.

[10] T. SAATY, *Elements of queuing theory*, McGraw-Hill, New York, 1961.

[11] J. LITTLE, *A proof of the queuing formula $L = \lambda W$*, Operations Res., 9 (1961), pp. 383–387.

[12] L. KLEINROCK AND E. COFFMAN, *Distribution of attained service in time-shared systems*, J. Comput. System Sci., 1 (1967), pp. 287–298.

[13] L. KLEINROCK, *Analysis of a time shared processor*, Naval Res. Logist. Quart., 11 (1964), pp. 59–73.

[14] N. PATEL, *A mathematical analysis of computer time-sharing systems*, Interim Tech. Rep. 20, Army Res. Office (Durham), Grant DA-ARO (D)-31-124-6158, Oper. Res. Center, Mass. Inst. of Tech., Cambridge, Mass., 1964.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

ON PRESERVING PROXIMITY IN EXTENDIBLE ARRAYS*

D. BOLLMAN†

Abstract. A. L. Rosenberg [4] has shown that fully extendible array storage schemes cannot preserve proximity of array positions in any global sense but only in a local sense. The purpose of this note is to consider two problems suggested in [4]. Our first result shows that, for any k array positions of an extendible array, there is a storage scheme which stores array elements that are "close to" the given k positions optimally "close" to each other. The "local diameter of preservation" of a storage scheme is essentially the cardinality of the smallest set of contiguous locations assigned to array positions in a "neighborhood" of a given array position. Our second result shows the nonexistence of "almost everywhere" polynomial bounds of a certain type for local diameters of preservation.

Key words. extendible array, array realization, local preservation of proximity

1. Introduction. In some recent papers, A. L. Rosenberg, [2], [3], [4], has investigated storage allocation schemes for arrays which allow for "easy" dynamic extendibility. Extendibility along given directions is modeled by viewing arrays schematically as multiple-dimensional spaces of tuples of positive integers which are infinite along the given directions, and the allocation scheme ("realization") as a one-to-one function from this space to the set of positive integers.

In [4], certain questions regarding the "preservation of proximity" in extendible arrays are studied. A metric is defined on array schemes, and the basic question dealt with is of the following type: what is the relation between proximity of array positions and proximity of their allocated positions? The purpose of this note is to resolve two problems suggested in that study. In the next section we introduce the necessary notation and definitions to formulate the problems precisely.

2. Definitions and background. Let N denote the set of positive integers. Call any set of the form $\{m, m+1, \cdots, m+k-1\}$, where $m, k \in N$, an *interval* of *length* k. For any $k \in N$, let N_k denote the interval $\{1, 2, \cdots, k\}$. For any $d \in N$, let N^d denote the Cartesian product $N \times N \times \cdots \times N$ taken d times. Denote $\langle 1, 1, \cdots, 1\rangle \in N^d$ by ε_d. For each $\pi \in N^d$ and each $i \in N_d$, let π_i denote the ith coordinate of π. Also define for each $\pi \in N^d$, $m(\pi) = \min\{\pi_i | i \in N_d\}$ and $M(\pi) = \max\{\pi_i | i \in N_d\}$ (with respect to the usual $\leqq$ order on N).

An (*extendible*) *realization* of N^d is any one-to-one function $\mathbf{r}$ from N^d to N. We follow the convention in [4] that every realization $\mathbf{r}$ is "normalized" so that $\mathbf{r}(\varepsilon_d) = 1$.

Let dist be a metric on N_d. For any $\pi \in N^d$ and any $n \in N \cup \{0\}$, define an *n-neighborhood* of π by $\mathbf{N}(\pi; n) = \{\rho \in N^d | \text{dist}(\pi, \rho) \leqq n\}$. Also let $\Delta(\pi; n) = \mathbf{N}(\pi; n) - \mathbf{N}(\pi; n-1)$ for each $n \in N$ and each $\pi \in N^d$. Let $\mathcal{N}(\pi; n)$ and $\mathcal{D}(\pi; n)$ denote the cardinalities of $\mathbf{N}(\pi; n)$ and $\Delta(\pi; n)$ respectively. In the rest of the paper, we assume a metric on N^d for which each $\mathcal{N}(\pi; n)$ is finite.

* Received by the editors January 3, 1975, and in revised form August 4, 1975.

† Department of Mathematics, University of Puerto Rico, Mayagüez, Puerto Rico 00708. This research was done while the author was visiting the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

For points $\pi \in N^d$ whose minimal distances from the coordinate axes is larger than n, the value of $\mathcal{N}(\pi; n)$ depends only on d and n. In such a case $\mathbf{N}(\pi; n)$ is a *radius n d-sphere*, and we shall denote the value of $\mathcal{N}(\pi; n)$ by $V(d, n)$. It is shown in [5] that for the "L_1-metric", defined by dist $(\rho, \sigma) = \sum_{i=1}^{d} |\rho_i - \sigma_i|$,

$$\sum_{k=0}^{d} \binom{d}{k}\binom{n}{k}2^k.$$

It is shown in [4] that realizations of N^d, $d > 1$, cannot "globally preserve n-neighborhoods". That is, for no realization $\mathbf{r}$ of N^d does there exist a function ("global diameter of preservation") $D_{\mathbf{r}}: N \to N$ with the property that for all $\pi \in N^d$ and all $\rho, \sigma \in \mathbf{N}(\pi; n)$, $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < D_{\mathbf{r}}(n)$. However, given a realization $\mathbf{r}$ of N^d, such a function can be defined at each $\pi \in N^d$. If $\mathbf{r}$ is a realization of N^d, the *local diameter of preservation* of $\mathbf{r}$, $D_{\mathbf{r}}: N \times N^d \to N$, is defined by $D_{\mathbf{r}}(n, \pi) = \max(\mathbf{r}(\mathbf{N}(\pi; n))) - \min(\mathbf{r}(\mathbf{N}(\pi; n))) + 1$, for each $n \in N$ and $\pi \in N^d$. Note that for any $n \in N$ and $\pi \in N^d$, $D_{\mathbf{r}}(n, \pi)$ is simply the length of the smallest interval of N containing the image of $\mathbf{N}(\pi; n)$ under $\mathbf{r}$.

An obvious lower bound for $D_{\mathbf{r}}$ is $\mathcal{N}$. ($D_{\mathbf{r}}(n, \pi) \geqq \mathcal{N}(\pi; n)$ for any realization $\mathbf{r}$ of N^d, $n \in N$, and $\pi \in N^d$, since any such $\mathbf{r}$ is one-to-one). This lower bound is attained at points π such that for each $n \in N$, $\mathbf{r}$ maps $\mathbf{N}(\pi, ; n)$ *onto* an interval of N. Such a situation is described by saying that $\mathbf{r}$ is "centered" at π. That is, a realization $\mathbf{r}$ of N^d is said to be *centered* at π, and π is called the *center* of $\mathbf{r}$, if for all $n \in N$, $\rho, \sigma \in \mathbf{N}(\pi; n)$ implies $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < \mathcal{N}(\pi; n)$.

Rosenberg [4] shows that, under the L_1-metric and for any $\pi \in N^d$, there is a realization of N^d which is centered at π, but that no realization can be centered at more than one point of N^d. However, in an earlier version of [4] (IBM Rep. RC-4874, 1974) he raises the question of whether given k points $\pi_1, \cdots, \pi_k$ of N^d, there exists a realization $\mathbf{r}$ of N^d which is "k-centered" at all of these points in the sense that, if $\rho, \sigma \in \bigcap_{i=1}^{k} N(\pi_i; n_i)$, then $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < k \cdot \min \mathcal{N}(\pi_i; n_i)$. Our first result answers this question in the affirmative.

Rosenberg also shows that for the L_1-metric on N^d, there is a constant $c > 0$ depending only on d, such that for all $n \in N$, $D_{\mathbf{r}}(n, \pi) > c \cdot n \cdot M(\pi)^{d-1}$ for infinitely many $\pi \in N^d$. This raises the question of whether this lower bound can be strengthened to an "almost everywhere" bound. Our second result shows that the answer is no if "almost everywhere" means for all but finitely many n (i.e., for all $n > A$ for some $A \in N$), and for all but finitely many $\pi \in N^d$ (say, for all $\pi \in N^d$ with $M(\pi) > B$ for some $B \in N$).

3. Quasi-centers. This section is devoted to proving the following.

THEOREM 1. *Let $X \in \{\pi_1, \pi_2, \cdots, \pi_k\}$ be an arbitrary finite subset of N^d. There exists a realization $\mathbf{r}$ of N^d such that, for any family $\{\mathbf{N}(\pi_i; n_i)\}_{i=1}^{k}$, if $\rho, \sigma \in \bigcap_{i=1}^{k} \mathbf{N}(\pi_i; n_i)$, then $|\mathbf{r}(\rho) - \mathbf{r}(\sigma)| < k \cdot \min \mathcal{N}(\pi_i; n_i)$.*

Proof. If $\varepsilon_d \notin X$, define $\mathbf{r}(\varepsilon_d) = 1$ and $\mathbf{r}(\pi_i) = i + 1$ for each $i \in N_k$. If $\varepsilon_d \in X$, say $\varepsilon_d = \pi_1$, then define $\mathbf{r}(\pi_i) = i$ for each $i \in N_k$.

The idea in assigning values to points $\rho \in N^d - X$ is as follows: let $\rho \in N^d$ be arbitrary, and let $\pi_i \in X$ be the "quasi-center" which is "closest" to ρ in the sense that $\mathcal{N}(\pi_i; \text{dist}(\pi_i, \rho))$ is minimal. (If there is more than one such quasi-center,

take the one with the smallest subscript). The point ρ is then assigned a value of the form $r(\pi_i)+lk$, where $l<\mathcal{N}(\pi_i;\operatorname{dist}(\pi_i;\rho))$.

More specifically, for each $n\in N$ and each $i\in N$, define images of points in $\Delta(\pi_i;n)$ recursively as follows: first, tag all members $\rho\neq\varepsilon_d$ of $\Delta(\pi_i;n)$ having the property that $\forall j\in N_k[\mathcal{N}(\pi_i;n)<\mathcal{N}(\pi_j;\operatorname{dist}(\pi_j;\rho))\vee[\mathcal{N}(\pi_j;n)=\mathcal{N}(\pi_j;\operatorname{dist}(\pi_j;\rho))\wedge j\geqq i]]$. Then assign, in any order, the values $m+k$, $m+2k$, $m+3k$, $\cdots$ to the tagged members of $\Delta(\pi_i;n)$, where m is the largest number assigned to a tagged member of $\Delta(\pi_i;n-1)$.

This assignment defines a realization $\mathbf{r}$ of N_d, since every $\rho\in N^d$ will be tagged (and hence assigned a value) exactly once. Furthermore, every value is of the form $i+lk$, where $1\leqq i\leqq k$ (or $2\leqq i\leqq k+1$ if $\varepsilon_d\notin X$) and $l\in N$. Hence, if $\mathbf{r}(\rho)=i_1+l_1k=i_2+l_2k=\mathbf{r}(\sigma)$, then $i_1=i_2$ and $l_1=l_2$, and consequently $\rho=\sigma$. Thus $\mathbf{r}$ is a realization of N^d.

Now suppose that $\rho_1,\rho_2\in\bigcap_{i=1}^{k}\mathbf{N}(\pi_i;n_i)$. For each $t\in\{1,2\}$, let i_t be the index of the quasi-center "closest" to ρ_t, i.e., let $i_t=\min\{i\in N|\forall j\in N_k[\mathcal{N}(\pi_i;\operatorname{dist}(\pi_i,\sigma_t))\leqq\mathcal{N}(\pi_j;\operatorname{dist}(\pi_j,\sigma_t))]\}$. Then for each $t\in\{1,2\}$, $\mathbf{r}(\sigma_t)=\mathbf{r}(\pi_{i_t})+l_tk$ for some l_t with $1\leqq l_t<\mathcal{N}(\pi_{i_t};\operatorname{dist}(\pi_{i_t},\sigma_t))$. By definition of i_t, $\mathcal{N}(\pi_{i_t};\operatorname{dist}(\pi_{i_t},\sigma_t))\leqq\mathcal{N}(\pi_i;n_i)$ for each $i\in N_k$. Hence, for each $i\in N$,

$$\begin{aligned}|\mathbf{r}(\sigma_1)-\mathbf{r}(\sigma_2)|&\leqq|\mathbf{r}(\sigma_{i_1})-\mathbf{r}(\sigma_{i_2})|+|l_1-l_2|k\\&<k+k\cdot(\mathcal{N}(\pi_i;n_i)-1)=k\cdot\mathcal{N}(\pi_i;n_i),\end{aligned}$$

which completes the proof of the theorem.

Taking $k=1$, we have the result that for any $\pi\in N^d$, there exists a realization $\mathbf{r}$ of N^d which is centered at π (a result proved in [4]).

4. An attempted lower bound for $D_{\mathbf{r}}$. Rosenberg [4] defines the *cumulative local diameter of* preservation by $D_{\mathbf{r}}^{*}(n,\pi)=\max\{D_{\mathbf{r}}(n,\xi)|M(\xi)\leqq M(\pi)\}$. He then shows that there is a constant $c>0$, depending only on d, such that, for all realizations $\mathbf{r}$ of N^d, and for all $n\in N$ and $\pi\in N^d$, $D_{\mathbf{r}}^{*}(n,\pi)>c\cdot n\cdot M(\pi)^{d-1}$. As a corollary, it follows that there is a constant $c>0$, depending only on d, such that, for all realizations $\mathbf{r}$ of N^d, and all $n\in N$, $D_{\mathbf{r}}(n,\pi)>c\cdot n\cdot M(\pi)^{d-1}$ for infinitely many $\pi\in N^d$. This raises the question of whether this bound can be strengthened to hold for "almost all" $\pi\in N^d$. It will follow from the theorem of this section that the answer is no. In fact, there is *no* positive-valued polynomial which serves as an almost everywhere lower bound for $D_{\mathbf{r}}(n,\pi)$.

In the proof of the theorem, we assume the L_1-metric, but as we point out following the proof, the result holds for more general metrics as well. The term "strictly increasing" in the statement of the theorem is intended to mean under the strict partial order $<_d$ on N^d defined by: $\pi<_d\rho$ if and only if $\pi_i<\rho_i$ for each $i\in N_d$.

THEOREM 2. *For any $d\in N-\{1\}$, there exists a realization $\mathbf{r}$ and a strictly increasing sequence $\langle\pi_i\rangle_{i=1}^{\infty}$ of points in N^d such that, for all $n\in N$ and $m\leqq n$, $D_{\mathbf{r}}(m,\pi_n)=V(d,m)$.*

Proof. The idea of the proof is to define a sequence $\langle\pi_i\rangle_{i=1}^{\infty}$ of points on the diagonal of N^d, all sufficiently far from the origin to ensure that for each $n\in N$, $\mathbf{N}(\pi_n;n)$ is a d-sphere and such that, for $m\neq n$, $\mathbf{N}(\pi_m;n)\cap\mathbf{N}(\pi_n;n)=\varnothing$. Values are assigned to points of each $\mathbf{N}(\pi_m;n)$ in a one-to-one fashion so that

each $\mathbf{r}(\mathbf{N}(\pi_n; n))$, as well as each $\mathbf{r}(\mathbf{N}(\pi_n; m))$, $m \leqq n$, constitutes an interval of N. These values are chosen in such a way that the complement of $\mathbf{r}(\cup\mathbf{N}(\pi_n, n))$ is infinite, so that $\mathbf{r}$ can be extended in a one-to-one fashion to all of N^d.

Given $d \in N$, $d \geqq 2$, define a sequence $\langle a_i \rangle_{i=1}^{\infty}$ in N as follows:

$$a_1 = V(d, 1),$$
$$a_{i+1} = a_i + V(d, i) + 1 \quad \text{for each } i \in N.$$

Define $\pi_i = a_i \varepsilon_d$. Note that $\langle a_i \rangle_{i=1}^{\infty}$ is a strictly increasing sequence of points in N, and hence $\langle \pi_i \rangle_{i=1}^{\infty}$ is a strictly increasing sequence of points in N^d.

Now for each $n \in N$, assign values to points $\rho \in \mathbf{N}(\pi_n; n)$, $\rho \neq \varepsilon_d$, as follows: assign π_n the value a_n. Assign points of $\mathbf{N}(\pi_n; 1)$ the values $a_n + 1, a_n + 2, \cdots, a_n + \mathscr{D}(\pi_n; 1)$, in any order. Assume inductively that values have been assigned to points in $\mathbf{N}(\pi_n; m)$ for $1 \leqq m < n$. If $b = \max(\mathbf{r}(\mathbf{N}(\pi_n; m))$, then assign, in any order, the values $b+1, b+2, \cdots, \mathscr{D}(\pi_n; m+1)$, to points in $\Delta(\pi_n; m+1)$.

Note that for each $n \in N$ and each $m \leqq n$, points in $\mathbf{N}(\pi_n; m)$ are assigned the values $a_n, a_n + 1, \cdots, a_n + \mathscr{N}(\pi_n, m) - 1$. But for any n, $m(\pi_n) = a_n > n$, and so $\mathscr{N}(\pi_n; m) = V(d, m)$ for any $n \in N$ and any $m \leqq n$. Thus, for any $n \in N$ and any $m \leqq n$, the points of $\mathbf{N}(\pi_n; m)$ are assigned the values $a_n, a_n + 1, \cdots, a_n + V(d, m) - 1$.

To complete the proof, we need only show that there is indeed a realization $\mathbf{r}$ of N^d which assigns values to points $\rho \in \mathbf{N}(\pi_n; n)$ according to the above rule. That is, we must show that via the above assignment,

(i) no point of $\cup\mathbf{N}(\pi_n; n)$ is assigned more than one value;

(ii) there are infinitely many $n \in N$ which are not values of points in $\cup\mathbf{N}(\pi_n; n)$.

To prove (i), note that $\mathbf{N}(\pi_m; m) \cap \mathbf{N}(\pi_n; n) = \varnothing$ for $m \neq n$. For suppose that $\rho \in \mathbf{N}(\pi_m; m) \cap \mathbf{N}(\pi_n; n)$ where $n = m + k$ for some $k \in N$. Then for $d \geqq 2$,

$$\begin{aligned} m + n &\geqq \operatorname{dist}(\pi_m, \rho) + \operatorname{dist}(\pi_n, \rho) \geqq \operatorname{dist}(\pi_m, \pi_n) = d \cdot (a_n - a_m) \\ &= d \cdot [V(d, m) + \cdots + V(d, m + k - 1) + k] \\ &\geqq d \cdot [k \cdot V(d, m) + k] \geqq 2V(d, m) + k > 2m + k = m + n, \end{aligned}$$

a contradiction.

To prove (ii), note that for any $n \in N$,

$$\begin{aligned} \max \mathbf{r}(\mathbf{N}(\pi_n; n)) = a_n + V(d, n) - 1 &< a_n + V(d, n) + 1 = a_{n+1} \\ &= \min \mathbf{r}(\mathbf{N}(\pi_{n+1}; n + 1)). \end{aligned}$$

Hence, for any sequence $\langle \rho_n \rangle_{n=1}^{\infty}$ in N^d with $\rho_n \in \mathbf{N}(\pi_n; n)$, we have

$$\begin{aligned} \mathbf{r}(\rho_1) < a_2 + V(d, 2) < \mathbf{r}(\rho_2) < a_3 + V(d, 3) < \cdots < \mathbf{r}(\rho_n) \\ < a_{n+1} + V(d, n + 1) < \mathbf{r}(\rho_{n+1}) < \cdots. \end{aligned}$$

Thus, $\langle a_n + V(d, n) \rangle_{n+1}^{\infty}$ is a strictly increasing sequence of positive integers none of which is a value of any point in $\cup\mathbf{N}(\pi_n; n)$.

The reader can readily verify that the proof of the theorem, with minor changes, carries through for any metric dist on N^d for which dist $(\pi, \rho) \geqq \max\{|\pi_i - \rho_i|\}$ and for which radius n d-spheres contain more than n points. (In this general case, to guarantee (i), we could define $a_{i+1} = a_i + 2V(d, i) + 1$). Note that in addition to the L_1-metric, the "Euclidean (or L_2-) metric" defined by dist $(\pi, \rho) = (\sum_{i=1}^{d} (\pi_i - \rho_i)^2)^{1/2}$, and the "maximum (or L_∞-) metric" defined by dist $(\pi, \rho) = \max\{|\pi_i - \rho_i|\}$, also have these properties.

COROLLARY. *There is no polynomial, $p(x, y) > 0$, depending on both x and y such that for all realizations r of N^d, $d \geqq 2$, $D_{\mathbf{r}}(n, \pi) > p(n, M(\pi))$ for all but finitely many $n \in N$ and all but finitely many $\pi \in N^d$.*

Proof. Let $\mathbf{r}$ be a realization of N^d and let $\langle \pi_i \rangle_{i=1}^{\infty}$ be a sequence in N^d having the properties described in Theorem 2. Suppose that there is a polynomial $p(x, y) > 0$ such that for some $A, B \in N$,

$$D_{\mathbf{r}}(n, \pi) > p(n, M(\pi))$$

for all $n > A$ and all π with $M(\pi) > B$. Then

$$D_{\mathbf{r}}(A+1, \pi)) > p(A+1, M(\pi))$$

for all $\pi \in N^d$ with $M(\pi) > B$. Since $\langle \pi_i \rangle_{i=1}^{\infty}$ is a strictly increasing sequence, there exists $C \in N$ such that

$$D_{\mathbf{r}}(A+1, \pi_n) > p(A+1, M(\pi_n)) \quad \text{for all } n > C.$$

By definition of $\langle \pi_i \rangle_{i=1}^{\infty}$,

$$D_{\mathbf{r}}(A+1, \pi_n) = V(d, A+1) \quad \text{for } n > A+1.$$

Thus $\langle p(A+1, M(\pi_n)) \rangle_{n=1}^{\infty}$ is bounded. But this implies that $p(A+1, M(\pi_n))$ does not depend on n, which is a contradiction.

The question remains as to whether or not there exists a polynomial $p(x, y) > 0$ which serves as a lower bound for $D_{\mathbf{r}}(n, \pi)$ under some other reasonable interpretation of "almost everywhere". We conclude by commenting on two remaining possibilities.

First, given $d \in N$, does there exist a polynomial $p(x, y) > 0$ and a function $f: N \to N$ such that, for any realization $\mathbf{r}$ of N^d and $n \in N$, $D_{\mathbf{r}}(n, p) > p(n, M(\pi))$ for all $\pi \in N^d$ with $M(\pi) > f(n)$? The answer is no, since, as indicated in the proof of the corollary, for any $d \in N$, $d \geqq 2$, there is a strictly increasing sequence $\langle \pi_i \rangle_{i=1}^{\infty}$ in N^d and a realization $\mathbf{r}$ of N^d such that, for any fixed $m \in N$, the value of $D_{\mathbf{r}}(m, \pi_n)$ eventually becomes constant (i.e., for $n > A$ for some $A \in N$). However, for any polynomial $p(x, y) > 0$, $p(m, M(\pi_n))$ grows with increasing n.

Lastly, given $d \in N$, does there exist a polynomial $p(x, y) > 0$ and a function $f: N^d \to N$ such that, for any realization $\mathbf{r}$ of N^d and any $\pi \in N^d$, $D_{\mathbf{r}}(n, \pi) > p(n, M(\pi))$ for all $n > f(\pi)$? The answer in this case is yes. It is shown in [4] that for the L_1-metric, $\mathcal{N}(\pi, n) \geqq \binom{n+d}{d}$. Hence for any $d \in N$, any realization $\mathbf{r}$ of N^d, and any $\pi \in N^d$, $D_{\mathbf{r}}(n, \pi) \geqq \mathcal{N}(\pi; n) > n^d > n \cdot M(\pi)^{d-1}$ for all $n > M(\pi)$.

Acknowledgment. I am grateful to Dr. A. L. Rosenberg for his suggestions and encouragement.

REFERENCES

[1] D. BOLLMAN, *Some tailor-made extendible array realizations*, Rep. RC-5L21, IBM Watson Res. Center, Yorktown Heights, N.Y., 1974.

[2] A. L. ROSENBERG, *Allocating storage for extendible arrays*, J. Assoc. Comput. Mach., 21 (1974), pp. 652–670.

[3] ———, *Managing storage for extendible arrays.* Reps. RC-4433, RC-4450, IBM Watson Res. Center, Yorktown Heights, N.Y., 1973; this Journal 4 (1975), pp. 287–306.

[4] ———, *Preserving proximity in arrays*, Rep. RC-4874, IBM Watson Res. Center, Yorktown Heights, N.Y., 1974; this Journal, 4 (1975), pp. 443–460.

[5] C. K. WONG AND T. W. MADDOCKS, *A generalized Pascal's triangle*, Fibonacci Quart., 13 (1975), pp. 134–136.

SIAM J. COMPUT.
Vol. 5, No. 2, June 1976

COMPUTATIONAL COMPLEXITY OVER FINITE FIELDS*

VOLKER STRASSEN†

Abstract. We give nonlinear lower bounds for the complexity of evaluation of a polynomial function at many points and of interpolation in the case of a finite groundfield.

Key words. computational complexity, symbolic manipulation, interpolation, modular method, finite field

1. Introduction. If S is a set, $\#S$ means its cardinality, log always means $\log_2$.

Let k be a finite field, d a natural number, D a subset of k^d and $f_1, \cdots, f_r$ functions from D to k. We are concerned with the minimum number of multiplications and divisions sufficient to evaluate the functions $f_1, \cdots, f_r$ at some point, when an arbitrary number of additions, subtractions and multiplications by fixed elements of k are allowed for free (we restrict ourselves to straight line algorithms, i.e., to algorithms which do not contain any branching instructions; see, however, Remark 2 at the end of the paper). This minimum number is called the complexity of the set $\{f_1, \cdots, f_r\}$ and is denoted by $L(f_1, \cdots, f_r)$. Our aim is to find lower bounds for the complexity of interesting sets of functions.

To this end, we try to employ the following result of [4, Satz 2.3] (see also [1]): let K be an algebraically closed field and let $F_1, \cdots, F_r$ be rational functions from K^d to K (so each F_i has its individual domain of definition, which is a dense Zariski-open subset of K^d). Define $L(F_1, \cdots, F_r)$ in a similar manner as $L(f_1, \cdots, f_r)$ above. Let γ be the degree (in the sense of algebraic geometry) of the graph of the rational map from K^d to K^r associated with the functions $F_1, \cdots, F_r$. Then

$$L(F_1, \cdots, F_r) \geqq \log \gamma.$$

How can this be used for our present purpose? We take for K an algebraic closure of k. It is an immediate consequence of the notion of a straight line algorithm or computation (see [2]), that any such algorithm for computing $f_1, \cdots, f_r$ over the finite field k can be set to work over the algebraic closure K as well and serves there to compute rational functions $F_1, \cdots, F_r$ from K^d to K, which are everywhere defined on D and whose restrictions to D coincide with $f_1, \cdots, f_r$. Let S be the graph of the map from D to k^r defined by $f_1, \cdots, f_r$. Then S is a finite subset of k^{d+r} and therefore of K^{d+r}. What we need is a practical condition on S which ensures that any graph W of a rational map from K^d to K^r containing S has large degree. (Then any set of rational functions $F_1, \cdots, F_r$ extrapolating $f_1, \cdots, f_r$ will be hard to compute by the result quoted above, and therefore $f_1, \cdots, f_r$ will be hard to compute.)

Below we will give such a condition, which in fact works in a slightly more general situation: S may be any finite subset and W an arbitrary irreducible subvariety of dimension d of some K^N.

* Received by the editors December 27, 1974, and in revised form July 12, 1975.

† Seminar für Angewandte Mathematik der Universität Zürich, 8032 Zürich, Switzerland.

As a concrete application, let $d = 2n+2$ and $D = k^{2n+2}$, and define functions $f_0, \cdots, f_n$ from k^{2n+2} to k by

$$f_0(\alpha_0, \cdots, \alpha_n, \xi_0, \cdots, \xi_n) := \alpha_0\xi_0^n + \cdots + \alpha_n,$$
$$\cdots\cdots$$
$$f_n(\alpha_0, \cdots, \alpha_n, \xi_0, \cdots, \xi_n) := \alpha_0\xi_n^n + \cdots + \alpha_n$$

for any $(\alpha_0, \cdots, \alpha_n, \xi_0, \cdots, \xi_n) \in k^{2n+2}$. Then

$$cn \log m < L(f_0, \cdots, f_n) < c'n \log m,$$

where $m = \min\{n+1, \# k\}$ and c, c' are universal constants. Thus the order of magnitude of the complexity of evaluating a polynomial of degree n at $n+1$ points over a finite field is $n \log m$. A similar result holds for interpolation. (Here, of course, we have to assume that the field k has at least $n+1$ elements, since that many different base points are needed.)

2. The method. Let K be an algebraically closed field, t and N natural numbers. We will call a finite subset S of K^N a *t-set* iff for any d with $0 \leqq d \leqq N$ and any d-dimensional closed irreducible subvariety W of K^N, we have

$$\deg W \geqq \#(S \cap W)/t^d. \tag{2.1}$$

Obviously any set with at most t elements is a t-set and any subset of a t-set is again a t-set. On the other hand, taking $d = N$, one sees that a t-set can have at most t^N elements. The following lemma is crucial for the construction of large t-sets.

LEMMA 2.1. *Let $S \subset K^N$ and let $H_1, \cdots, H_m$ be hypersurfaces in K^N with the following properties*:

1. $\sum_i \deg H_i \leqq t$.
2. $S \subset \bigcup H_i$.
3. *For every i, $S \cap H_i$ is a t-set.*

Then S is a t-set.

Proof. Let W be a closed irreducible subvariety of K^N of dimension d. Without loss of generality, $d > 0$. Assume first that $W \subset H_i$ for some i. Then since $S \cap H_i$ is a t-set, we have

$$\deg W \geqq \#(S \cap H_i \cap W)/t^d = \#(S \cap W)/t^d. \tag{2.2}$$

Next assume that W is not contained in any H_i. Since the H_i are hypersurfaces, W intersects each H_i properly. Therefore, by Bézout's theorem,

$$\deg(W) \cdot \deg(H_i) \geqq \sum_{\substack{C \text{ component} \\ \text{of } W \cap H_i}} \deg(C). \tag{2.3}$$

Each C has dimension $d-1$ and is contained in H_i, so (2.2) applies to C, and we obtain

$$\deg C \geqq \#(S \cap C)/t^{d-1}.$$

Together with (2.3), we have, therefore,

$$\deg(W) \cdot \deg(H_i) \geqq \sum_{\substack{C \text{ component} \\ \text{of } W \cap H_i}} \#(S \cap C)/t^{d-1}.$$

Summing over i, we get

$$t \cdot \deg(W) \geqq \sum_i \deg(W) \cdot \deg(H_i)$$

$$\geqq \sum_i \sum_{\substack{C \text{ component} \\ \text{of } W \cap H_i}} \#(S \cap C)/t^{d-1} \geqq \#(S \cap W)/t^{d-1}.$$

(The first inequality follows from condition 1, the last inequality from condition 2 of the lemma.) This completes the proof.

In this paper we will use Lemma 2.1 only by means of the following

COROLLARY 2.2. *Let $S \subset K^N$, let t be a natural number and let $l_1, \cdots, l_q$ be linear forms on K^N with the following properties*:

1. *For any i with $1 \leqq i \leqq q$ and for any $c_1, \cdots, c_{i-1} \in K$, the linear form l_i restricted to the set $S \cap \{x : l_1(x) = c_1, \cdots, l_{i-1}(x) = c_{i-1}\}$ assumes at most t different values.*
2. *$l_1, \cdots, l_q$ separate the points of S, i.e., for any $c_1, \cdots, c_q \in K$ there is at most one $x \in S$ such that $l_1(x) = c_1, \cdots, l_q(x) = c_q$.*

Then S is a t-set.

Proof. The proof is by induction on q. In case $q = 1$, the assumptions of Corollary 2.2 imply that S has at most t points. In case $q > 1$, let $l_1, \cdots, l_q$ be given and let Corollary 2.2 be true for $q-1$ linear forms. By condition 1, the form l_1 assumes on S at most t values, say $b_1, \cdots, b_m$. We apply Lemma 2.1 with

$$H_i := \{x \in K^N : l_1(x) = b_i\}.$$

Conditions 1 and 2 of Lemma 2.1 obviously hold, and condition 3 follows from the induction hypothesis.

Corollary 2.2 may be viewed as a generalization of the following well-known fact: if $T \subset K$ has t elements, then any nonzero polynomial in $K[x_1, \cdots, x_N]$ vanishing on all points of T^N has degree $\geqq t$.

Now let k be a finite field, let $D \subset k^d$ be nonempty and A be the set of functions from D to k. In A we specify the following operations:

0-*ary operations*: the constant functions 0 and 1, for any $i \leqq d$ the projection function

$$(\xi_1, \cdots, \xi_d) \to \xi_i$$

restricted to D.

Unary operations: for each $\beta \in k$, the operation "multiplication by β".

Binary operations: pointwise addition, subtraction and multiplication of functions, as well as division by functions which do not vanish anywhere on D (these are just the units of the ring A).

For the definition of computational complexity in A, we also need a cost function: we will just count binary multiplications and divisions, all other operations being free (thus in the terminology of [2], $z = 1_{\{*,/\}}$). The complexity $L(f_1, \cdots, f_r)$ of a finite set of functions $\{f_1, \cdots, f_r\} \subset A$ is then defined as in [2].

THEOREM 2.3. *Let $f_1, \cdots, f_r \in A$ and let $S \subset k^{d+r}$ be the graph of the map*

$$(f_1, \cdots, f_r) : D \to k^r.$$

Suppose that there are k-linear forms $l_1, \cdots, l_q$ on k^{d+r} with the following properties:

1. *For any i with $1 \leqq i \leqq q$ and for any $c_1, \cdots, c_{i-1} \in k$, the form l_i assumes at most t different values on the set*

$$S \cap \{v \in k^{d+r} : l_1(v) = c_1, \cdots, l_{i-1}(v) = c_{i-1}\}.$$

2. *$l_1, \cdots, l_q$ separate the points of S.*

Then

$$L(f_1, \cdots, f_r) \geqq \log(\# D/t^d).$$

Proof. Let K be an algebraic closure of k and let B be the field of rational functions from K^d to K (whose elements we view concretely as partial functions). In B we allow the same operations as in A (so B is a K-field over the projection functions [2]) and use the same cost function. If β is a computation which computes $f_1, \cdots, f_r$ in A, the same β computes in B rational functions $F_1, \cdots, F_r$ such that $D \subset \operatorname{Def} F_i$ and $F_i \restriction D = f_i$ for all i (use induction along the steps of β). Thus the graph W of the rational map associated with $F_1, \cdots, F_r$ contains the set S. But the conditions on S of Theorem 2.3 imply the conditions of Corollary 2.2. Therefore S is a t-set. Moreover, the closure of W is a d-dimensional closed irreducible subvariety of K^{d+r}. The definition of a t-set now gives

$$\deg(W) \geqq \#(S \cap W)/t^d = \#S/t^d = \#D/t^d.$$

From [4, Satz 2.3], we conclude

$$L(\beta) \geqq L(F_1, \cdots, F_r) \geqq \log(\#D/t^d).$$

Since β is an arbitrary computation, which computes $f_1, \cdots, f_r$, this yields

$$L(f_1, \cdots, f_r) \geqq \log(\#D/t^d).$$

3. Applications. We keep the notation of the previous section. In particular, k is a fixed finite field.

PROPOSITION 3.1. *Let $n \geqq 2$ be an integer and let A be the ring of all functions from k^{2n+2} to k. Define $f_0, \cdots, f_n \in A$ by*

$$f_0(\boldsymbol{\alpha}, \boldsymbol{\xi}) := \alpha_0 \xi_0^n + \cdots + \alpha_n,$$
$$\cdots\cdots$$
$$f_n(\boldsymbol{\alpha}, \boldsymbol{\xi}) := \alpha_0 \xi_n^n + \cdots + \alpha_n$$

for any $(\boldsymbol{\alpha}, \boldsymbol{\xi}) := (\alpha_0, \cdots, \alpha_n, \xi_0, \cdots, \xi_n) \in k^{2n+2}$. Then

$$L(f_0, \cdots, f_n) \geqq \frac{n+1}{2} \log m - n,$$

where $m = \min\{n+1, \#k\}$.

Proof. We choose pairwise different $\lambda_1, \cdots, \lambda_m \in k$, and put

$$t := \lceil \sqrt{m} \rceil.$$

Next we choose $\alpha_0, \cdots, \alpha_n \in k$ such that on the set $\{\lambda_1, \cdots, \lambda_m\}$, the polynomial

function

$$\xi \mapsto \alpha_0 \xi^n + \cdots + \alpha_n$$

assumes only the values $\lambda_1, \cdots, \lambda_t$ and each value at most t times (since $m \leqq n+1$ this can be done by interpolation). Now let

$$\tilde{D} := \{\lambda_1, \cdots, \lambda_m\}^{n+1} \subset k^{n+1},$$

and define functions $\tilde{f}_i$ from $\tilde{D}$ to k as follows:

$$\tilde{f}_i(\xi_0, \cdots, \xi_n) := f_i(\alpha_0, \cdots, \alpha_n, \xi_0, \cdots, \xi_n)$$

for any $(\xi_0, \cdots, \xi_n) \in \tilde{D}$. Since there is a homomorphism from A to the ring $\tilde{A}$ of functions from $\tilde{D}$ to k which carries f_i into $\tilde{f}_i$, we have

$$L_A(f_0, \cdots, f_n) \geqq L_{\tilde{A}}(\tilde{f}_0, \cdots, \tilde{f}_n). \tag{3.1}$$

We will apply Theorem 2.3 to $\tilde{f}_0, \cdots, \tilde{f}_n$. Let $\tilde{S}$ be the graph of the map

$$(\tilde{f}_0, \cdots, \tilde{f}_n) : \tilde{D} \to k^{n+1},$$

i.e., $\tilde{S}$ is the set of all $(\xi_0, \cdots, \xi_n, \eta_0, \cdots, \eta_n) \in k^{2n+2}$ which satisfy $(\xi_0, \cdots, \xi_n) \in \tilde{D}$ and

$$\begin{aligned} \eta_0 &= \alpha_0 \xi_0^n + \cdots + \alpha_n, \\ &\cdots \\ \eta_n &= \alpha_0 \xi_n^n + \cdots + \alpha_n. \end{aligned}$$

Let $x_0, \cdots, x_n, y_0, \cdots, y_n$ be the coordinate projections of k^{2n+2} (i.e., $x_i(\boldsymbol{\xi}, \boldsymbol{\eta}) = \xi_i$, $y_i(\boldsymbol{\xi}, \boldsymbol{\eta}) = \eta_i$), and put

$$l_1 = y_0, \cdots, l_{n+1} = y_n, \qquad l_{n+2} = x_0, \cdots, l_{2n+2} = x_n.$$

Then condition 2 of Theorem 2.3 is trivially satisfied since $l_1, \cdots, l_{2n+1}$ even separate the points of k^{2n+2}. As to condition 1, the projection y_i assumes on all of $\tilde{S}$ at most the values $\lambda_1, \cdots, \lambda_t$, since by definition of $\tilde{S}$ any value of y_i on $\tilde{S}$ is of the form $\alpha_0 \xi^n + \cdots + \alpha_n$ with $\xi \in \{\lambda_1, \cdots, \lambda_m\}$, and therefore the defining property of $\alpha_0, \cdots, \alpha_n$ applies. Similarly, the projection x_i assumes on $\tilde{S} \cap \{y_i = c\}$ at most t values, since by the definition of $\tilde{S}$ any such value ξ satisfies

$$\xi \in \{\lambda_1, \cdots, \lambda_m\}, \qquad c = \alpha_0 \xi^n + \cdots + \alpha_n,$$

and therefore the defining property of $\alpha_0, \cdots, \alpha_n$ applies again.

So both conditions of Theorem 2.3 hold. As a consequence, we have

$$\begin{aligned} L_{\tilde{A}}(\tilde{f}_0, \cdots, \tilde{f}_n) &\geqq \log(\#\tilde{D}/t^{n+1}) \\ &\geqq (n+1)\log\left(m/\lceil\sqrt{m}\rceil\right) \geqq (n+1)\log\left(\tfrac{2}{3}\sqrt{m}\right) \\ &\geqq \frac{n+1}{2}\log m - n. \end{aligned}$$

Together with (3.1), this yields the conclusion of Proposition 3.1.

COROLLARY 3.2. *With the notation of Proposition* 3.1, *we have*

$$(n/4)\log m \leqq L(f_0, \cdots, f_n) \leqq cn \log m,$$

where c is a universal constant (e.g., $c = 40$ will do).

Proof. Let $a_0, \cdots, a_n, x_0, \cdots, x_n$ be the projection functions of k^{2n+2}. Then

$$f_i = a_0 x_i^n + \cdots + a_n$$

for all i. The set $\{a_0, \cdots, a_n, x_0, \cdots, x_n, f_0, \cdots, f_n\} \subset A$ is k-linearly independent: let

$$\kappa_0 a_0 + \cdots + \kappa_n a_n + \lambda_0 x_0 + \cdots + \lambda_n x_n + \mu_0 f_0 + \cdots + \mu_n f_n = 0, \tag{3.2}$$

where $\kappa_i, \lambda_i, \mu_i \in k$. Evaluating this equation at $(\boldsymbol{\alpha}, \boldsymbol{\xi}) \in k^{2n+2}$ with $\xi_j = 1$ and all other coordinates $= 0$, we get $\lambda_j = 0$. Evaluating (3.2) at $(\boldsymbol{\alpha}, \boldsymbol{\xi})$ with $\alpha_j = 1$ and all other coordinates $= 0$, we obtain $\kappa_j = 0$ unless $j = n$. Thus we have

$$\kappa_n a_n + \mu_0 f_0 + \cdots + \mu_n f_n = 0.$$

Evaluating this equation at $(\boldsymbol{\alpha}, \boldsymbol{\xi})$ with $\alpha_{n-1} = 1$ and $\xi_j = 1$ and all other coordinates $= 0$, we find $\mu_j = 0$. Then also $\kappa_n = 0$. Therefore $a_0, \cdots, a_n, x_0, \cdots, x_n, f_0, \cdots, f_n$ are indeed linearly independent. As is well known and easy to prove, this implies

$$L(f_0, \cdots, f_n) \geqq n + 1.$$

If we combine this lower bound with the lower bound of Proposition 3.1, we get the left-hand inequality of the present corollary.

The right-hand inequality follows directly from [4] or [1] in case $m = n + 1$. Now if $m = \#k$, we have $\xi^m = \xi$ for all $\xi \in k$, and therefore

$$\alpha_0 \xi_i^n + \cdots + \alpha_n = \beta_0 \xi_i^{m-1} + \cdots + \beta_{m-1},$$

where each β_j is a sum of certain of the α_l. Thus the problem is to evaluate a polynomial of degree $m-1$ at $n+1$ points, which can be done by separately treating blocks of m points each.

PROPOSITION 3.3. *Let k have at least* $n+1$ *elements, and let*

$$D := \{(\xi_0, \cdots, \xi_n, \eta_0, \cdots, \eta_n) \in k^{2n+2} : \xi_i \neq \xi_j \text{ for } i \neq j\}$$

and let A be the ring of all functions from D to k. Define $g_0, \cdots, g_n \in A$ *by*

$$\begin{aligned} \eta_0 &= g_0(\boldsymbol{\xi}, \boldsymbol{\eta}) \xi_0^n + \cdots + g_n(\boldsymbol{\xi}, \boldsymbol{\eta}), \\ &\cdots \\ \eta_n &= g_0(\boldsymbol{\xi}, \boldsymbol{\eta}) \xi_n^n + \cdots + g_n(\boldsymbol{\xi}, \boldsymbol{\eta}) \end{aligned}$$

for any $(\boldsymbol{\xi}, \boldsymbol{\eta}) \in D$. *Then*

$$L(g_0, \cdots, g_n) \geqq (n/2)\log n - 3n,$$

and therefore $L(g_0, \cdots, g_n)$ *has the order of magnitude* $n \log n$ (by [4]).

Proof. Since the graph of $(g_0, \cdots, g_n)$ is the same as the graph of $(f_0, \cdots, f_n)$ up to a permutation of the coordinate axes, the proof of Proposition 3.1 would seem to carry over immediately. This is not the case, however, as there is no

obvious way to handle the transition from $(f_0, \cdots, f_n)$ to $(\tilde{f}_0, \cdots, \tilde{f}_n)$. (Intersecting with a linear space does not work because the dimension of the intersection might be too large.) So we will give a separate proof, which is nevertheless quite similar to the proof of Proposition 3.1. We choose $\lambda_0, \cdots, \lambda_n \in k$ to be pairwise different and put

$$t := \lceil \sqrt{n+1} \rceil.$$

Let

$$\tilde{D} := \{(\xi_0, \cdots, \xi_n) \in \{\lambda_0, \cdots, \lambda_n\}^{n+1} : \xi_i \neq \xi_j \text{ for } i \neq j\} \subset k^{n+1},$$

and define

$$\tilde{g}_i(\xi_0, \cdots, \xi_n) := g_i(\xi_0, \cdots, \xi_n, \tilde{\lambda}_0, \cdots, \tilde{\lambda}_n)$$

for any $(\xi_0, \cdots, \xi_n) \in \tilde{D}$, where

$$\tilde{\lambda}_i := \lambda_{\lfloor i/t \rfloor}$$

for $0 \leqq i \leqq n$. Because of $\lfloor n/t \rfloor \leqq t-1$, there are at most t different $\tilde{\lambda}_i$. If $\tilde{A}$ denotes the ring of all functions from $\tilde{D}$ to k, we have

$$L_A(g_0, \cdots, g_n) \geqq L_{\tilde{A}}(\tilde{g}_0, \cdots, \tilde{g}_n). \tag{3.3}$$

We will apply Theorem 2.3 to $\tilde{g}_0, \cdots, \tilde{g}_n$. Let $\tilde{S}$ be the graph of the map

$$(\tilde{g}_0, \cdots, \tilde{g}_n) : \tilde{D} \to k^{n+1},$$

i.e., $\tilde{S}$ is the set of all $(\boldsymbol{\xi}, \boldsymbol{\alpha}) \in k^{2n+2}$ such that $\boldsymbol{\xi} \in \tilde{D}$ and

$$\tilde{\lambda}_i = \alpha_0 \xi_i^n + \cdots + \alpha_n \tag{3.4}$$

for $0 \leqq i \leqq n$. Denote the coordinate projections of k^{2n+2} by $x_0, \cdots, x_n, a_0, \cdots, a_n$, and put

$$\begin{aligned} l_1 &= a_0 \lambda_0^n + \cdots + a_n, \\ &\cdots \\ l_{n+1} &= a_0 \lambda_n^n + \cdots + a_n, \\ l_{n+2} &= x_0 \\ &\cdots \\ l_{2n+2} &= x_n. \end{aligned}$$

Then condition 2 of Theorem 2.3 is satisfied, since $l_1, \cdots, l_{2n+2}$ even separate the points of k^{2n+2}. As to condition 1, if $i \leqq n+1$, then l_i assumes on all of $\tilde{S}$ at most the values $\tilde{\lambda}_0, \cdots, \tilde{\lambda}_n$, since for $(\boldsymbol{\xi}, \boldsymbol{\alpha}) \in \tilde{S}$ some ξ_j is equal to λ_{i-1} and therefore

$$\begin{aligned} l_i(\boldsymbol{\xi}, \boldsymbol{\alpha}) &= \alpha_0 \lambda_{i-1}^n + \cdots + \alpha_n \\ &= \alpha_0 \xi_j^n + \cdots + \alpha_n = \tilde{\lambda}_j \end{aligned}$$

by (3.4). But there are not more than t different elements among $\tilde{\lambda}_0, \cdots, \tilde{\lambda}_n$. On the other hand, $l_{n+2+i} = x_i$ assumes at most t values on $T := \tilde{S} \cap \{l_1 = c_1, \cdots, l_{n+1} = c_{n+1}\}$ for the following reason: we may assume that T is nonempty. The conditions $l_1 = c_1, \cdots, l_{n+1} = c_{n+1}$ are equivalent to fixing the last

$n+1$ coordinates of points $(\boldsymbol{\xi}, \boldsymbol{\alpha})$. Therefore all points of T have the same $\alpha_0, \cdots, \alpha_n$. Considering one such point, we conclude from (3.4) that $\alpha_0, \cdots, \alpha_n$ are the coefficients of a polynomial which assumes the value $\tilde{\lambda}_i = \lambda_{\lfloor i/t \rfloor}$ at most t times on the set $\{\lambda_0, \cdots, \lambda_n\}$, say for $\gamma_1, \cdots, \gamma_s$. But then (3.4) implies that any point $(\boldsymbol{\xi}, \boldsymbol{\alpha}) \in T$ has $\xi_i \in \{\gamma_1, \cdots, \gamma_s\}$. This confines the range of x_i to at most t values. Theorem 2.3 now implies

$$L_{\tilde{A}}(\tilde{g}_0, \cdots, \tilde{g}_n) \geqq \log(\# \tilde{D}/t^{n+1})$$

$$\geqq \log((\sqrt{n+1}+1)^{-(n+1)}(n+1)!) \geqq (n/2)\log n - 3n.$$

Together with (3.3), this yields Proposition 3.3.

4. Remarks. 1. The complexity of the set of elementary symmetric functions in n inputs over a (small) finite field remains open. On the one hand, no upper bound better than $n \log n$ seems to be known even for $k = \mathbb{Z}_2$; on the other hand, the method of this paper gives only linear lower bounds as long as, e.g., $\# k \leqq n$ (since then the image of the elementary symmetric map has only exponential size).

2. While for the evaluation of rational functions over an infinite field the model of a straight line algorithm is adequate (see [3]), this is not the case for the evaluation of finite functions (unless one is interpreting algorithms as hardware constructions). In fact, if we allow branching instructions of the form "if $f = 0$ then goto i else goto j" free of charge and if, e.g., $k = \mathbb{Z}_p$, then by a table look-up procedure we may evaluate any function from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$ without a single multiplication or division.

Nevertheless, if we count the branching instructions in the same way as multiplications and divisions, the results of this paper remain valid. We can argue as follows: given an algorithm for computing functions $f_1, \cdots, f_r$ from k^d to k in time τ (worst case), we may apply it to inputs of K^d. The algorithm will then compute partial functions $F_1, \cdots, F_r$ from K^d to K which in general will not be rational but which will extend $f_1, \cdots, f_r$. A variant of [5, § 5, second Theorem] shows that the graph of the map $(F_1, \cdots, F_r)$ is contained in a union of finitely many closed irreducible subvarieties $P_1, \cdots, P_m$ of dimension $\leqq d$ of K^{d+r} such that

$$\tau \geq \log\left(\sum_i \deg P_i\right).$$

Now we can apply (2.1) to each P_i.

Acknowledgments. Many thanks go to Joachim von zur Gathen and to a referee for improving the presentation of the paper.

REFERENCES

[1] A. BORODIN AND I. MUNRO, *Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, to appear.

[2] V. STRASSEN, *Berechnung und Program I*, Acta Informatica, 1 (1972), pp. 320–335.

[3] ———, *Berechnung und Program II*, Ibid., 2 (1973), pp. 64–79.

[4] ———, *Die Berechnungskomplexität von elementar-symmetrischen Funktionen und von Interpolationskoeffizienten*, Numer. Math., 20 (1973), pp. 238–251.

[5] ———, *The Computational Complexity of Continued Fractions*, in preparation.

PREFACE

The last decade has seen significant advances in our knowledge of programming language semantics and of logics for reasoning about programs. This decade is seeing these developments evaluated in practice. This special issue of the *SIAM Journal on Computing* is devoted entirely to these two closely related subjects of "semantics" and "correctness" of programs.

One way to perceive past developments is to consider the problem of proving statements about algorithmic processes. Proofs about constructions in Euclidean geometry are particularly vivid examples of how this has been done. There the proofs serve to crystalize our intuitions about space. When we prove statements about computer programs, we must deal with concepts such as "memory location", "instruction sequence", etc. some of which may be special to a particular computer; and therefore our proofs and even our axioms may not embody any widely understood intuitions about computing. As we deal with higher level programming languages and get further away from the idiosyncrasies of machines, we might expect our reasoning to become more like the logic of constructions learned in mathematics. However, this has not yet proved true.

Existing high level programming languages such as FORTRAN, ALGOL 60 or PL/I are quite sophisticated and involve new concepts not found in ordinary mathematics. Indeed, the principles of reasoning needed for these languages are subtle; and when it becomes necessary to be quite certain that a program is correct, one must move to a precise and more formal style of thinking. We are led to seek explicit rules of reasoning and perhaps even systems of *proof*.

For rules of proof about algorithms to be useful, they should correspond to our intuitive understanding of programs. They should also be general enough to apply to programming language *concepts* rather than particular features of a specific language. (These apparently simple criteria preclude using the logician's trick of arithmetizing programs and thereby reducing proofs about them to number theory.) In the technical vocabulary, this means that the principles of a *programming logic* should be applicable to abstract programs or *program schemas*.

In order to be precise about these rules, one must have a precise definition of the programming language. Certainly one can give such a definition in terms of a compiler or in terms of some other translation of the language into a simpler algorithmic language or into a more primitive logical setting. But, if one is to use this definition to ascertain the validity of rules for reasoning about programs, then the definition should be of roughly the same level of detail as the logic. Moreover, the definition should allow the nonspecialist to understand the language while at the same time serving the expert as a standard for implementation and as a definitive arbiter of disputes.

Progress toward a rigorous but intuitive logic and a precise but nonmachine oriented semantics for high level programming languages has been slow. Not only are there serious conceptual problems to be solved, but the results of the work must be applicable to "real" programming languages. In fact, much of the early work on semantics was oriented toward the very real language ALGOL 60.

The ALGOL 60 report generated early interest in formal semantics partly because the syntax of the language was presented in an elegant formal way which contrasted with its vague and informal semantics. The challenge was to give a similarly formal semantics.

In this same period, computer scientists recognized the need for fresh logical concepts to enable them to reason about new notions appearing in ALGOL 60, such as block structure, assignment statements, recursive procedures, etc.; however, no logical systems nor successful semantic descriptions were proposed for this language until the late sixties.

One of the reasons that progress on the axiomatics of ALGOL-like languages was slow is that early work on their semantics encountered unexpected complexity. For instance, the use of the lambda calculus to model the procedure mechanisms suggested that finding a high level semantics for ALGOL 60 would be as hard as finding one for the lambda calculus. But the lambda calculus poses serious logical problems because it involves type free procedures, e.g., procedures which could take themselves as arguments. Logicians had already discovered serious difficulties in developing a logic for such objects. Dana Scott's paper in this volume reports substantial progress on this difficult problem, while the paper of Christopher Wadsworth is concerned with the lambda calculus as representative of questions about programming languages that arise when using a "denotational semantics" for programs in the style of D. Scott and C. Strachey.

Essentially, progress on the logic of ALGOL-like programs had to wait until the effort was concentrated on fragments of ALGOL. It is now argued by some that we cannot hope to develop understandable proof rules for languages as complex as ALGOL 60. The paper by Edward Ashcroft and William Wadge presents the language LUCID which illustrates the type of logic that one can expect if the language is tailored to the logic.

The earliest progress on the problem of specifying a logic for reasoning about programs came from the direction of applicative languages like LISP. In his 1963 paper, *A basis for a mathematical theory of computation*,[1] John McCarthy proposed the principle of *recursion induction* for reasoning about recursive programs. He illustrated his rule using program schemas in the form of recursion equations. (The Russian Y. I. Ianov had presented a study of program schemas of a more elementary form (flowcharts) which influenced McCarthy.)

A substantial mathematical theory of program schemas was developed in the late sixties. The paper by Ashok Chandra in this volume illustrates the style of some of this work, and the paper by Steven S. Muchnick points to new directions for the subject. This schema theory proved helpful in formulating logics for reasoning about programs.

The discovery of *fixed point induction* as a proof rule and its formulation in various logical systems such as the *mu-calculus* and the *relational calculus*, has clarified some of the conceptual problems about *programming logics*. The success of these logics has in turn affected the style of formal semantics. It is clearly important to use a semantics compatible with the logic. This means that the use of *least fixed point* methods has become important in formal semantics. The paper of

[1] See D. Scott's bibliography.

Peter Downey and Ravi Sethi discusses the connection between fixed point semantics and other more operational approaches to program meaning. The paper of Zohar Manna and Adi Shamir considers the alternatives to the *least* fixed point semantics. The paper of Gordon Plotkin considers extending the theory to cover parallel programs.

The logical foundation for programming logics provided by the fixed point induction rule can also be used to support proof systems for ALGOL-like languages. The most popular logics of this type are based on C. A. R. Hoare's adaptation and extensions of the *inductive assertion* method of Floyd (see Hoare (1969 and Floyd (1967) in D. Scott's bibliography). In particular, Hoare's axiom for recursive procedures is a version of the fixed point induction rule. In this volume some practical aspects of axiomatics for ALGOL-like languages are considered in the paper by Susan Gerhart.

As of 1976 the methods and results of the past decade of study of programming language semantics and programming logics stand on the verge of their most serious testing. Computer scientists are trying to apply these methods and results to the design and description of programming languages and to the construction of embryonic program verification systems. Developments in the next decade should be exciting and critical. We hope that the interested reader can find his way into this diverse subject through this special issue.

There are many figures whose influence on these fields cannot adequately be reflected in the dry statistics of bibliographic cross-reference. Among these we have chosen to single out Christopher Strachey, whose death in 1975 came as a personal blow to many of those contributing to this volume. Strachey's dogged insistence that the act of programming was often an excursion into a world of ill-understood abstractions was profound stimulus to his colleagues. Without that insistence, and without Strachey's own pioneering work in the field, much that is described below would not have happened.

ROBERT L. CONSTABLE
DAVID PARK

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

LUCID—A FORMAL SYSTEM FOR WRITING AND PROVING PROGRAMS*

E. A. ASHCROFT† AND W. W. WADGE‡

Abstract. Lucid is both a programming language and a formal system for proving properties of Lucid programs. The programming language is unconventional in many ways, although programs are readily understood as using assignment statements and loops in a "structured" fashion. Semantically, an assignment statement is really an equation between "histories", and a whole program is simply an unordered set of such equations.

From these equations, properties of the program can be derived by straightforward mathematical reasoning, using the Lucid formal system. The rules of this system are mainly those of first order logic, together with extra axioms and rules for the special Lucid functions.

This paper formally describes the syntax and semantics of programs, and justifies the axioms and rules of the formal system.

Key words. program proving, formal semantics, formal systems

Introduction. Lucid is both a language in which programs can be written, and a formal system for proving properties of such programs. These properties are also expressed in Lucid. This is possible because a Lucid program is itself simply an unordered set of assertions, or axioms, from which other assertions may be derived by fairly conventional mathematical reasoning. The statements in Lucid programs are special cases of Lucid terms.

In this paper we present the formal basis for Lucid, giving its semantics and justifying various axioms and rules of inference that are used in Lucid proofs. An informal introduction to Lucid can be found in [1], together with a discussion of implementation considerations. This paper will be rather formal, with motivating explanations and examples confined mainly to this introduction.

The language considered here might be called Basic Lucid, since it does not include features like arrays and defined functions. Such extensions are considered in [1].

The main idea in Lucid is that programs should be "denotational" and "referentially transparent", even when they contain assignment statements. This means that all expressions in a program must mean something, and that two occurrences of the same expression in a program must denote the same "something". Lucid achieves this aim, and yet manages to treat assignment statements as equations. (Thus, if a Lucid program contains the (assignment) statement $Y = X + Z$, every occurrence of Y in the program can be replaced by $X + Z$, without changing the meaning of the program.) This is accomplished by considering the program to be talking about the "histories" of the various variables. Semantically, all expressions in programs without nested loops will denote infinite sequences of

* Received by the editors April 22, 1975, and in revised form November 20, 1975.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

‡ Computer Science Department, University of Warwick, Coventry, England CV4 7AL.

data objects. Reassignment to a variable must be done by using the special Lucid function **next**, and the initialization of a variable must also be explicit, by using the function **first**. Thus the two statements **first** $X = 0$, **next** $X = X + 1$ define the value, or history, of X to be the infinite sequence $\langle 0, 1, 2, 3, \cdots \rangle$. (The numeral 1 denotes $\langle 1, 1, 1, \cdots \rangle$, and + works pointwise. The function **next** drops off the first item of its argument.) Note that these two statements imply the existence of a "loop", and explicit control statements are unnecessary. Also, the order of the two statements is irrelevant. If we also have **first** $Y = 0$, **next** $Y = Y + X \times X$, then this loop also generates a history for Y, namely, $\langle 0, 0, 1, 5, 14, \cdots \rangle$. We can get out of the infinite iteration using the Lucid function **as soon as**: e.g., *output* = Y **as soon as** $X > 3$ gives the variable *output* the value of Y when $X > 3$ is first true, i.e., the fifth value, 14. (In fact, *output* is $\langle 14, 14, 14, \cdots \rangle$.) With these three functions it is possible to write programs without nested loops, in a very natural way.

For programs with nested loops we must generalize our notion of "history". Consider the following Lucid program, which determines whether the first integer N on the input stream is a prime number or not.

Program Prime.

$N =$ **first** *input*
first $I = 2$
begin
    first *multiple* $= I \times I$
    next *multiple* $=$ *multiple* $+ I$
    IdivN $=$ *multiple* eq N **as soon as** *multiple* $\geqq N$
end
next $I = I + 1$
output $= \neg$*IdivN* **as soon as** *IdivN* $\vee$ $I \times I \geqq N$.

The program contains one loop within another. The inner loop is delimited by *begin* and *end*. Intuitively, the outer loop generates successive values of potential divisors I of N, starting $2, 3, 4, \cdots$, and, for each value of I, the inner loop generates successive multiples of I, beginning with I^2. The variable *IdivN* is set *true* or *false* depending on whether or not a multiple of I is found which is equal to N. In the outer loop, *output* is set *false* or *true* depending on whether *IdivN* is ever *true* or not.

The predicate "eq" is like "=" except that its value is *undefined* if either of its arguments is *undefined* (of course *undefined* = *undefined* is *true*).

The program as it stands is not strictly speaking a set of assertions because of the *begin* and *end*. Informally, the effect of *begin* and *end* is to "freeze" the values of the *global* variables I, N and *IdivN*. (The global variables of a loop are all those variables mentioned outside the loop.) The *begin* and *end* can be removed by replacing all enclosed occurrences of I, N and *IdivN* by **latest** I, **latest** N and **latest** *IdivN*. (The meaning of the function **latest** will be given later.) Thus the first line of the inner loop becomes **first** *multiple* = **latest** $I \times$ **latest** I. The resulting transformed program Prime′ is an unordered set of assertions which can be used as axioms from which to derive further assertions.

In practice, it is easier to write programs using the *begin* ... *end* notation rather than **latest** and, moreover, we have rules of inference which allows us to

carry out proofs of programs in the *begin* ... *end* notation, without using **latest**, as follows. We keep track of the loop associated with a program statement or other assertion that we have derived. Informally, an assertion that does not contain any of the special Lucid functions can be moved into loops, and can be moved out of loops if in addition it only refers to global variables of the loop. Moreover, within a loop we can add the assertion $X = \textbf{first}\ X$ for any global variable X (which states that X is *quiescent*, i.e., constant for the duration of the loop). When proving things "within a loop" we may only use statements from within the loop (which may have been brought there or have been added as above).

This might be called the technique of "nested proofs". It reduces reasoning about nested loops to reasoning about simple loops. Before we discuss generalized histories we can illustrate this sort of reasoning by deriving from the statements of Prime and the assumption **first** *input* > 0 the assertion

$$ouput = \neg\exists L \exists K\ 2 \leqq K < \textbf{first}\ input\ \wedge\ L \times K = \textbf{first}\ input.$$

Proof. We will assume the only data objects are the integers, *true*, *false* and the special object *undefined*.

The first step is to prove the correctness of the inner loop. We introduce a new variable J by setting **first** $J = I$ and **next** $J = J + 1$ so that between the *begin* and *end* we have

first $J = I$
next $J = J + 1$
first *multiple* $= I \times I$
next *multiple* $=$ *multiple* $+ I$
IdivN $=$ *multiple* eq N **as soon as** *multiple* $\geqq N$.

Since J does not appear elesewhere in the program, any assertion not involving J which is true for the expanded program is true for the original. With J so defined we can prove

(1) $$multiple = I \times J.$$

The proof uses the basic Lucid induction rule:

(R1) $$\textbf{first}\ P,\ P \rightarrow \textbf{next}\ P \models P,$$

where for any assertion A and set Γ of assertions, $\Gamma \models A$ means that the truth of A is implied by the truth of every assertion in Γ.

If we let P be "*multiple* $= I \times J$", then

$$\begin{aligned}\textbf{first}\ P &= \textbf{first}\ (multiple = I \times J)\\ &= (\textbf{first}\ multiple = \textbf{first}\ I \times \textbf{first}\ J)\\ &= (I \times I = I \times I)\end{aligned}$$

which is true (we used the fact that **first** $I = I$ inside the inner loop).

Now we assume that P is true at some stage; i.e., we assume $multiple = I \times J$. Then,

$$\begin{aligned} \textbf{next}\ (multiple = I \times J) &= (\textbf{next}\ multiple = \textbf{next}\ I \times \textbf{next}\ J) \\ &= (multiple + I = I \times (J+1)) \\ &= (multiple + I = I \times J + I) \end{aligned}$$

which is true because of the induction assumption. We can discharge the assumption P, giving $P \rightarrow \textbf{next}\ P$, and so we have proved $multiple = I \times J$ by induction.

We also used an axiom which says that **first** and **next** "commute" with conventional operations like "+": for any expression A not containing any special Lucid functions and having free variables $X_1, X_2, \cdots, X_K$ we have

(A1) $$\models \textbf{first}\ A = A(X_1/\textbf{first}\ X_1, \cdots, X_K/\textbf{first}\ X_K)$$

(A2) $$\models \textbf{next}\ A = A(X_1/\textbf{next}\ X_1, \cdots, X_K/\textbf{next}\ X_K),$$

where $A(X/Q)$ denotes term A with free variable X replaced by term Q.

Having proved $multiple = I \times J$ we can replace any occurrence of *multiple* in our program by $I \times J$. The program can then be simplified, to give program Prime_1:

$N = \textbf{first}\ input$
first $I = 2$
begin
  first $J = I$
  next $J = J + 1$
  $IdivN = I \times J$ eq N **as soon as** $I \times J \geqq N$
end
next $I = I + 1$
$ouput = \neg IdivN$ **as soon as** $IdivN \vee I \times I \geqq N$

and if A is any assertion without free occurrences of J, then $\text{Prime} \models A$ iff $\text{Prime}_1 \models A$.

To finish the proof of correctness of the inner loop we must determine the value of $IdivN$.

To do this we first introduce the function **hitherto**, defined by

(A3) $$\begin{aligned} &\models \textbf{first hitherto}\ P = \text{T} \\ &\wedge\ \textbf{next hitherto}\ P = P \wedge \textbf{hitherto}\ P. \end{aligned}$$

Using this we can establish by induction that

(2) $$\textbf{hitherto}\ (I \times J < N) \rightarrow (\forall K\, I \leqq K < J \rightarrow I \times K < N).$$

If we define **firstime** P to mean $P \wedge$ **hitherto** $\neg P$, we can conveniently state an axiom for the function **as soon as**, and a rule for **firstime**:

(A4) $$\models \textbf{firstime}\ P \rightarrow X\ \textbf{as soon as}\ P = X,$$

(R2) $$\textbf{firstime}\ P \rightarrow \textbf{first}\ Q, \textbf{eventually}\ P \models \textbf{first}\ Q.$$

The first states that the value of X **as soon as** P is the value of X when P is true for the first time. The second states that if some property Q holds when P is true for the first time, and Q is quiescent and P does eventually become true, then property Q holds.

Using (2) we can establish

(3) $\quad$ **firstime** $(I+J \geqq N) \rightarrow I \times J$ eq $N = (\exists K\, I \leqq K < N \wedge I \times K = N)$.

(A proof of this step can be found in [2].)

Since (A4) gives us

$$\textbf{firstime}\ (I \times J \geqq N) \rightarrow IdivN = I \times J \text{ eq } N$$

we can conclude

$$\textbf{firstime}\ (I \times J \geqq N) \rightarrow IdivN = (\exists K\, I \leqq K < N \wedge I \times K = N).$$

Since the term on the right-hand side is quiescent, to apply rule (R2) we simply need to prove **eventually** $(I \times J \geqq N)$. For this we use the following: "termination" rule for integers:

(R3) $\quad$ $integer\, L, L >$ **next** $L \models$ **eventually** $(L \leqq 0)$.

To apply (R3) we first prove that *integer* $(N - I \times J)$ and $N - I \times J >$ **next** $(N - I \times J)$. This is straightforward (but note that $I > 0$ must be established by induction in the outer loop, and then brought into the inner loop).

Now, applying the **as soon as** rule (R2), we get

(4) $\quad$ $IdivN = \exists K\, I \leqq K < N \wedge I \times K = N.$

Assertion (4) contains no Lucid functions and all its free variables are globals, and so it may be taken outside the inner loop. We now discard the inner loop yielding "program" Prime_2:

$N =$ **first** *input*
first $I = 2$
$IdivN = \exists K\, I \leqq K < N \wedge I \times K = N$
next $I = I + 1$
output $= \neg IdivN$ **as soon as** $IdivN \vee I \times I \geqq N$

and as before $\text{Prime}_2 \models A$ implies $\text{Prime} \models A$ for any assertion A. Actually Prime_2 is no longer a program but rather a hybrid object halfway between a program and statement of correctness.

Note that $IdivN$ is always either true or false (it could be *undefined* if N were *undefined*, but we know $N > 0$).

Now to finish the proof that

$$output = \neg \exists L \exists K\ 2 \leqq K < N \wedge L \times K = N$$

we must first show that

(5) **firstime** $(IdivN \vee I \times I \geqq N) \rightarrow IdivN = \exists L \exists K\ 2 \leqq K < N \wedge L \times K = N.$

The proof of this is similar to the proof of (3), provided we first prove that

(6) **hitherto**$\neg$*IdivN* $\rightarrow$ ($\forall L\, L<I \rightarrow (\neg\exists K\, 2\leqq K<N \wedge L\times K=N)$).

This requires a straightforward induction proof, making extensive use of properties of integers, and the property $N>0$.

Since *IdivN* is always either true or false, to establish the second premise for the **as soon as** rule, namely, **eventually** (*IdivN* $\vee$ $I\times J\geqq N$), it is sufficient to show that **eventually** ($I\times I\geqq N$). This follows from the termination rule (R3).

So finally, we can eliminate all the variables except *output* leaving "program" Prime_3:

$$output = \neg\exists L\exists K\, 2\leqq K<\textbf{first}\ input \wedge L\times K=\textbf{first}\ input. \qquad \square$$

Note that $\neg\exists L\exists K\, 2\leqq K<$ **first** *input* $\wedge$ $K\times L=$ **first** *input* is either *true* or *false*, when *integer* **first** *input*. Thus *output* is not *undefined*, and so program Prime terminates with the correct result.

This sample proof shows that it is possible to reason about programs knowing very little of the formal semantics, in particular knowing very little about the semantics of nested loops. But we must give a semantics for nested loops to justify the nested proof style of reasoning.

In the program Prime, the history of I can be thought of as $\langle 2, 3, 4, \cdots\rangle$, but the history of *multiple* must be $\langle\langle 4, 6, 8, \cdots\rangle$, $\langle 9, 12, 15, \cdots\rangle$, $\langle 16, 20, 24, \cdots\rangle, \cdots\rangle$, i.e., a two-dimensional infinite sequence. A one-dimensional infinite sequence can be considered as a function from the natural numbers $\mathcal{N}$ (including zero) to data elements, and, similarly, a two-dimensional sequence is a function from $\mathcal{N}\times\mathcal{N}$ to data elements. If we write I_n instead of $I(n)$, we see that $I_n = n+2$. For two dimensions, we adopt the convention that the first subscript is the more rapidly varying time parameter, the number of iterations of the *inner* loop. Thus $multiple_{nm} = (m+2)(m+n+2)$. The Lucid functions (except **latest**) act on the **first** time parameter, e.g., (**first** *multiple*)$_{nm}$ = $multiple_{0m}$.

To make this work we need to do two things. Firstly, we get rid of the *begin* ... *end* notation as indicated previously, by introducing the function **latest** which increases the number of time dimensions, e.g., (**latest** I)$_{nm}$ = I_m. (Note that **latest**) act on the first time parameter, e.g., (**first** *multiple*)$_{nm}$ = $multiple_{0m}$.
levels of loop nesting by considering *all* histories as depending on an *infinite number* of time parameters (only a finite number of which will usually be necessary for each variable).

Thus, in the rest of this paper we consider Lucid programs that use **latest** instead of *begin* ... *end*, and the semantics of Lucid is given in terms of functions of infinite sequences of time parameters.

As a formal system Lucid is similar, in some respects, to first order logic. On the other hand, Lucid can be viewed as a tense logic, a branch of modal logic which formalizes certain kinds of reasoning about time. (The suitability of modal logic for proofs about programs has already been recognized by Burstall [3].) In Lucid a term, such as $X>Y$, need not be simply true or false. It can be *true* at some "times" and *false* at others (and even *undefined* at others). As we have seen, semantically, the value of $X>Y$ depends on various time parameters because the

values of the variables X and Y themselves depend on time parameters. As a result of this, certain properties of first order logic, such as the deduction theorem, fail to hold for Lucid, except in special circumstances.

Lucid also differs from first order logic in that we wish to allow programs to compute truth values, and therefore we have to allow an "undefined" truth value, for sub-programs which do not terminate. (Since we have this undefined truth value, we can abolish the distinction between terms and formulas, logical connectives applied to non-truth-values acting as they would for the undefined value. This uniform treatment is not essential however—it merely simplifies the formal treatment.) The formal system must be able to deal with "undefined" within the logic. This means, for example, that the law of the excluded middle does not hold.

Nevertheless, the rules of inference for Lucid are almost identical to those for first order logic.

Sections 1 to 3 of the paper are devoted to setting up the interpretations on which the semantics is based. Then in § 4 we define the class of sets of terms that are Lucid programs. We show that every program has a unique minimal solution, or "meaning". In the rest of the paper we discuss a formal system for proving properties of programs, justifying the sort of reasoning used in the proofs given in the Introduction. In particular, in § 7 we justify the "nested proof" technique for proving things about programs with nested loops.

1. Formalism. The meanings of programs will be based on "computation structures", which in turn are defined in terms of simple structures. We first define a general notion of structure, and build on this in later sections.

1.1. Syntax. A Lucid *alphabet* Σ is a set containing the symbols "U", "$\exists$" and, for each natural number n, any number of n-ary operation symbols, including, for $n = 0$, the nullary operation symbol T.

We also have at our disposal a set of variables, e.g., X, Y, Z.

The set of Σ-*terms* is defined as follows:

(a) every variable is a Σ-term;

(b) if G is an n-ary operation symbol in Σ and $A_1, \cdots, A_n$ are Σ-terms, then $G(A_1, \cdots, A_n)$ is a Σ-term;

(c) if V is a variable and A is a Σ-term, then $\exists VA$ is a Σ-term.

1.2. Semantics. If Σ is an alphabet, then a Σ-*structure* S is a function which assigns to each symbol σ in Σ a "meaning" σ_S in such a way that U_S is a set, $\exists_S$ is a function from subsets of U_S to elements of U_S and, if G is an n-ary operation symbol, G_S is an n-ary operation on U_S.

An S-*interpretation* $\mathcal{I}$ extends S to assign to each variable V an element $V_{\mathcal{I}}$ of U_S.

If $\mathcal{I}$ is an S-interpretation, V is a variable and α is an element of U_S, then $\mathcal{I}(V/\alpha)$ denotes the S-interpretation differing from I only in that it assigns α to V.

If A is a Σ-term, S a Σ-structure and $\mathcal{I}$ an S-interpretation, then we define an element $|A|_{\mathcal{I}}$ of U_S (the "meaning" of A) as follows:

(a) for variable V, $|V|_{\mathcal{I}}$ is $V_{\mathcal{I}}$,

(b) for Σ-terms $A_1, A_2, \cdots, A_n$ and n-ary operation symbol G of Σ,

$$|G(A_1, \cdots, A_n)|_{\mathcal{I}} \quad \text{is} \quad G_S(|A_1|_{\mathcal{I}}, |A_2|_{\mathcal{I}}, \cdots, |A_n|_{\mathcal{I}}),$$

(c) for Σ-term A and variable V

$$|\exists VA|_{\mathscr{I}} = \exists_S(\{|A|_{\mathscr{I}(V/\alpha)} : \alpha \in U_S\}).$$

We say: $\models_{\mathscr{I}} A$ ($\mathscr{I}$ *satisfies* A or A is *valid* in $\mathscr{I}$) iff $|A|_{\mathscr{I}} = T_S$; if Γ is a set of terms, then $\models_{\mathscr{I}} \Gamma$ iff $\models_{\mathscr{I}} B$ for each B in Γ; $\Gamma \models_S A$ iff $\models_{\mathscr{I}} \Gamma$ implies $\models_{\mathscr{I}} A$ for all S-interpretations $\mathscr{I}$.

2. Basic results. Even in general structures we can establish several useful properties.

2.1. Substitution. An occurrence of a variable V in a Σ-term A is *bound* if and only if the occurrence is in a sub-term of A of the form $\exists VB$; otherwise the occurrence is *free*. If A and Q are Σ-terms and V is a variable, then $A(V/Q)$ is the term formed by replacing all free occurrences of V in A by Q. In this situation V is said to be *free for* Q *in* A iff this substitution does not result in a free variable in Q becoming bound in $A(V/Q)$, i.e., iff V does not occur free in A in a sub-term of the form $\exists WB$ for some variable W occurring free in Q.

LEMMA 1. *For Σ-structure S, S-interpretation $\mathscr{I}$, Σ-terms A and Q and variable V, if V is free for Q in A, then*

$$|A(V/Q)|_{\mathscr{I}} = |A|_{\mathscr{I}(V/|Q|_{\mathscr{I}})}.$$

Proof. The proof of the analogous result for first order logic carries over directly. □

2.2. Power structures. One way of building structures out of simpler structures is by a generalized Cartesian product.

2.2.1. For any Σ-structure S and any set X, S^X is the unique Σ-structure $\mathscr{C}$ such that

(a) $U_{\mathscr{C}}$ is the set of all functions from X to U_S. If $x \in X$ and $\alpha \in U_{\mathscr{C}}$, we will write α_x instead of $\alpha(x)$.

(b) If G is an operation symbol in Σ and $\alpha, \beta, \cdots \in U_{\mathscr{C}}$ and $x \in X$, then $(G_{\mathscr{C}}(\alpha, \beta, \cdots))_x = G_S(\alpha_x, \beta_x, \cdots)$.

(c) If K is a subset of $U_{\mathscr{C}}$ and $x \in X$, then $(\exists_{\mathscr{C}}(K))_x = \exists_S(\{\alpha_x : \alpha \in K\})$.

Thus S^X carries over the operations and quantifiers of S by making them work "pointwise" on the elements of $U_{\mathscr{C}}$. Thus all nullary operation symbols are assigned constant functions. In particular, $T_{\mathscr{C}}$ is the constant function on X with value T_S.

2.2.2. For S^X-interpretation $\mathscr{I}$ and $x \in X$, $\mathscr{I}_x$ denotes the unique S-interpretation which assigns each variable V the value $(V_{\mathscr{I}})_x$.

The following lemma establishes that every Σ-term acts "pointwise" in S^X, even those terms containing quantifiers.

LEMMA 2. *For any Σ-structure S, S^X-interpretation $\mathscr{I}$, Σ-term A and element $x \in X$,*

$$(|A|_{\mathscr{I}})_x = |A|_{\mathscr{I}_x}.$$

Proof. Let $\mathscr{C}$ be the structure S^X. The proof proceeds by structural induction on A.

(a) If A is a variable the result is immediate.

(b) If A is $G(A_1, \cdots, A_n)$ for n-ary operation symbol G in Σ, and Σ-terms $A_1, \cdots, A_n$, then

$$\begin{aligned}(|A|_{\mathscr{I}})_x &= (|G(A_1, \cdots, A_n)|_{\mathscr{I}})_x \\ &= G_S((|A_1|_{\mathscr{I}})_x, \cdots, (|A_n|_{\mathscr{I}})_x) \\ &= G_S(|A_1|_{\mathscr{I}_x}, \cdots, |A_n|_{\mathscr{I}_x}) \\ &= |G(A_1, \cdots, A_n)|_{\mathscr{I}_x}.\end{aligned}$$

(c) If A is $\exists VB$ for some variable V and Σ-term B, then

$$\begin{aligned}(|\exists VB|_{\mathscr{I}})_x &= (\exists_{\mathscr{C}}(\{|B|_{\mathscr{I}(V/\alpha)} : \alpha \in U_{\mathscr{C}}\}))_x \\ &= \exists_S(\{(|B|_{\mathscr{I}(V/\alpha)})_x : \alpha \in U_{\mathscr{C}}\}) \\ &= \exists_S(\{|B|_{\mathscr{I}_x(V/\alpha_x)} : \alpha \in U_{\mathscr{C}}\}) \\ &\qquad \text{(since if } \mathscr{I}' = \mathscr{I}(V/\alpha), \text{ then } \mathscr{I}'_x = \mathscr{I}_x(V/\alpha_x)) \\ &= \exists_S(\{|B|_{\mathscr{I}_x(V/\alpha)} : a \in U_S\}) \\ &\qquad \text{(since as } \alpha \text{ ranges over } U_{\mathscr{C}}, \alpha_x \text{ ranges over } U_S) \\ &= |\exists VB|_{\mathscr{I}_x}. \qquad \square\end{aligned}$$

It follows that S and S^X have the same theory:

COROLLARY 2.1. *For any Σ-structure S, any set X, any Σ-term A and set Γ of Σ-terms, if $\mathscr{C} = S^X$ then*

$$\Gamma \models_S A \quad \textit{iff} \quad \Gamma \models_{\mathscr{C}} A.$$

Proof. Suppose first that $\Gamma \models_S A$. Let $\mathscr{J}$ be a $\mathscr{C}$-interpretation such that $\models_{\mathscr{J}} \Gamma$. Then for any B in Γ and any x in X, $T_S = (T_{\mathscr{C}})_x = (|B|_{\mathscr{J}})_x = |B|_{\mathscr{J}_x}$ by Lemma 2. Thus $\models_{\mathscr{J}_x} \Gamma$ and so $\models_{\mathscr{J}_x} A$; hence $|A|_{\mathscr{J}_x} = T_S$. Therefore $(|A|_{\mathscr{J}})_x = |A|_{\mathscr{J}_x} = T_S = (T_{\mathscr{C}})_x$. Since x was arbitrary, $|A|_{\mathscr{J}} = T_{\mathscr{C}}$ and so $\models_{\mathscr{J}} A$.

Now suppose $\Gamma \models_{\mathscr{C}} A$. Let $\mathscr{I}$ be an S-interpretation such that $\models_{\mathscr{I}} \Gamma$. Define the $\mathscr{C}$-interpretation $\mathscr{J}$ by setting $(V_{\mathscr{J}})_x = V_{\mathscr{I}}$ for each x in X and each variable V, i.e., $\mathscr{J}_x = \mathscr{I}$ for each x. Then, for any B in Γ, $(|B|_{\mathscr{J}})_x = |B|_{\mathscr{J}_x} = |B|_{\mathscr{I}} = T_S = (T_{\mathscr{C}})_x$ for any x in X and so $\models_{\mathscr{J}} \Gamma$. Therefore, $\models_{\mathscr{J}} A$ and, choosing any x in X, $T_S = (|A|_{\mathscr{J}})_x = |A|_{\mathscr{J}_x} = |A|_{\mathscr{I}}$ and so $\models_{\mathscr{I}} A$. $\square$

3. Models of computation. We now build up the structures necessary to give meaning to programs. These will be power structures, based on certain elementary structures called standard structures.

We define Spec to be the set of special Lucid function symbols {**first, next, as soon as, hitherto, latest, followed by**}.

3.1. Standard structures. An alphabet Σ is *standard* if in addition to T *and* $\exists$ it contains the nullary operation symbols $\perp$ and F, the unary operation symbol $\neg$, the binary operation symbols $\vee$ and $=$ and the ternary operation symbol "if then

else", but none of the special Lucid symbols in Spec. (Σ may contain numerals 0, 1, etc., as nullary operation symbols.)

A *standard structure* is a structure S whose alphabet is standard and such that

(a) T_S, F_S and $\perp_S$ are *true*, *false*, and *undefined*, respectively.

(b) $\neg_S$ yields *true* if its argument is *false*, *false* if its argument is *true*, *undefined* otherwise.

(c) $\vee_S$ yields *true* if at least one argument is *true*, *false* if both are *false*, *undefined* otherwise.

(d) $=_S$ yields *true* if its arguments are identical, *false* otherwise.

(e) if then else$_S$ yields its second argument if its first is *true*, its third if its first is *false*, *undefined* otherwise.

(f) For any subset K of U_S, $\exists_S(K)$ is *true* if *true* $\in K$, *false* if $K = \{$*false*$\}$, *undefined* otherwise.

(g) All other operations of S are monotonic, for the ordering on U_S defined by $x \sqsubseteq y$ iff $x = y$ or $x =$ *undefined*. (Note then that the only nonmonotonic operation is $=_S$.)

Standard structures are our basic domains of data objects and correspond most closely to ordinary first order structures. Note that if we restrict $\neg_S$ and $\vee_S$ to *true*, *false* and *undefined*, they agree with the corresponding operators in the three-valued logic of Lukasiewicz.

3.2. Computation structures. Programs will use the special Lucid functions Spec, and these functions are interpreted over certain types of power structures.

3.2.1. Comp(S). If S is a standard Σ-structure, then Comp(S) is the unique ($\Sigma \cup$ Spec)-structure $\mathscr{C}$ which extends[1] $S^{\mathcal{N}^1}$ to the larger alphabet as follows:

For $\alpha, \beta \in U_{\mathscr{C}}$ and $\bar{t} = t_0 t_1 t_2 \cdots \in \mathcal{N}^{\mathcal{N}}$,

(i) $(\mathbf{first}_{\mathscr{C}}(\alpha))_{\bar{t}} = \alpha_{0 t_1 t_2 \cdots}$,

(ii) $(\mathbf{next}_{\mathscr{C}}(\alpha))_{\bar{t}} = \alpha_{t_0+1 t_1 t_2 \cdots}$,

(iii) $(\alpha \ \mathbf{as\ soon\ as}_{\mathscr{C}}\ \beta)_{\bar{t}} = \alpha_{s r_1 t_2 \ldots}$ if there is a unique s such that $\beta_{s t_1 t_2 \ldots}$ is *true* and $\beta_{r t_1 t_2 \ldots}$ is *false* for all $r < s$, *undefined* if no such s exists,

(iv) $(\mathbf{hitherto}_{\mathscr{C}}(\alpha))_{\bar{t}} =$ *true* if $\alpha_{s t_1 t_2 \ldots}$ is *true* for all $s < t_0$, *false* if $\alpha_{s t_1 t_2 \ldots}$ is *false* for some $s < t_0$, *undefined* otherwise,

(v) $(\mathbf{latest}_{\mathscr{C}}(\alpha)_{\bar{t}} = \alpha_{t_1 t_2 \cdots}$,

(vi) $(\alpha \ \mathbf{followed\ by}_{\mathscr{C}}\ \beta)_{0 t_1 t_2} = \alpha_{0 t_1 t_2 \cdots}$,

$(\alpha \ \mathbf{followed\ by}_{\mathscr{C}}\ \beta)_{t_0+1 t_1 t_2 \ldots} = \beta_{t_0 t_1 t_2 \ldots}$.

Note that all other operations are pointwise extensions of the corresponding operations in S.

3.2.2. The function **latest** is used to formalize nested loops. If we have no nested loops, a simpler structure suffices.

Loop(S). If Σ and S are as above and Σ' is the alphabet of Comp(S), omitting **latest**, then Loop(S) is the unique Σ'-structure $\mathscr{C}'$ which extends $S^{\mathcal{N}}$ to Σ' in such a way that $\mathbf{first}_{\mathscr{C}}$, $\mathbf{next}_{\mathscr{C}}$, etc., are defined as for Comp(S), but with $t_1 t_2 \cdots$ omitted. For example,

$$(\mathbf{first}_{\mathscr{C}'}(\alpha))_{t_0} = \alpha_{t_0} \quad \text{and} \quad (\mathbf{next}_{\mathscr{C}'}(\alpha))_{t_0} = \alpha_{t_0+1}.$$

[1] $\mathcal{N}$ is the set of natural numbers and $\mathcal{N}^{\mathcal{N}}$ is the set of functions from $\mathcal{N}$ to $\mathcal{N}$, i.e., the set of infinite sequences of natural numbers.

The usefulness of Loop(S) lies in the fact that Loop(S) is easier to understand and Loop(S) and Comp(S) have the same theory for terms not involving **latest**:

THEOREM 1. *For any standard structure S and any term A and set of terms Γ all in the language of* Loop(S),

$$\Gamma \models_{\text{Comp}(S)} A \text{ iff } \Gamma \models_{\text{Loop}(S)} A.$$

Proof. Let $\mathscr{C}'$ be the restriction of Comp(S) to the language of Loop(S). It is easily verified that $\mathscr{C}'$ is isomorphic to $\text{Loop}(S)^{\mathscr{N}^1}$ and so the result follows from Corollary 2.1. □

3.2.3. Note that if S is a standard structure and $\mathscr{C}$ is an extension of S^X for some set X, then $=_{\mathscr{C}}$ is *not* the identity relation on $\mathscr{C}$. Nevertheless, $\models_{\mathscr{C}} A = B$ iff $|A|_{\mathscr{C}}$ and $|B|_{\mathscr{C}}$ are identical.

3.2.4. Quiescence and constancy. Let $\mathscr{C} = \text{Comp}(S)$ and $\alpha \in U_{\mathscr{C}}$. Then α is a function from infinite sequences $t_0t_1t_2 \cdots$ of natural numbers to U_S. If α is the value of a variable V, then, intuitively, the value of V depends on the time parameters $t_0t_1t_2 \cdots$, where t_0 is the number of iterations of the loop defining V, t_1 is the number of iterations of the next outer loop, and so on. If $\alpha_{\bar{t}}$ is independent of the first element of $\bar{t}$ (i.e., $\alpha_{t_0t_1t_2\ldots} = \alpha_{0t_1t_2\ldots}$ for all t_0) then we say α is *quiescent.* A term A is quiescent (in $\mathscr{C}$) if $\models_{\mathscr{C}} A = \textbf{first } A$. Note that for terms A and B, **first** A, **latest** A and A **as soon as** B are all quiescent.

If $\alpha_{\bar{t}}$ is independent of $\bar{t}$, then α is said to be *constant.* Note that $G_{\mathscr{C}}$ is constant for any nullary operation symbol G.

In Loop(S) we can use the same definitions, but then there is no difference between quiescence and constancy.

4. Programs. We impose minimal syntactic restrictions on programs, to simplify the formal treatment. In practice, other restrictions would probably be required.

4.1. Syntax. A Σ-program P is a set of ($\Sigma \cup$ Spec)-terms such that

(a) each element of P is an equation of the form $\phi = \psi$, where ψ is a quantifier-free term having no occurrences of $=$, and ϕ is of the form X, **first** X, **next** X or **latest** X for some variable X.

(b) The variable *input* may not occur on the left-hand side of any equation in P.

(c) Every other variable X occurring in P, when appearing on the left-hand side of an equation, may only do so as part of a definition of X.

X must be defined exactly once, in one of the following ways:

directly, i.e., $X = \psi_1$,

indirectly, i.e., **latest** $X = \psi_2$,

iteratively, i.e., **first** $X = \psi_3$,

next $X = \psi_4$.

In the above, the terms ψ_2 and ψ_3 must be *syntactically quiescent* in P, a property which is defined as follows:

(i) **first** ϕ, **latest** ϕ and ϕ **as soon as** ψ are syntactically quiescent in P.

(ii) If $\phi_1, \phi_2, \cdots, \phi_n$ are syntactically quiescent in P and G is an n-ary operation symbol in Σ, then $G(\phi_1, \phi_2, \cdots, \phi_n)$ is syntactically quiescent in P.

(iii) If $Y = \phi$ is in P and ϕ is syntactically quiescent in P, then Y is syntactically quiescent in P.

4.2. Semantics. The meanings of programs are specified by Comp(S)-interpretations, where S is the standard structure corresponding to the domain of data.

4.2.1. Solutions. For any Σ-program P and standard Σ-structure S, if $\mathscr{C} =$ Comp(S) and α is an element of $U_{\mathscr{C}}$, then a *(S, α)-solution of P* is a $\mathscr{C}$-interpretation $\mathscr{I}$ such that $\text{input}_{\mathscr{I}} = \alpha$ and $\models_{\mathscr{I}} P$.

4.2.2.

THEOREM 2. *For any Σ-program P and standard Σ-structure S, if $\mathscr{C} =$ Comp(S) and $\alpha \in U_{\mathscr{C}}$, then there is a ($S, \alpha$)-solution $\mathscr{I}$ of P that is* minimal, *i.e., for any (S, α)-solution $\mathscr{I}'$ of P, for all $\bar{t} \in \mathscr{N}^{\mathscr{N}}$ and all variables V in Σ, $(V_{\mathscr{I}})_{\bar{t}} \sqsubseteq (V_{\mathscr{I}'})_{\bar{t}}$.*

Proof (sketch). The first step is to transform P into a set of simple equations. This is done by replacing each pair of equations of the form **first** $X = \phi$, **next** $X = \phi'$ by the single simple equation $X = \phi$ **followed by** ϕ', and replacing each equation of the form **latest** $X = \phi$ by the simple equation $X = \textbf{latest}^{-1}\ \phi$. The operation $\textbf{latest}_{\mathscr{C}}^{-1}$ is defined by $(\textbf{latest}_{\mathscr{C}}^{-1}(\alpha))_{t_0 t_1 \ldots} = \alpha_{0 t_0 t_1 \ldots}$.

This transforms the program P into a "program" P' of the form $\bar{X} = \tau(\bar{X})$, where $\bar{X}$ is the vector of all the variables in P other than *input*.

We now note that the "programs" P and P' have the same solutions. That every solution of the original program P is a solution of the transformed program P' is clear, and the converse follows from the quiescence restrictions on P, as follows.

Suppose that $\models_{\mathscr{I}} X = \phi$ **followed by** ϕ'. Then by the definition of **followed by**, $\models_{\mathscr{I}}$ **first** $X =$ **first** ϕ and $\models_{\mathscr{I}}$ **next** $X = \phi'$. But if $X = \phi$ **followed by** ϕ' in P' came from **first** $X = \phi$ and **next** $X = \phi'$ in P, then the syntactic quiescence of ϕ ensures that $\models_{\mathscr{I}} \phi =$ **first** ϕ, so $\models_{\mathscr{I}}$ **first** $X = \phi$. Similarly, $\models_{\mathscr{I}} X =$ **latest**$^{-1}\ \phi$ implies $\models_{\mathscr{I}}$ **latest** $X =$ **first** ϕ, and so by syntactic quiescence $\models_{\mathscr{I}}$ **latest** $X = \phi$.

Now we note that the ordering on $U_{\mathscr{C}}$ given in the statement of the theorem makes $U_{\mathscr{C}}$ into a cpo (complete partial order), and it is easily verified that all the operations in $\mathscr{C}$ that are used in the "term" τ are continuous. Moreover, although $=_{\mathscr{C}}$ is not equality on U_C, by § 3.2.3 the solutions of P' are fixpoints of τ.

Therefore, the transformed program P' has a unique minimal (S, α)-solution $\mathscr{I}$, and hence so does P. (In fact $\bar{X}_{\mathscr{I}} = \bigsqcup_{i=0}^{\infty} |\tau^i(\bar{\perp})|_{\mathscr{I}}$; see, e.g., [5].) □

4.3. Syntactic enrichment. To facilitate the writing of programs we introduce "nesting" in programs, as a syntactic abbreviation. We say that the expression

begin
ϕ_1
ϕ_2
⋮
ϕ_n
end

is shorthand for the set of terms

$$\phi_1'$$
$$\phi_2'$$
$$\vdots$$
$$\phi_n',$$

where ϕ_i' is obtained from ϕ_i by replacing each "global" variable V by **latest** V. A global variable is one which occurs within the rest of the program enclosing the original *begin ... end* expression. The symbols *begin* and *end* are used to delimit inner loops, and the formulation using **latest** shows that within inner loops global variables become quiescent. Loops can be nested to any depth. Note that for a program using *begin ... end* to be legal, the result of removing the *begin ... end*'s must be a legal program according to § 4.1.

5. Axioms. We now describe the formal system of axioms and rules of inference used for proving properties of Lucid programs.

5.1. The following abbreviations will be used in the rest of the paper:

$$A \wedge B \quad \text{means} \quad \neg(\neg A \vee \neg B),$$
$$A \to B \quad \text{means} \quad \neg(A = \mathrm{T}) \vee B,$$
$$\forall V A \quad \text{means} \quad \neg \exists V \neg A.$$

Note that, in standard structures, $\wedge$ agrees with the three-valued logic of Lukasiewicz, but $\to$ does not. In particular, we have $\models \bot \to \mathrm{F}$, but in standard three-valued logic $\bot \to F$ would be $\bot$. This difference is crucial, and allows, for example, the use of the deduction theorem in standard structures. (However, $\to$ is defined in terms of $=$, and therefore it may not be used in programs.)

5.2. Parentheses will be (and have been) dropped from terms by using the following ranking of priorities for operators (from highest to lowest):

first, next, latest, hitherto, $\neg$, $\wedge$, $\vee$, if then else, **as soon as**,

followed by, $=$, $\to$.

Note the low priority of **as soon as**, and the even lower priorities of $=$ and $\to$. Thus $A \to B = C$ **as soon as** $D \wedge E$ means $A \to (B = (C$ **as soon as** $(D \wedge E)))$.

5.3.

THEOREM 3. *The following are valid in* Comp(S) *for any standard* Σ*-structure* S, *and* ($\Sigma \cup$ Spec)*-terms* X, Y *and* P:

(a) $(X = Y) \vee \neg(X = Y)$,
(b) $((A = \mathrm{T}) = (\neg\neg A = \mathrm{T})) \wedge ((A = \mathrm{F}) = (\neg A = \mathrm{T}))$,
(c) (**first first** X = **first** X) $\wedge$ (**next first** X = **first** X),
(d) (**first**(X **followed by** Y) = **first** X) $\wedge$ (**next**(X **followed by** Y) = Y),
(e) (**first hitherto** P = T) $\wedge$ (**next hitherto** $P = P \wedge$ **hitherto** P),
(f) X **as soon as** P = if **first** P then **first** X else (**next** X **as soon as next** P),
(g) X **as soon as** $P = X$ **as soon as** $P \wedge$ **hitherto** $\neg P$,
(h) **first**(X **as soon as** P) = X **as soon as** P,
(i) $P \wedge$ **hitherto** $\neg P \to X$ **as soon as** $P = X$,

(j) T **as soon as** $P \rightarrow$ **first** X **as soon as** $P =$ **first** X,

(k) (if T then X else $Y = X$) $\wedge$ (if F then X else $Y = Y$).

Proof. These results (for *variables* X, Y and P) are easily verified in Loop(S) and carry over to Comp(S) by Theorem 1. The variables can then be replaced by ($\Sigma \cup$ Spec)-terms. □

If we define **eventually** P to be T **as soon as** P (with the same priority as **as soon as**), we have the following corollary.

COROLLARY 3.1. *With S, X and P as above, the following are valid in* Comp(S):

(a) **eventually** $P \rightarrow$ **first** X **as soon as** $P =$ **first** X,

(b) **eventually** $P =$ **eventually** $P \wedge$ **hitherto** $\neg P$,

(c) **eventually** $P =$ if **first** P then T else **eventually next** P.

Proof. These follow from axioms of Theorem 3. □

5.4. The next theorem justifies "pushing" **first** and **next** past quantifiers and non-Lucid operations.

THEOREM 4. *For any standard Σ-structure S and any Σ-term A in which $X_1, X_2, \cdots, X_k$ are the variables occurring freely*:

(a) **first** $A = A(X_1/$**first** $X_1, X_2/$**first** $X_2, \cdots)$ *is valid in* Comp(S), *along with corresponding equations for* **next** *and* **latest**.

(b) **eventually** $P \rightarrow A$ **as soon as** $P = A(X_1/X_1$ **as soon as** $P, X_2/X_2$ **as soon as** $P, \cdots)$ *is valid in* Comp(S).

Proof. We will consider only Loop(S). The results carry over to Comp(S) by Theorem 1. Let $\mathcal{I}$ be a Loop(S)-interpretation and let t be any natural number. Then, if $\bar{X}$ denotes $X_1, X_2, \cdots, X_k$,

$$(|\textbf{first } A|_{\mathcal{I}})_t = (|A|_{\mathcal{I}})_0$$

$$= |A|_{\mathcal{I}_0} \quad \text{(by Lemma 2)} \quad = |A|_{\mathcal{I}_0(\bar{X}/(\bar{X}_{\mathcal{I}})_0)}$$

$$= |A|_{\mathcal{I}_0(\bar{X}/(|\textbf{first } \bar{X}|_{\mathcal{I}})_t)} = |A|_{\mathcal{I}_t(\bar{X}/(|\textbf{first } \bar{X}|_{\mathcal{I}})_t)}$$

(since A has no other free variables)

$$= (|A|_{\mathcal{I}(\bar{X}/|\textbf{first } \bar{X}|_{\mathcal{I}})})_t \quad \text{(by Lemma 2)} \quad = (|A(\bar{X}/\textbf{first } \bar{X})|_{\mathcal{I}})_t$$

(by Lemma 1).

Similar reasoning verifies the other results. □

6. Rules of inference. Lucid cannot be a *complete* formal system because the Lucid functions are powerful enough to characterize unsolvable problems that are not even partially decidable. All we can do is add to Lucid whatever axioms and rules of inference seem natural and useful. In this section we give rules of inference for the logical connectives, and useful rules for the special Lucid functions. The "logical" rules of inference are those of a simple natural deduction system (see, for example [4]).

6.1. Natural deduction rules.

6.1.1.

THEOREM 5. *The following rules are valid for standard Σ-structure S, Σ-terms A, B, C, D, P, Q, finite sets Γ and Δ of Σ-terms and variable V, provided V does not*

occur freely in Γ *or* D, *and is free for* P *and* Q *in* A:

$(\wedge I)$	$A, B \models_S A \wedge B$	$(\wedge E)$	$A \wedge B \models_S A$ $A \wedge B \models_S B$
$(\vee I)$	$A \models_S A \vee B$ $B \models_S A \vee B$	$(\vee E)$	$A \to C, B \to C, A \vee B \models_S C$
(FI)	$A, \neg A \models_s \mathrm{F}$	(FE)	$\mathrm{F} \models_S B$
$(\to I)$	*if* $\Delta, A \models_S B$ *then* $\Delta \models_S A \to B$	$(\to E)$	$A \to B, A \models_S B$
$(\forall I)$	*if* $\Gamma \models_S A$ *then* $\Gamma \models_S \forall VA$	$(\forall E)$	$\forall VA \models_S A(V/Q)$
$(\exists I)$	$A(V/Q) \models_S \exists VA$	$(\exists E)$	*if* $\Gamma \models_S A \to D$ *then* $\Gamma, \exists VA \models_S D$
$(=I)$	$\models_S V = V$	$(=E)$	$A(V/P), P = Q \models_S A(V/Q)$.
(TI)	$A \models_S A = \mathrm{T}$	(TE)	$A = \mathrm{T} \models_S A$

Proof. The validity of the rules can be established by straightforward calculation from the definitions. □

There are no rules for $\neg$ because we do not have the law of the excluded middle: $A \vee \neg A$ is not valid in general, because A may not be truth-valued. This means that some of the tautologies and derived rules of first order logic are not valid in standard structures. For example $(A \to B) \to \neg A \vee B$ is not valid, and if we were to define $A \leftrightarrow B$ to mean $(A \to B) \wedge (B \to A)$, then we would not have substitutivity of $\leftrightarrow$ (note, for example, that $\perp \leftrightarrow \mathrm{F}$).

6.1.2. Most of the rules of Theorem 5 hold also for Comp(S):

THEOREM 6. *All the rules of Theorem* 5, *except* $(\to I)$, *are valid for* $\mathscr{C}$ *in place of* S (*where* $\mathscr{C}$ *is* Comp(S)), *and* $\Sigma \cup \mathrm{Spec}$ *in place of* Σ.

Proof. Apart from the quantifier rules, and $(\to I)$, all rules are of the form $\phi \models \psi$ and carry over directly because of the pointwise definition of the connectives. We illustrate this for the $(\vee E)$ rule. Consider any $\mathscr{C}$-interpretation $\mathscr{I}$ for which $\models_{\mathscr{I}} A \to C$, $\models_{\mathscr{I}} B \to C$ and $\models_{\mathscr{I}} A \vee B$. Then, for all $\bar{t} \in \mathscr{N}^{\mathscr{N}}$, $(|A \to C|_{\mathscr{I}})_{\bar{t}}$, $(|B \to C|_{\mathscr{I}})_{\bar{t}}$ and $(|A \vee B|_{\mathscr{I}})_{\bar{t}}$ are all *true*. By definition of $\mathscr{C}$, we then have $(|A|_{\mathscr{I}})_{\bar{t}} \to_S (|C|_{\mathscr{I}})_{\bar{t}}$, $(|B|_{\mathscr{I}})_{\bar{t}} \to_S (|C|_{\mathscr{I}})_{\bar{t}}$ and $(|A|_{\mathscr{I}})_{\bar{t}} \vee_S (|B|_{\mathscr{I}})_{\bar{t}}$ are all *true*. By the $(\vee E)$ rule for S (Theorem 1) we then have $(|C|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$. This holds for all $\bar{t} \in \mathscr{N}^{\mathscr{N}}$, so $\models_{\mathscr{I}} C$.

We illustrate the proof for the quantifier rules by considering $(\forall E)$ and $(\exists E)$.

Rule $(\forall E)$. Let $\mathscr{I}$ be any $\mathscr{C}$-interpretation for which $|\forall VA|_{\mathscr{I}} = \mathrm{T}_{\mathscr{I}}$. Then for all $\bar{t} \in \mathscr{N}^{\mathscr{N}}$

$$\mathit{true} = (|\forall\, VA|_{\mathscr{I}})_{\bar{t}}$$

$$= \forall_S\{(|A|_{\mathscr{I}(V/\alpha)})_{\bar{t}} : \alpha \in U_{\mathscr{C}}\}.$$

Therefore for all $\bar{t} \in \mathscr{N}^{\mathscr{N}}$ and all $\alpha \in U_{\mathscr{C}}$ we have $(|A|_{\mathscr{I}(V/\alpha)})_{\bar{t}} = \mathit{true}$. Now $|A|_{\mathscr{I}(V/|Q|_{\mathscr{I}})} = |A(V/Q)|_{\mathscr{I}}$, by Lemma 1, and so, for all $\bar{t} \in \mathscr{N}^{\mathscr{N}}$, $(|A(V/Q)|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$, that is $\models_{\mathscr{I}} A(V/Q)$.

Rule $(\exists E)$. Assume $\Gamma \models_{\mathscr{C}} A \to D$ and consider any $\mathscr{C}$-interpretation $\mathscr{I}$ for which $\models_{\mathscr{I}} B$, for all $B \in \Gamma$, and $\models_{\mathscr{I}} \exists VA$. Consider any $\bar{t} \in \mathscr{N}^{\mathscr{N}}$. By the definition of $\exists_{\mathscr{C}}$, there is some $\alpha \in U_{\mathscr{C}}$ such that $(|A|_{\mathscr{I}(V/\alpha)})_{\bar{t}} = \mathit{true}$. Now $\mathscr{I}(V/\alpha)$ is a $\mathscr{C}$-interpretation and $\models_{\mathscr{I}(V/\alpha)} \Gamma$, since V is not free in Γ. Thus $\models_{\mathscr{I}(V/\alpha)} A \to D$, and so $(|D|_{\mathscr{I}(V/\alpha)})_{\bar{t}} = \mathit{true}$. Since V is not free in D, we then have $(|D|_{\mathscr{I}})_{\bar{t}}$. We choose $\bar{t}$ arbitrarily, so $\models_{\mathscr{I}} D$. □

6.2. Lucid rules.

6.2.1. One of the most important rules is that a standard Σ-structure S and Comp(S) have the same theory, when restricted to Σ-terms, so any "elementary" property can be used directly in any proof about a program.

THEOREM 7. *For any standard Σ-structure S, any Σ-term A and any set Γ of Σ-terms, $\Gamma \models_S A$ iff $\Gamma \models_{\mathscr{C}} A$, where $\mathscr{C} = \text{Comp}(S)$.*

Proof. Since Γ and A are in the language of S and since Comp(S) is an extension of $S^{\mathcal{N}_1}$ the result follows immediately from Corollary 2.1. □

6.2.2. Other Lucid rules including induction and termination are given by the following theorem.

THEOREM 8. *For any standard Σ-structure S, if $\mathscr{C} = \text{Comp}(S)$, then*

(a) $P \models_{\mathscr{C}}$ **first** P *and* $P \models_{\mathscr{C}}$ **next** P,

(b) **first** P, $P \to$ **next** $P \models_{\mathscr{C}} P$ (Induction),

(c) $P \wedge$ **hitherto** $\neg P \to$ **first** Q, **eventually** $P \models_{\mathscr{C}}$ **first** Q,

(d) $P \to \neg$**hitherto** $(P = \text{F}) \models_{\mathscr{C}} X$ **as soon as** $P = \bot$,

(e) *integer* Y, $Y >$ **next** $Y \models_{\mathscr{C}}$ **eventually** $Y \leqq 0$ (Termination),

(f) $X =$ **next** $X \models_{\mathscr{C}} X =$ **first** X,

where in (e) *we assume S includes the integers.*

Proof. By calculation from the definitions. □

6.3. Recovering the deduction theorem. We have seen that the $(\to I)$ rule is not valid in Comp(S). However, we can recover this rule, at the expense of weakening the $(=E)$ rule, by a form of reasoning which intuitively corresponds to confining oneself to a particular moment during the execution of a program.

6.3.1. Definition of $\approx$. If S is a Σ-structure, $\mathscr{C} = \text{Comp}(S)$ and A is a term and Γ a set of terms on the alphabet of $\mathscr{C}$, then we define $\Gamma \approx_{\mathscr{C}} A$ to mean that for any $\mathscr{C}$-interpretation $\mathscr{I}$, if $(|B|_{\mathscr{I}})_{\bar{t}} =$ *true* for every B in Γ, then $(|A|_{\mathscr{I}})_{\bar{t}} =$ *true*.

Thus $\Gamma \approx_{\mathscr{C}} A$ means that, at any time, if all the terms in Γ are *true*, then A is *true*. It is immediate that $\models A$ implies $\approx A$, and that $\Gamma \approx_{\mathscr{C}} A$ implies $\Gamma \models_{\mathscr{C}} A$ but not vice versa; e.g., $P \models_{\mathscr{C}}$ **next** P but not $P \approx_{\mathscr{C}}$ **next** P.

6.3.2.

THEOREM 9. *For any standard Σ-structure S, if $\mathscr{C} = \text{Comp}(S)$*

(a) *every rule of Theorem 5 except the $(=E)$ rule remains valid if $\models_S$ is replaced by $\approx_{\mathscr{C}}$,*

(b) *for A, P, Q and V as in Theorem 5, if A contains no Lucid functions, then $A(V/P), P = Q \approx_{\mathscr{C}} A(V/Q)$,*

(c) *Theorem 7 is valid for $\approx_{\mathscr{C}}$ in place of $\models_{\mathscr{C}}$.*

Proof. Let Δ, A, D and V be as in Theorem 5.

(a) We will illustrate the proof by considering the $(\exists E)$ rule and the $(\to I)$ rule.

(i) Assume $\Gamma \approx_{\mathscr{C}} A \to D$ and $\Gamma \approx_{\mathscr{C}} \exists VA$. Let $\bar{t} \in \mathcal{N}^{\mathcal{N}}$ and let $\mathscr{I}$ be a $\mathscr{C}$-interpretation such that $(|B|_{\mathscr{I}})_{\bar{t}}$ for every B in Γ. Then by the second assumption $(|\exists VA|_{\mathscr{I}})_{\bar{t}} =$ *true* and so $(|A|_{\mathscr{I}(V/\alpha)})_{\bar{t}} =$ *true* for some α in $U_{\mathscr{C}}$ by the definition of $\exists_{\mathscr{C}}$. Since V does not occur in any B in Γ, $(|B|_{\mathscr{I}(V/\alpha)})_{\bar{t}} = (|B|_{\mathscr{I}})_{\bar{t}} =$ *true* for any such B, and so by the first assumption $(|D|_{\mathscr{I}})_{\bar{t}} =$ *true*. Therefore $\Gamma \approx_{\mathscr{C}} D$.

(ii) Let $\bar{t} \in \mathcal{N}^{\mathcal{N}}$ and suppose that every $\mathscr{C}$-interpretation which makes A and

everything in Γ *true* at $\bar{t}$ also makes B *true* at $\bar{t}$. Suppose now that $\mathscr{C}$-interpretation $\mathscr{I}$ makes everything in Γ *true* at $\bar{t}$. If $\mathscr{I}$ makes A *true* at $\bar{t}$, then it must make B *true* at $\bar{t}$ and so makes $A \to B$ *true* at $\bar{t}$. On the other hand, if $\mathscr{I}$ makes A other than *true* at $\bar{t}$, then $A \to B$ will be *true* at $\bar{t}$ regardless of the value $\mathscr{I}$ assigns B at $\bar{t}$. In either case $A \to B$ is *true* at $\bar{t}$ and so $\Gamma \models_{\mathscr{C}} A \to B$.

(b) Suppose that $(|A(V/P)|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$ and $(|P = Q|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$. Now $|A(V/P)|_{\mathscr{I}} = |A|_{\mathscr{I}(V/|P|_{\mathscr{I}})}$ by Lemma 1 and $(|A|_{\mathscr{I}(V/|P|_{\mathscr{I}})})_{\bar{t}} = |A|_{\mathscr{I}_{\bar{t}}(V/(|P|_{\mathscr{I}})_{\bar{t}})}$ since A contains no Lucid functions. But $(|P = Q|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$ implies $(|P|_{\mathscr{I}})_{\bar{t}} = (|Q|_{\mathscr{I}})_{\bar{t}}$. Thus

$$\begin{aligned} |A|_{\mathscr{I}_{\bar{t}}(V/(|P|_{\mathscr{I}})_{\bar{t}})} &= |A|_{\mathscr{I}_{\bar{t}}(V/(|Q|_{\mathscr{I}})_{\bar{t}})} \\ &= (|A|_{\mathscr{I}(V/|Q|_{\mathscr{I}})})_{\bar{t}} \\ &= (|A(V/Q)|_{\mathscr{I}})_{\bar{t}}. \end{aligned}$$

Therefore $(|A(V/Q)|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$.

(c) Assume $\Gamma \models_S A$ and let $\mathscr{I}$ be a $\mathscr{C}$-interpretation. For any $\bar{t} \in \mathscr{N}^{\mathscr{N}}$, if $(|B|_{\mathscr{I}})_{\bar{t}}$ is *true* for all B in Γ, then by Lemma 2, $|B|_{\mathscr{I}_{\bar{t}}}$ is *true* for all B in Γ; i.e., $\models_{\mathscr{I}_{\bar{t}}} \Gamma$. Hence $\models_{\mathscr{I}_{\bar{t}}} A$, and so $(|A|_{\mathscr{I}})_{\bar{t}}$. Conversely, assume $\Gamma \models_{\mathscr{C}} A$. Therefore, $\Gamma \models_{\mathscr{C}} A$, and hence $\Gamma \models_S A$, by Theorem 7. □

We call the rule in Theorem 9(b) the (weak $= E$) rule. To illustrate that $(=E)$ does not work for $\models$, note that **next** P, $P = Q \models$ **next** Q is not valid (informally, if P equals Q at some time when P will be *true* at the next step, it does not necessarily follow that Q will be *true* at the next step).

We use $\models$ in the following way. Suppose we wish to prove $\Gamma \models_{\mathscr{C}} A \to B$. We assume Γ and A, and try to prove B using only axioms and Theorem 7 and the natural deduction rules of Theorem 5, but with the (weak $= E$) rule instead of the $(=E)$ rule. (We may not use Theorem 8.) If we manage to do this we have $\Gamma, A \models_{\mathscr{C}} B$ and we can use $(\to I)$ to get $\Gamma \models_{\mathscr{C}} A \to B$. Thus $\Gamma \models_{\mathscr{C}} A \to B$. We see that to use the deduction theorem we must not use any of the Lucid rules in Theorem 8, and use only the weak version of the $(=E)$ rule.

6.3.3. There is another way in which we can regain the deduction theorem. If we are reasoning about a simple loop, and we have made an assumption that is quiescent, then the assumption can be canceled:

THEOREM 10. *For any Σ-structure S, if $\mathscr{C} = \mathrm{Comp}(S)$ and A and B are terms and Γ a set of terms on the alphabet of $\mathscr{C}$ omitting* **latest**, *then*

$$\Gamma, \textbf{first}\ A \models_{\mathscr{C}} B \textit{ implies } \Gamma \models_{\mathscr{C}} \textbf{first}\ A \to B.$$

Proof. The theorem holds for Loop(S) in place of Comp(S), because if **first** A is ever true, it is always true. The result carries over to Comp(S) by Theorem 1. □

7. Proofs within loops. The structuring of programs that is made possible by the use of *begin* and *end* also allows "structured proofs". We will show that

(i) Within a *begin* ... *end* loop, all the rules of inference are valid and so is the assumption that $X =$ **first** X for every global variable X. Anything that follows by introducing **latest** also follows without **latest**, in this fashion.

(ii) Any assertion about the globals of a *begin* ... *end* loop, that does not use Lucid functions, can be moved into or out of the loop.

THEOREM 11. *For any standard* Σ*-structure* S*, if* $\mathscr{C} = \mathrm{Comp}(S)$*, then for any term* A *and set of terms* Γ *in the alphabet of* $\mathscr{C}$*, and any finite set of variables* $\bar{X}$*,*

(a) $\bar{X} = \textbf{first}\ \bar{X}, \Gamma \models_{\mathscr{C}} A$ *iff*

$$\Gamma(\bar{X}/\textbf{latest}\ \bar{X}) \models_{\mathscr{C}} A(\bar{X}/\textbf{latest}\ \bar{X}).$$

(b) *If* A *is a* Σ*-term and* $\bar{X}$ *is the set of variables occurring freely in* A*, then*

$$\Gamma \models_{\mathscr{C}} A \text{ iff } \Gamma \models_{\mathscr{C}} A(\bar{X}/\textbf{latest}\ \bar{X}).$$

Proof. (a) Assume $\bar{X} = \textbf{first}\ \bar{X}$, $\Gamma \models_{\mathscr{C}} A$, and that, for $\mathscr{C}$-interpretation $\mathscr{I}$, $\models_{\mathscr{I}} \Gamma(\bar{X}/\textbf{latest}\ \bar{X})$. Let $\bar{\alpha}$ be $|\textbf{latest}\ \bar{X}|_{\mathscr{I}}$ and $\mathscr{I}' = \mathscr{I}(\bar{X}/\bar{\alpha})$. Then $\models_{\mathscr{I}'} \bar{X} = \textbf{first}\ \bar{X}$ and $\models_{\mathscr{I}'} \Gamma$; therefore $\models_{\mathscr{I}'} A$, and so $\models_{\mathscr{I}} A(\bar{X}/\textbf{latest}\ \bar{X})$.

Conversely, assume that $\Gamma(\bar{X}/\textbf{latest}\ \bar{X}) \models_{\mathscr{C}} A(\bar{X}/\textbf{latest}\ \bar{X})$ and that for $\mathscr{C}$-interpretation $\mathscr{I}$, $\models_{\mathscr{I}} \bar{X} = \textbf{first}\ \bar{X}$ and $\models_{\mathscr{I}} \Gamma$. Then $|\bar{X}|_{\mathscr{I}}$ is $\textbf{latest}_{\mathscr{C}}\ \bar{\alpha}$ for some $\bar{\alpha}$ in $U_{\mathscr{C}}$.[2] Let $\mathscr{I}'$ be $\mathscr{I}(\bar{X}/\bar{\alpha})$. Then $\models_{\mathscr{I}'} \Gamma(\bar{X}/\textbf{latest}\ \bar{X})$, and so $\models_{\mathscr{I}'} A(\bar{X}/\textbf{latest}\ \bar{X})$. Hence $\models_{\mathscr{I}} A$.

(b) Let $\mathscr{I}$ be a $\mathscr{C}$-interpretation such that $\models_{\mathscr{I}} \Gamma$. Then since $(|\textbf{latest}\ A|_{\mathscr{I}})_{t_0 t_1 t_2 \ldots} = (|A|_{\mathscr{I}})_{t_1 t_2 \ldots}$, $(|\textbf{latest}\ A|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$ for all $\bar{t}$ iff $(|A|_{\mathscr{I}})_{\bar{t}} = \mathit{true}$ for all $\bar{t}$. Then since $\models_{\mathscr{I}} \textbf{latest}\ A = A(\bar{X}/\textbf{latest}\ \bar{X})$ by Theorem 4(a), the result follows. □

The theorem justifies (i) and (ii) of § 7 as follows. Consider the program Prime again.

$$\theta \begin{cases} N = \textbf{first}\ \mathit{input} \\ \textbf{first}\ I = 2 \\ \textbf{next}\ I = I + 1 \\ \mathit{output} = \neg \mathit{IdivN}\ \textbf{as soon as}\ \mathit{IdivN} \vee I \times I \geqq N \end{cases}$$

begin

$$\Gamma \begin{cases} \textbf{first}\ \mathit{multiple} = I \times I \\ \textbf{next}\ \mathit{multiple} = \mathit{multiple} + I \\ \textbf{first}\ J = I \\ \textbf{next}\ J = J + 1 \\ \mathit{IdivN} = \mathit{multiple}\ \mathrm{eq}\ N\ \textbf{as soon as}\ \mathit{multiple} \geqq N \end{cases}$$

end

Prime is actually equivalent to Prime′:

$$\theta \begin{cases} N = \textbf{first}\ \mathit{input} \\ \textbf{first}\ I = 2 \\ \textbf{next}\ I = I + 1 \\ \mathit{output} = \mathit{IdivN}\ \textbf{as soon as}\ \mathit{IdivN} \vee I \times I \geqq N \end{cases}$$

[2] In fact $\bar{\alpha} = |\textbf{latest}^{-1}\ \bar{X}|_{\mathscr{I}}$ (*see the proof of Theorem* 2, § 4.2.2).

$$\Gamma' \begin{cases} \textbf{first } multiple = \textbf{latest } I \times \textbf{latest } I \\ \textbf{next } multiple = multiple + \textbf{latest } I \\ \textbf{first } J = \textbf{latest } I \\ \textbf{next } J = J + 1 \\ IdivN = multiple \text{ eq } \textbf{latest } N \textbf{ as soon as } multiple \geqq \textbf{latest } N \end{cases}$$

For program Prime′ it is possible to prove that

$$IdivN = \exists K\ 2 \leqq K < N \wedge I \times K = N.$$

In the Introduction, a "nested" proof of this, using Prime, proceeded by the following steps. First we proved, inside the inner loop, that $multiple = I \times J$. Then, still inside the loop, we used this to prove that $IdivN = \exists K\ 2 \leqq K < N \wedge I \times K = N$. For this we needed that $I \geqq 0$. This had to be proved in the outer loop, and could then be brought inside the inner loop, for use in the proof, because it is a statement not involving Lucid functions. Finally, the statement $IdivN = \exists K\ 2 \leqq K < N \wedge I \times K = N$ could be brought out of the inner loop because it does not use Lucid functions, and its free variables are all globals of the inner loop.

Formally we have $\theta \models I \geqq 0$ and $\Gamma,\ I \geqq 0,\ I = \textbf{first } I,\ N = \textbf{first } N \models IdivN = \exists K\ 2 \leqq K < N \wedge I \times K = N$. From these we establish $\theta,\ \Gamma' \models \exists K\ 2 \leqq K < N \wedge I \times K = N$ as follows:

By Theorem 11(a) we have

$$\Gamma' \textbf{ latest } I \geqq 0 \models \textbf{latest } IdivN = \exists K\ 2 \leqq K < \textbf{latest } N \wedge \textbf{latest } I \times K = \textbf{latest } N.$$

But $\theta \models I \geqq 0$, and so by Theorem 11(b) we have $\theta \models \textbf{latest } I \geqq 0$. Therefore,

$$\theta, \Gamma' \models \textbf{latest } IdivN = \exists K\ 2 \leqq K < \textbf{latest } N \wedge \textbf{latest } I \times K = \textbf{latest } N.$$

Again using Theorem 11(b) we get

$$\theta, \Gamma' \models IdivN = \exists K\ 2 \leqq K < N \wedge I \times K = N.$$

This same sort of reasoning can be extended for loops nested to arbitrary depth.

REFERENCES

[1] E. A. ASHCROFT AND W. W. WADGE, *Lucid, a non-procedural language with iteration*, Comm. ACM, 1976, to appear.

[2] ———, *Program proving without tears*, Proc. Intl. Symp. on Proving and Improving Programs, Arc et Senans, IRIA, Domain de Voluceau, France, 1975.

[3] R. BURSTALL, *Program proving as hand simulation with a little induction*, Proc. IFIP Congress 1974, Stockholm, North-Holland, Amsterdam.

[4] Z. MANNA, *Introduction to Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[5] R. MILNER, *Models of LCF*, Memo AIM/CS 332, Stanford Univ., Stanford, Calif., 1973.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

PROOF THEORY OF PARTIAL CORRECTNESS VERIFICATION SYSTEMS*

SUSAN L. GERHART†

Abstract. The verification rules proposed by Hoare are an example of a system which can serve as the basis for a mathematical theory of partial correctness of programs. The purpose of this paper is to extend the previously developed basic theory by

(i) defining alternative verification systems and comparing them with the Hoare rules,

(ii) deriving several types of useful rules from the basic systems,

(iii) showing an ordering on verification systems,

(iv) discussing how semi-interpreted program schemas play the role of theorems in a more fully developed theory,

(v) formulating a notion of correctness-preserving program transformations and giving a procedure for their use.

These extensions provide a more flexible and efficient methodology for proving partial correctness of programs and point to the potential of a mathematical theory which effectively organizes knowledge about programs.

Key words. program correctness, program schemas, program transformations, reasoning about programs, systematic program construction

1. Introduction. Partial correctness is defined to be the property that a given program satisfies a given set of specifications in the form of an input-output relation, if it halts. Total correctness adds to partial correctness the requirement of proper termination whenever the program is executed for data satisfying the input specification. In the inductive assertion method, the most common way of proving partial correctness, a global proof is broken into a collection of local proofs by inserting appropriate assertions into the program.

When first proposed by Floyd [8] and King [19], the method used flow chart representations of programs. An algorithm was applied to a flow chart to generate *verification conditions*, or lemmas, which were logical statements to be proved in a mathematical theory which embodied the semantics of the expressions of the assertion and programming languages. Basically, verification condition generation expressed how the semantics of statements controlling sequencing and changes to variables interacted with assertions.

In a series of papers [13], [14], [15], Hoare defined and illustrated a variation of the inductive assertion method where the algorithm for generating verification lemmas was replaced by a system of axioms and rules of inference. The global proof of correctness is a deduction within the system from lemmas which are similar to the logical statements obtained by verification condition generation. Hoare's formulation is attractive because it has the standard form of a mathematical system and because it works on program texts. The text form of programs is presently preferred because it is a direct representation of programs in modern programming languages and provides for easier restriction of control flow, which has practical effects on the ease of construction of assertions.

This paper will present two forms of mathematical systems which correspond to the verification condition generation version of the inductive assertion method,

* Received by the editors April 21, 1975, and in final revised form March 12, 1976.

† Department of Computer Science, Duke University, Durham, North Carolina 27706. This research was supported in part by the National Science Foundation under Grant MC5 75-08146.

which will be called *verification operator systems* in contrast to *verification rule systems.* A well known property, which is usually stated and used informally, is that, when properly defined, there are equivalent forward and backward directed methods of constructing verification conditions. This result will be formally proved and verification operator systems will be proved consistent with rule systems. All these results hold for a simple programming language.

Thus, it will be formally shown that there are several formulations of the inductive assertion method which are the basis for a mathematical theory of partial correctness of programs. However, a useful mathematical theory should be developed beyond well-defined systems. It should have derived rules which serve to reduce proof length by condensing common deductive chains. There should be theorems which hold for classes of the mathematical objects of the theory so that both more insight into the theory can be gained and properties of individual objects can be proved more easily as instances of the theorems. A theory often has a useful replacement theorem which permits certain types of replacements within a known valid statement to obtain another statement known to be valid without proof.

We will show that in our theory the role of general theorems is played by semi-interpreted program schemas and that the notion of correctness-preserving program transformations is supported by a replacement theorem. Also, the inherent characteristic of the inductive assertion method that information must be added to the program leads to a notion of ordering of verification systems where degrees of completeness of assertions is expressed. Finally, some examples will bring together these results to illustrate a new methodology for proving partial correctness where it is possible to build upon previously proved results.

This paper uses a very simple programming language with unspecified expression and assertion languages. In fact, every verification system and first order theory must define the semantics of a specific language and thus there will be one theory for each programming language. Of course, no such theory is acceptable unless it has soundness and completeness properties, in appropriate senses. This topic is treated in other papers; e.g., see de Bakker and Meertens [3] and Cook [2]. The proof theory results of the present paper shoud remain substantially the same for all programming language theories.

2. Verification systems. We will use a very simple programming language to illustrate verification systems. The syntax is

⟨program⟩ ::= ⟨statement − list⟩
⟨statement-list⟩ ::= ⟨statement⟩|⟨statement⟩; ⟨statement-list⟩
⟨statement⟩ ::= **null**|⟨variable⟩ := ⟨expression⟩
|**if** ⟨boolean-expression⟩ **then** ⟨statement-list⟩
else ⟨statement-list⟩ **fi**
|**while** ⟨boolean-expression⟩ **do**
⟨statement-list⟩ **end**

⟨boolean-expression⟩ and ⟨expression⟩ are not further prescribed here, but it is assumed that ⟨boolean-expression⟩ is always truth-valued and expressions have no side effects. We will consistently use the following notation for syntactic units:

G or S for program or statement list; V for variable; E for expression; B for boolean-expression; P, Q, A, for assertion.

An example of one of the verification rule systems proposed by Hoare is

System H.

Axioms

HN. $P\{\textbf{null}\}P$ *HA.* $Q_E^{V}\{V := E\}Q$

Rules

HC. $\dfrac{P \wedge B\{S'\}Q,\ P \wedge \sim B\{S''\}Q}{P\{\textbf{if } B \textbf{ then } S' \textbf{ else } S'' \textbf{ fi}\}Q}$ *HW.* $\dfrac{A \wedge B\{S\}A}{A\{\textbf{while } B \textbf{ do } S \textbf{ end}\}A \wedge \sim B}$

HS. $\dfrac{P\{S'\}P',\ P'\{S''\}Q}{P\{S'; S''\}Q}$ *HI.* $\dfrac{P \supset P',\ P'\{S\}Q',\ Q' \supset Q}{P\{S\}Q}$

Example 1. For the following program, G, and unspecified P and Q

$V := E1;$
while B **do**
$V := E2$
end

the proof in H is

1.	$A_{E1}^{V}\{V := E1\}A$	*HA*
2.	$A_{E2}^{V}\{V := E2\}A$	*HA*
3.	$A \wedge B \supset A_{E2}^{V}$	*l*
4.	$A \supset A$	*l*
5.	$A \wedge B\{V := E2\}A$	*HI*, 2, 3, 4
6.	$A\{\textbf{while } B \textbf{ do } V := E2 \textbf{ end}\}A \wedge \sim B$	*HW*, 5
7.	$A_{E1}^{V}\{V := E1; \textbf{while } B \textbf{ do } V := E2 \textbf{ end}\}A \wedge \sim B$	*HS*, 1, 6
8.	$A \wedge \sim B \supset Q$	*l*
9.	$P \supset A_{E1}^{V}$	*l*
10.	$P\{V := E1; \textbf{while } B \textbf{ do } V := E2 \textbf{ end}\}Q$	*HI*, 7, 8, 9

"*l*" as a reason indicates that the statement should be proved in a first order logical theory L which contains the semantics of the assertion and expression languages. Thus if we interpreted V, $E1$, $E2$, B and A and proved the statements reasoned *l* in L, we would be able to claim $\vdash_{H,L} P\{G\}Q$. This displays the two levels of such a proof: H shows how to translate control flow through statements and the effects of assignment into logical expressions which are then submitted for proof in the system L. Another important feature of system H is that assertions are not directly attached to the program, but instead are entered as part of the proof. There are many ways of constructing proofs in H. One systematic way is to view the premises of the verification rules as subgoals for the conclusion. Such a procedure is nondeterministic in that there are choices where to break up statement lists using rule *HS* and when to apply rule *HI*.

It is advantageous to treat assertions as parts of programs for two reasons: (i) from a programming standpoint, assertions may be viewed as documentation and requiring them in the program encourages some program proving during, rather

than after, program construction and (ii) the proof generation process may be made somewhat more systematic. Therefore we will hereafter usually discuss asserted programs where the iteration statement has the form **while** ⟨boolean-expression⟩ **assert** ⟨logical expression⟩: ⟨statement-list⟩ **end** with the ⟨logical expression⟩ being from the assertion language. This gives rise to verification rule systems like the following:

System HB.

HBN. $$\frac{P \supset Q}{P\{\textbf{null}\}Q}$$

HBNN. $$\frac{P\{S\}Q}{P\{S;\ \textbf{null}\}Q}$$

HBS. $$\frac{P\{\textbf{null};\ S\}Q}{P\{S\}Q}$$

HBA. $$\frac{P\{S\}Q_E^V}{P\{S;\ V := E\}Q}$$

HBC. $$\frac{P1\{S'\}Q,\ P2\{S''\}Q,\ P\{S\}B \wedge P1 \vee \sim B \wedge P2}{P\{S;\ \textbf{if } B \textbf{ then } S' \textbf{ else } S'' \textbf{ fi}\}Q}$$

HBW. $$\frac{A \wedge \sim B \supset Q,\ A \wedge B\{S'\}A,\ P\{S\}A}{P\{S;\ \textbf{while } B \textbf{ assert } A;\ S' \textbf{ end}\}Q}$$

System HF.

HFN. $$\frac{P \supset Q}{P\{\textbf{null}\}Q}$$

HFS. $$\frac{P\{S;\ \textbf{null}\}Q}{P\{S\}Q}$$

HFA. $$\frac{(\exists V')(P_{V'}^V \wedge V = E_{V'}^V)\{S\}Q}{P\{V := E;\ S\}Q}$$ where V' is not used in V, E, P, Q, S

HFC. $$\frac{P \wedge B\{S'\}Q1,\ P \wedge \sim B\{S''\}Q2,\ Q1 \vee Q2\{S\}Q}{P\{\textbf{if } B \textbf{ then } S' \textbf{ else } S'' \textbf{ fi};\ S\}Q}$$

HFW. $$\frac{P \supset A,\ A \wedge B\{S'\}A,\ A \wedge \sim B\{S\}Q}{P\{\textbf{while } B \textbf{ assert } A\text{: } S' \textbf{end};\ S\}Q}$$

HFNN. $$\frac{P\{S\}Q}{P\{\textbf{null};\ S\}Q}$$

Example 2. Proof in *HB* for program *G* of Example 1:

1. $A \wedge \sim B \supset Q$	*l*
2. $A \wedge B \supset A^{V}_{E2}$	*l*
3. $A \wedge B\{\textbf{null}\}A^{V}_{E2}$	*HBN*, 2
4. $A \wedge B\{\textbf{null};\ V := E2\}A$	*HBA*, 3
5. $A \wedge B\{V := E2\}A$	*HBS*, 4
6. $P \supset A^{V}_{E1}$	*l*
7. $P\{\textbf{null}\}A^{V}_{E1}$	*HBN*, 6
8. $P\{\textbf{null};\ V := E1\}A$	*HBA*, 7
9. $P\{V := E1\}A$	*HBS*, 8
10. $P\{V := E1;\ \textbf{while}\ B\ \textbf{assert}\ A : V := E2\ \textbf{end}\}Q$	*HBW*, 1, 5, 9

The basic technique for generating a proof in either system is to work forward through the statement list in *HF* or backward through the statement list in *HB*. In this technique, no choices for assertions need be made during the proof: for example, in *HBC* there are assertions $P1$ and $P2$, but these may be generated by following the backward subgoaling technique on $P1\{S'\}Q$ and $P2\{S''\}Q$. The systems are related by the following:

THEOREM 1. *Let G be a program in the original syntax, A denote a collection of assertions (called an annotation), and* G^A *be G with the assertions of A attached to the appropriate iteration statements of G.*

$$\vdash_{H,L} P\{G\}Q \quad \textit{iff} \quad \textit{there exists } A \textit{ such that } \vdash_{HB,L} P\{G^A\}Q,$$

$$\vdash_{H,L} P\{G\}Q \quad \textit{iff} \quad \textit{there exists } A \textit{ such that } \vdash_{HF,L} P\{G^A\}Q.$$

Proof. Show by induction on the syntactic structure of program *G* the annotation. Likewise, from the annotation and the expressions generated by the rules come the necessary assertions for the proof in *H*. □

Remark. It is possible to have two annotations A' and A'' where $\vdash_{HB,L} P\{G^{A'}\}Q$ and not $\vdash_{HB,L} P\{G^{A''}\}Q$ because at least one assertion of A'' may not be sufficient to support application of rule *HBW*. The same remark holds for *HF*.

The proof technique used here and in later proofs is structural induction: assuming that a property holds for all substructures, show that it holds for each type of structure. The syntactic structure of programs is being used here and the types of structures are the statement types in the language. Hence, we will refer to this as *statement induction.* The proofs are usually very long and occasionally tricky. Many of the proofs omitted here will be found in a technical report, Gerhart [9].

The other approach mentioned in the introduction uses an algorithm for extracting verification lemmas from the program. The notion which captures this approach will be called a *verification operator system.* Each such system has two components: *VO*, a set of definitions of verification operators, and *CS*, a predicate expressed in terms of the verification operators on the program and the specifica-

tions. Examples of verification operator systems are

System BA.

$VO = \{bt, bvc\}$
$CS(P, G, Q) = (P \supset bt(G, Q)) \wedge bvc(G, Q)$
where bt and bvc are defined

G	$bt(G, Q)$	$bvc(G, Q)$
$V := E$	Q_E^V	true
null	Q	true
if B **then** S' **else** S'' **fi**	$(B \supset bt(S', Q)) \wedge (\sim B \supset bt(S'', Q))$	$bvc(S', Q) \wedge bvc(S'', Q)$
while B **assert** A: S **end**	A	$(A \wedge \sim B \supset Q) \wedge (A \wedge B \supset bt(S, A)) \wedge bvc(S, A)$
$S'; S''$	$bt(S', bt(S'', Q))$	$bvc(S'', Q) \wedge bvc(S', bt(S'', Q))$

System FA.

$VO = \{ft, fvc\}$
$CS(P, G, Q) = (ft(G, P) \supset Q) \wedge fvc(G, P)$
where ft and fvc are defined

G	$ft(G, P)$	$fvc(G, P)$
$V := E$	$(\exists V')(V = E_{V'}^V \wedge P_{V'}^V)$ where V' is unique	true
null	P	true
if B **then** S' **else** S'' **fi**	$ft(S', P \wedge B) \vee ft(S'', P \wedge \sim B)$	$fvc(S', P \wedge B) \wedge fvc(S'', P \wedge \sim B)$
while B **assert** A: S **end**	$A \wedge \sim B$	$(P \supset A) \wedge (ft(S, A \wedge B) \supset A) \wedge fvc(S, A \wedge B)$
$S'; S''$	$ft(S'', ft(S', P))$	$fvc(S', P) \wedge fvc(S'', ft(S', P))$

The basic idea of these operators is that bt and ft traverse the program backward and forward, respectively, and transform a logical expression according to the semantics of the statements traversed. At each **while** statement, the previously transformed expression is dropped and the process continues with the assertion attached to the statement. The bvc and fvc operators pick up this expression and form the associated verification conditions. The correctness statement then takes the form of a conjunction of expressions in the system L.

Example 3. Again using the asserted version of G from Example 1,

$$bt(V := E1; \textbf{while } B \textbf{ assert } A : V := E2 \textbf{ end}, Q)$$
$$\equiv bt(V := E1\ bt(\textbf{while } B \textbf{ assert } A : V := E2 \textbf{ end}, Q))$$
$$\equiv bt(V := E1, A)$$
$$\equiv A^V_{E1}$$

$$bvc(V := E1; \textbf{while } B \textbf{ assert } A : V := E2 \textbf{ end}, Q)$$
$$\equiv bvc(\textbf{while } B \textbf{ assert } A : V := E2 \textbf{ end}, Q) \wedge bvc(V := E1, A)$$
$$\equiv (A \wedge {\sim}B \supset Q) \wedge (A \wedge B \supset bt(V := E2, A)) \wedge bvc(V := E2, A) \wedge \text{true}$$
$$\equiv (A \wedge {\sim}B \supset Q) \wedge (A \wedge B \supset A^V_{E2})$$

$$CS_{BA}(P, G, Q) = (P \supset A^V_{E1}) \wedge (A \wedge {\sim}B \supset Q) \wedge (A \wedge B \supset A^V_{E2})$$

The reader should note the correspondence in Examples 2 and 3 between the logical expressions generated in the proof of *HB* and the correctness statement generated by *BA*. If we view the definition of the operators as a set of axioms, then correctness in *BA* can be denoted $\vdash_{BA,L} CS_{BA}(P, G, Q)$. The following theorem formalizes the above relationship.

THEOREM 2.

$$\vdash_{HB,L} P\{G\}Q \quad \textit{iff} \quad \vdash_{BA,L} CS_{BA}(P, G, Q),$$
$$\vdash_{HF,L} P\{G\}Q \quad \textit{iff} \quad \vdash_{FA,L} CS_{FA}(P, G, Q).$$

There is yet another version of verification operator systems which corresponds to the original formulation of the inductive assertion method where paths were explicitly used. This is captured in the systems

System BP.

$VO = \text{path-operators} \cup \{bp\}$ path-operators $= \left\{ \begin{matrix} pp, cp, \text{append, first,} \\ \text{middle, last} \end{matrix} \right\}$

$CS(P, G, Q) = (\forall p \in \text{paths}\ (P, G, Q))(\text{first}\ (p) \supset bp(\text{middle}\ (p), \text{last}\ (p)))$
where paths $(P, G, Q) = cp(G, \{P\}) \cup \text{append}\ (Q, pp(G, \{P\}))$ and pp and cp are defined

G	$pp(G, ps)$	$cp(G, ps)$
$V := E$	append $(V := E, ps)$	$\{\ \}$
if B **then** S' **else** S'' **fi**	$pp(S', \text{append}\ (B, ps)) \cup pp(S'', \text{append}\ ({\sim}B, ps))$	$cp(S', \text{append}\ (B, ps)) \cup cp(S'', \text{append}\ ({\sim}B, ps))$
while B **assert** A: S **end**	$\{A \wedge {\sim}B\}$	append $(A, ps) \cup cp(S, \{A \wedge B\})$ $\cup$ append $(A, pp(S, \{A \wedge B\}))$
$S'; S''$	$pp(S'', pp(S', ps))$	$cp(S', ps) \cup cp(S'', pp(S', ps))$
null	ps	$\{\ \}$

$$\text{append } (x, ps) \text{ is } \left\{ p' \middle| \begin{array}{c} p' = p \text{ with } x \text{ attached at the end} \\ \text{and } p \in ps \end{array} \right\}$$

"first", "last", and "middle" return the first, last, and list with first and last deleted, respectively

bp is defined (and *fp* is defined for *FP* below)

$\dfrac{p}{V := E}$	$\dfrac{bp(\langle p\rangle f)}{f_E^V}$	$\dfrac{fp(\langle p\rangle, f)}{(\exists V')(V = E_{V'}^V \wedge f_{V'}^V)}$ where V' is unique
B	$(B \supset f)$	$f \wedge B$
p', p''	$bp(p', bp(p'', f))$	$fp(p'', fp(p', f))$

System FP.

$$VO = \text{path-operators} \cup \{fp\} \qquad CS(P, G, Q) = (\forall p \in \text{paths } (P, G, Q))$$
$$(fp(\text{middle } (p), \text{first } (p)) \supset \text{last } (p))$$

The path operators *pp* and *cp* are similar to *ft* and *fvc*. *pp* forms partial paths starting with an assertion, appending the Boolean expressions of conditional statements and assignment statements as they are traversed. *cp* completes partial paths by appending assertions when they are reached and accumulating these complete paths as the whole program is traversed. *bp* and *fp* then translate a path into the correctness predicates for the path.

Example 4. In system *BP*, for program *G* we get

$$\begin{aligned} &pp(G, \{\langle P\rangle\}) \\ &\quad = pp(\textbf{while } B \textbf{ assert } A : V := E2 \textbf{ end}, pp(V := E1, \{\langle P\rangle\})) \\ &\quad = pp(\textbf{while } B \textbf{ assert } A : V := E2 \textbf{ end}, \{\langle P; V := E1\rangle\}) \\ &\quad = \{\langle A \wedge \sim B\rangle\} \end{aligned}$$

$$\begin{aligned} &cp(G, \{\langle P\rangle\}) \\ &\quad = cp(V := E1, \{\langle P\rangle\}) \cup \text{append } (A, pp(V := E1, \{\langle P\rangle\})) \\ &\qquad\qquad \cup cp(V := E2, \{\langle A \wedge B\rangle\}) \\ &\qquad\qquad \cup \text{append } (A, pp(V := E2, \{\langle A \wedge B\rangle\})) \\ &\quad = \{\ \} \cup \text{append } (A, \{\langle P, V := E1\rangle\}) \cup \text{append } (A, \{\langle A \wedge B, V := E2\rangle\}) \\ &\quad = \{\langle P, V := E1, A\rangle, \langle A \wedge B, V := E2, A\rangle\} \end{aligned}$$

$$\text{paths } (P, G, Q) = \{\langle P, V := E1, A\rangle, \langle A \wedge B, V := E2, A\rangle, \langle A \wedge \sim B, Q\rangle\}$$

$$\begin{aligned} CS_{BP}(P, G, Q) &= (P \supset bp(\langle V := E1\rangle, A)) \wedge \\ &\quad (A \wedge B \supset bp(\langle V := E2\rangle, A)) \wedge (A \wedge \sim B \supset bp(\langle\rangle, Q)) \\ &= (P \supset A_{E1}^V) \wedge (A \wedge B \supset A_{E2}^V) \wedge (A \wedge \sim B \supset Q) \end{aligned}$$

Again there is an obvious relationship in the example between the correctness statement of the explicit path method and the previous system, which might

be called the implicit path method because the paths are subsumed in the verification operators *bt* and *bvc*. This relationship is

THEOREM 3.

$$\vdash_{FA,L} CS_{FA}(P, G, Q) \quad \textit{iff} \quad \vdash_{FP,L} CS_{FP}(P, G, Q).$$

Proof. By statement induction, the following relations may be proved:

$$(\forall p \in cp(G, \{P\}))(fp(\text{middle}\ (p), \text{first}\ (p)) \supset \text{last}\ (p)) \equiv fvc(G, P),$$

$$(\forall p \in \text{append}\ (pp(G, \{P\}), Q))(fp(\text{middle}\ (p), \text{first}\ (p)) \supset \text{last}\ (p)) \equiv (ft(G, P) \supset Q).$$

The conjunction of the left-hand sides and the conjunction of the right-hand sides of the identities are the respective correctness statements of *FP* and *FA*. □

It has long been informally recognized that the forward and backward approaches should be consistent if the proper definitions are used. This can be formally stated and proved.

THEOREM 4.

$$\vdash_{BA,L} CS_{BA}(P, G, Q) \quad \textit{iff} \quad \vdash_{FA,L} CS_{FA}(P, G, Q).$$

THEOREM 5.

$$\vdash_{BP,L} CS_{BP}(P, G, Q) \quad \textit{iff} \quad \vdash_{FP,L} CS_{FP}(P, G, Q).$$

Proof. The paths are the same in both systems. By induction on the length of middle (p), it can be shown for each p,

$$(\text{first}\ (p) \supset bp(\text{middle}\ (p), \text{last}\ (p))) \equiv (fp(\text{middle}\ (p), \text{first}\ (p)) \supset \text{last}\ (p)).$$ □

To summarize, from Theorems 3, 4, and 5, we have

THEOREM 6. *The systems HB, HF, BA, FA, BP, and FP are all consistent in the sense that if given program G can be proved partially correct with respect to given P and Q in one system, the same can be proved in any other system.*

Thus, we have now formalized several implicitly assumed results of earlier work in partial correctness theory. The key unifying idea is that of the verification operator system which has been shown to be an alternative to verification rules. There are two specific gains made by having these results about consistent formulations of the inductive assertion method:

1. We can now intermix the systems in the proof of derived partial correctness properties. It will be seen that this flexibility condenses the size and very often the proof of properties which we will want to state and derive. Thus, we will hereafter regard our partial correctness theory as based on the combination of all of these systems, that is having as axioms and rules the union of those of all the systems described above.

2. Semantic definitions of language constructs not discussed here, such as for statements, jumps, and procedure calls, can become exceedingly complex. To make sure a definition is adequate, that is, correctly captures the intuitive and implemented semantics of the construct, the definition is often given in several systems and then their consistency is proved. The alternative formulations both give insight into different aspects of the construct and provide a checking

procedure on the definition. In addition, a computational type of definition in terms of a function Comp is often used to prove soundness of a denotational type of definition above as follows:

$$\vdash_L (\forall \text{ states } \xi)(P(\xi) \supset Q(\text{Comp}\,(G, \xi))) \quad \text{iff} \quad \vdash_{S,L} CS_S(P, G, Q).$$

It may be easier to prove this result for each system S indirectly using symmetry and consistency as above. These topics are discussed in papers by Hoare and Lauer [16], Igarashi et al. [17], Cook [2], and Donahue [6].

The formulation of verification operator systems is a generalization and unification of notations and ideas in Dijkstra [5], Igarashi et al. [17], King [19], and Sintzoff and Lamsweerde [23]. Gerhart [11] shows that the verification operators may be viewed as semantic attributes.

3. Derived rules. A standard technique in the practical use of logical systems is to state only a few rules in order to simplify proof theoretic results, such as consistency of these systems. However, as shown in Example 1, while the rules may yield the desired theorem, economy of rules may lead to longer proofs with numerous trivial steps which serve only to make the rules fit together. Therefore an accompanying standard technique is to derive from these basic rules further rules which shorten proofs and permit a wider range of proof techniques. For example,

D1. $$\frac{P\{S'\}Q1,\ Q1 \supset Q2,\ Q2\{S''\}Q}{P\{S';\ S''\}Q} \qquad \text{(derived in } H\text{)}$$

leads to fewer steps in proving sequencing of statements and

D2. $$\frac{P\{G\}Q,\ Q \supset Q'}{P\{G\}Q'}$$

D3. $$\frac{P\{G\}Q,\ P' \supset P}{P'\{G\}Q}$$

(derived in *HB* or *HF*)

give the same effect as rule *HI*. Furthermore, rules (derived in *H*, *HB*, *HF*)

D4. $$\frac{P1\{G\}Q,\ P2\{G\}Q}{P1 \vee P2\{G\}Q}$$

D5. $$\frac{P\{G\}Q1,\ P\{G\}Q2}{P\{G\}Q1 \wedge Q2}$$

D6. $$\frac{P1\{G\}Q1,\ P2\{G\}Q2}{P1 \vee P2\{G\}Q1 \vee Q2}$$

permit proof by cases of the preassertion and by separate proofs of conjuncts of the postassertion. The following rule allows a condensation of a chain of applications of assignment rules:

D7. $$\frac{P \supset (\cdots((Q_{E1}^{V1})_{E2}^{V2})\cdots)_{EN}^{VN}}{P\{V1 := E1;\ V2 := E2;\ \cdots;\ VN := EN\}Q}.$$

The advantage of multiple verification systems is shown in the proof of rule D5 using the *FA* system and Theorem 2

$$\begin{aligned} P\{S\}Q' \text{ and } P\{S\}Q'' \quad &\text{iff} \quad (ft(S, P) \supset Q') \wedge fvc(S, P) \wedge (ft(S, P) \supset Q'') \wedge fvc(S, P) \\ &\equiv \quad (ft(S, P) \supset Q' \wedge Q'') \wedge fvc(S, P) \\ &\text{iff} \quad P\{S\}Q' \wedge Q'' \end{aligned}$$

whereas an argument on proof structure would be necessary for a direct proof in *HB* or *HF*.

If we use the theory obtained by combining the systems, the following rules which mix notation may be derived:

D8. $\dfrac{fvc(S, P)}{P\{S\}ft(S, P)}$ D9. $\dfrac{P\{S\}Q, ft(S, P) \supset Q'}{P\{S\}Q'}$

D10. $\dfrac{bvc(S, Q)}{bt(S, Q)\{S\}Q}$ D11. $\dfrac{P\{S\}Q, P' \supset bt(S, Q)}{P'\{S\}Q}$

D12. $\dfrac{fvc(S, P), \text{ no variable of } P \text{ assigned to in } S}{P\{S\}P}$

D13. $P\{S'; S''\}Q$ iff $P\{S'\}bt(S'', Q)$ and $ft(S', P)\{S''\}Q$.

Rules D8–D11 identify conditions under which pre- and post-assertions may be varied. D12 is an important rule which shows some minimal conditions under which an assertion holds on both sides of a program. D13 is another important rule which we will use to establish further proof theoretic results.

Some of these and other rules have been given by Manna [21].

4. Ordering of verification systems. The verification systems which we have been discussing have a very strict requirement on assertions; each assertion must be sufficient to serve as the pre-assertion of a local proof, i.e., as the antecedent in a verification lemma. Therefore assertions have the characteristic that they must summarize the input assertion and the effects of all statements on every path from the input assertion to the point of assertion such that together with the paths and assertions from that point to the program end, the final assertion must hold. This results in assertions often being redundant with respect to both other assertions and the program, in addition to the difficulty of finding the crucial inductive aspects of loops. Consequently, there has been considerable research on automatic and semi-automatic generation of inductive assertions from the program and the specifications; e.g., see Wegbreit [24], Elspas [7], and Katz and Manna [18].

The notion behind this work is that the addition of certain types of verification operators can result in relaxation of the strict inductive assertion method in terms of the amount of annotation required to support a proof. Our concept of verification operator systems can be used to formalize this notion. We will define the following relations between two verification operator systems V and W:

$V \leqq W$ iff $CS_V(P, G, Q) \supset CS_W(P, G, Q)$ for all P, G, Q

$V < W$ iff there exists a (P, G, Q) such that $CS_W(P, G, Q) \wedge \sim CS_V(P, G, Q)$

$V \leftarrow W$ iff for all (P, G, Q) such that $CS_W(P, G, Q)$ there is a way of constructing a G^* such that $CS_V(P, G^*, Q)$.

As an example, we will consider
System FE.

$$VO = \{ft, fvce, fte, C\}$$
$$CS(P, G, Q) = (ft(G, P) \wedge fte(G, P) \supset Q) \wedge fvce(G, P, P)$$

G	$fte(G, f)$	$fvce(G, P, f)$
$V := E$	$C^{-1}(\{c \in C(f) \mid V \text{ is not in } c\})$	true
null	f	true
if B **then** S'	$C^{-1}(C(fte(S', f)) \cap C(fte(S'', f)))$	$fvce(S', P \wedge B, f) \wedge$
else S'' **fi**		$fvce(S'', P \wedge \sim B, f)$
while B **assert** A: S **end**	$fte(S, f)$	$(P \wedge f \supset A) \wedge (ft(S, A \wedge B) \wedge fte(S, f) \supset A) \wedge fvce(S, A \wedge B, fte(S, f))$
$S'; S''$	$fte(S'', fte(S', f))$	$fvce(S', P, f) \wedge fvce(S'', ft(S', P), fte(S', f))$

$$C(P) = \{c \mid c \text{ is a conjunct of } P\} \qquad C^{-1}(S) = \bigwedge_{s \in S} s$$

The operator ft is defined in *FA*. C breaks a predicate into conjuncts and C^{-1} conjoins a set of predicates. *fte* propagates conjuncts of the input assertion throughout the program as long as no assignment is made to any variable of the conjunct. *fvce* uses both *ft* and *fte* in forming verification conditions. We will now prove that the above relations hold between *FA* and *FE*.

THEOREM 7. (a) $FA \leqq FE$. (b) $FA < FE$. (c) $FA \leftarrow FE$.

Proof. (a) We must show that $(ft(G, P) \supset Q) \wedge fvc(G, P) \supset (ft(G, P) \wedge fte(G, P) \supset Q) \wedge fvce(G, P)$, which follows immediately from logic and $fvc(G, P) \supset fvce(G, P, P)$, which can be proved by statement induction.

(b) The program G and specifications P and Q used in the examples of § 2 with the interpretation

$$E1: 0 \qquad E2: V+1 \qquad B: V \leqq N \qquad A: \text{true} \qquad P: N \geqq 1 \qquad Q: N \geqq 1$$

is provable in *FE* but not in *FA*.

(c) Define yet another verification operator

$$1ce(S, S, P) = P$$
$$1ce(S, \textbf{if } B \textbf{ then } S' \textbf{ else } S'' \textbf{ fi}, P) =$$
$$\quad 1ce(S, \textbf{if } B \textbf{ then } S' \textbf{ else } S \textbf{ fi}, P) = P$$
$$1ce(S, \textbf{while } B \textbf{ assert } A: S \textbf{ end}, P) = fte(S, P)$$
$$1ce(S, S; S', P) = P$$
$$1ce(S, S'; S, P) = fte(S', P)$$

$1ce(S, G, P)$ denotes the "left context of statement S in program G with respect to operator *fte*" and is obtained by applying the *fte* operator to P and the program statements preceding S.

For every statement S of G which is of the form **while** B **assert** $A : S'$ **end** change A to $A \wedge 1ce(S', G, P)$ to get the program G^*. By induction it can be proved that for every statement S of G where $P' = 1ce(S, G, P)$,

$$ft(S^*, P \wedge P') \supset ft(S, P) \wedge fte(S, P'),$$
$$fvce(S, P \wedge P', P') \supset fvc(S^*, P \wedge P').$$

from which $CS_{FE}(P, G, Q) \supset CS_{FA}(P, G^*, Q)$. □

Other operators higher in this ordering could propagate internal assertions or the Boolean expressions governing conditional and while statements, deduce monotonicity of variables, solve systems of difference equations which describe changes in variables through looping, or massage output specifications into internal assertions, among other ideas suggested in the above referenced papers. Each of these techniques can be viewed as a verification operator. With the above ordering definitions, the higher order techniques can (and should) be proved sound relative to the strict inductive assertion method. In practice, it is often useful to refer to at least the propagation types of operators to supplement program annotations. The intuitive extremes of this ordering are verification systems where no internal assertions are required on any program or even where a program can be synthesized to satisfy given specifications.

There is not a one-to-one correspondence between verification operator systems and verification rule systems; e.g., it is impossible to distinguish in the rule systems whether some predicate being propagated arose in the input or in an internal assertion. The analogue is the addition of rules of the form

$$\frac{\text{conditions on } P, B, S, \text{ and } A \text{ which yield an } A' \text{ such that } A \wedge A' \wedge B\{S\}A \wedge A'}{P\{\textbf{while } B \textbf{ assert } A : S \textbf{ end}\}A \wedge A' \wedge \sim B}$$

which allow the use of A as an inductive assertion even when additional conditions (as stated in the premise of the rule) must be deduced. The above ordering definitions still hold for rule systems. We conjecture that the ordering of verification operations systems is a refinement of the ordering of verification rule systems.

5. Program schemas. While the results about verification systems provide the basis for a theory of partial correctness of programs, that theory does not yet express and organize what programmers know about the programs covered by this theory. A natural approach to remedy this situation is to identify schemas expressing patterns which commonly occur as or within programs. In this section we will see two such schema and how they can be used. Yet another aspect of world knowledge about programming is that there are often many ways of writing programs to obtain the same effect with respect to specifications, but with different amounts of effort required to write the programs, levels of understandability, and degrees of efficiency. This fact, along with the observation that otherwise the number of schemas could grow excessively, motivates the notion of correctness-preserving program transformations which will be discussed in the next section.

Together, schemas and transformations constitute a formalization of a methodology for systematic construction of correct programs which will be illustrated in § 7.

Dijkstra [4] has observed a common pattern of programs called the "linear search":

$$\text{true}$$
$$\left\{\begin{array}{l} I := 1; D := f(0); \\ \textbf{while } \sim g(D) \textbf{ assert } D = f^I(0) \wedge \text{unfound } (I): \\ D := f(D); I := I+1 \textbf{ end} \end{array}\right\}$$
$$D = f^I(0) \wedge \text{unfound } (I) \wedge g(D)$$

where unfound $(I) = (\forall i)(1 \leqq i < I \supset \sim g(f^i(0)))$. The correctness statement for this schema is

$$\begin{array}{l} (\text{true} \supset f(0) = f^1(0) \wedge \text{unfound } (1)) \wedge \\ (D = f^I(0) \wedge \text{unfound } (I) \wedge \sim g(D) \supset f(D) = f^{I+1}(0) \\ \wedge \text{unfound } (I+1)) \end{array}$$

which is valid. Therefore, for any interpretation of D, f, g, the resulting program will also be correct. A variation of this schema is

$$\text{true}$$
$$\left\{\begin{array}{l} D := f(0); \textbf{while } \sim g(D) \textbf{ assert } (\exists i)(D = f^i(0) \wedge \text{unfound } (i)): \\ \qquad\qquad\qquad\qquad D := f(D) \textbf{ end} \end{array}\right\}$$
$$(\exists i)(D = f^i(0) \wedge \text{unfound } (i)) \wedge g(D)$$

Consider interpreting this latter schema to give a program which searches a vector V for an element x.

$$D\colon I, f(x)\colon \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x+1,\ g(x)\colon\ V(I) = X$$

The following program is partially correct as an instance of the above schema after simplifying using $f^i(0) = i$:

$$\text{true}$$
$$\left\{\begin{array}{l} I := 1; \\ \textbf{while } V[I] \neq X \textbf{ assert } (1 \leqq i < I \supset V[i] \neq X): \\ \quad I := I+1 \textbf{ end} \end{array}\right\}$$
$$(1 \leqq i < I \supset V[i] \neq X) \wedge V[I] = X$$

Notice that partial correctness becomes quite evident here since there is no guarantee that X is anywhere in the vector V and the program could produce a subscript error.

Another interpretation of the schema is

$$g(D)\colon 0 = X \bmod D, f(X)\colon \textbf{if } x = 0 \textbf{ then } 2 \textbf{ else } x+1$$

which finds the first divisor of X. A variant of this schema has been used in the proof of a complicated permutation algorithm; see Yelowitz and Duncan [27].

Another common pattern is the initialization of a vector. To avoid redefining our verification rules and operators for array assignment, assume that the semantics of array assignment and selection is incorporated in the following axiom:

$$\text{sel } (\text{assn } (V, i, X), j) = \textbf{if } i = j \textbf{ then } X \textbf{ else } \text{sel } (V, j)$$

The following schema can be proved correct:

$N \geqq 1$

$$\left\{\begin{array}{l} I := 1; \\ \textbf{while } I \leq N \textbf{ assert } 1 \leq I \leq N+1 \wedge (\forall i)(1 \leq i < I \supset \text{sel}(V, i) = g(i)): \\ \quad V := \text{assn}(V, I, g(I)); \\ \quad I := I+1 \textbf{ end} \end{array}\right\}$$

$(\forall i)(1 \leq i \leq N \supset \text{sel}(V, i) = g(i))$

An obvious interpretation is to let g be the constant function O. Section 7 will show how this schema can be manipulated when g is given a more specific, but still general, form.

It will not usually be the case that all of the schematic proof can be carried out, but the residue which is unprovable (due to lack of interpretation) can be left as premises and packaged in rule form

$$\frac{\text{the reduced correctness statement for } P\{G\}Q}{P\{G\}Q}.$$

The main point is that the proof of each program covered by the schema is factored into two parts: (i) the common part which need be proved only once at the schematic level and (ii) the more domain specific part.

The potential gains of such an approach are immense if a large enough body of schemas can be found, if these schemas are sufficiently organized that the match between programs and schemas can be readily seen, and if persons trying to prove programs can be trained to use them. Such a body of schemas must be acquired through constant generalization over a great variety of programs. Other examples of the use of schema are found in Wirth [25], Gerhart [12], and Manna [22].

6. Correctness-preserving program transformations. The idea of first writing a reasonably simple program which solves a set task and then improving that program through systematic manipulation has been mentioned in Knuth [20], Wirth [26], Burstall and Darlington [1], and Gerhart [11], [12]. Such transformations should be stated in terms of schemas in order to gain generality. We will now show that this informal idea corresponds to a formal Replacement Theorem which is often a part of a logical theory. Our goal will be to derive rules of the form

$$\frac{P\{G\}Q, \text{ (conditions on } P, G, \text{ and } Q)}{P\{G'\}Q}$$

where G' is formed by the replacement of some part of G. The rule should show that the replacement preserves correctness. An additional goal is to keep the amount of extra conditions sufficiently simple that the replacement rule can be applied with little required proof. We will often write rules like the above as

$$\frac{\text{conditions on } P, G, Q}{P\{G\}Q \Rightarrow P\{G'\}Q}.$$

We have mentioned previously that the essence of the inductive assertion method is that assertions serve to establish local contexts in which proofs of

correctness are performed. The following theorem shows that there is a context for each statement of the program. First we will define two verification operators *lc* and *rc* which stand for "left context", whatever can be shown to hold for the statement given preceding assertions and statements and "right context", whatever must hold after the statement such that it together with the succeeding statements will guarantee succeeding assertions.

G	$lc(S, G, P)$	$rc(S, G, Q)$
S	P	Q
if B **then** S **else** S' **fi**	$P \wedge B$	Q
if B **then** S' **else** S **fi**	$P \wedge \sim B$	Q
while B **assert** A: S **end**	$A \wedge B$	A
$S; S'$	P	$bt(S', Q)$
$S'; S$	$ft(S', P)$	Q

LEMMA 7.

$$lc(S', S, lc(S, G, P)) = lc(S', G, P),$$

$$rc(S', S, rc(S, G, Q)) = rc(S', G, Q).$$

THEOREM 8 (Decomposition). *$P\{G\}Q$ iff either*

(a) *for every substatement S of G, $lc(S, G, P) \supset bt(S, rc(S, G, Q))$ and $ft(G, P) \supset Q$.*

or (b) *for every substatement S of G, $ft(S, lc(S, G, P)) \supset rc(S, G, Q)$ and $P \supset bt(G, Q)$.*

COROLLARY. *$P\{G\}Q$ iff*

for every proper substatement S of G, $lc(S, G, P)\{S\}rc(S, G, Q)$ and $(ft(G, P) \supset Q) \wedge (P \supset bt(G, Q))$.

THEOREM 9 (Replacement). *Let S be a statement of G and G' be G with S replaced by S'. Then*

$$\frac{lc(S, G, P)\{S'\}rc(S, G, Q)}{P\{G\}Q \Rightarrow P\{G'\}Q}.$$

Proof. Using the decomposition theorem on the premise $P\{G\}Q$ we have that every statement of G satisfies its context. The second premise requires that S' satisfy the context of S in G, but this context is the same as the context of S' in G'. The contexts of other statements of G' may be different than in G, but it can be shown by induction that the second premise guarantees that each statement of G' other than S' also satisfies its context. The decomposition theorem then applies to yield $P\{G'\}Q$. □

The replacement theorem suggests the following procedure for performing a correctness-preserving transformation on a proved correct program.

1. Compute the context of the statement to be replaced.
2. Show that the replacing statement satisfies that context.
3. Perform the actual replacement yielding another correct program.

Concrete examples will be given in the next section. For now, consider two types of somewhat abstract transformations. The syntax of the language includes

the units statement, assertion, boolean-expressions, and expressions. These give rise to generic types of transformations, for example, statement insertion and deletion

$$T1a.\ \frac{P\{S\}bt(S', Q),\ bvc(S', Q)}{P\{S;\ S'\}Q} \qquad T1c.\ \frac{ft(S', P)\{S\}Q,\ fvc(S', P)}{P\{S';\ S\}Q}$$

$$T1b.\ \frac{P\{S\}P}{P\{S'\}Q \Rightarrow P\{S;\ S'\}Q} \qquad T1d.\ \frac{Q\{S\}Q}{P\{S'\}Q \Rightarrow P\{S';\ S\}Q}$$

$$T2a.\ \frac{P \supset bt(S', Q)}{P\{S;\ S'\}Q \Rightarrow P\{S'\}Q} \qquad T2b.\ \frac{ft(S', P) \supset Q}{P\{S';\ S\}Q \Rightarrow P\{S'\}Q}$$

and assertion replacement

$T3$.

$$\frac{H}{P\{\textbf{while } B \textbf{ assert } A : S \textbf{ end}\}Q = > P\{\textbf{while } B \textbf{ assert } A' : S \textbf{ end}\}Q}$$

where H is one of the following:

(a) (Replacement) $P \supset A'$, $A' \wedge B\{S\}A'$, $A' \wedge \sim B \supset Q$
(b) (Weakening) $A \supset A'$, $A' \wedge B\{S\}A'$, $A' \wedge \sim B \supset Q$
(c) (Strengthening) $A' \supset A$, $P \supset A'$, $A' \wedge B\{S\}A'$
(d) ($\wedge$-introduction) $A' \equiv A \wedge A''$, $P \supset A''$, $B \wedge A'\{S\}A''$
(e) ($\wedge$-elimination) $A \equiv A' \wedge A''$, $A' \wedge B\{S\}A'$, $A' \wedge \sim B \supset Q$
(f) ($\vee$-introduction) $A' \equiv A \vee A''$, $A'' \wedge B\{S\}A'$, $A'' \wedge \sim B \supset Q$
(g) ($\vee$-elimination) $A \equiv A' \vee A''$, $P \supset A'$, $A' \wedge B\{S\}A'$.

Another type of transformation is more specific, for example manipulations of assignment statements

$$T4a.\ \frac{V \text{ not free in } bt(S, Q)}{P\{S\}Q \Leftrightarrow P\{V := E;\ S\}Q} \qquad T4b.\ \frac{v \text{ not free in } Q}{P\{S\}Q \Leftrightarrow P\{S;\ V := E\}Q}$$

$$T4c.\ \frac{\begin{array}{c}VO \text{ not free in } Q \\ P \supset (E1' = E1^{VO}_{EO}) \wedge (E2' = E2^{VO}_{EO})\end{array}}{\begin{array}{l}P\{VO := EO;\ V1 := E1;\ V2 := E2\}Q \\ \Leftrightarrow P\{V1 := E1';\ V2 := E2'\}Q\end{array}} \qquad T4d.\ \frac{\begin{array}{c}P \supset E2' = E2^{V1}_{E1} \\ V2 \text{ not free in } E1\end{array}}{\begin{array}{l}P\{V1 := E1;\ V2 := E2\}Q \\ \Leftrightarrow P\{V2 := E2';\ V1 := E1\}Q\end{array}}$$

$$T4e.\ \frac{\begin{array}{c}P \supset E1 = E1'^{V1}_{E1} \\ V1 \text{ not free in } E2\end{array}}{\begin{array}{l}P\{V1 := E1;\ V2 := E2\}Q \Leftrightarrow \\ P\{V2 := E2;\ V1 := E1\}Q\end{array}} \qquad T4f.\ \frac{P \supset E = E'}{P\{V := E\}Q \Leftrightarrow P\{V := E'\}Q}$$

and loops:

Loop manipulations.

$$T5a.\ \frac{\text{true}}{P\{\textbf{while } B1 \textbf{ assert } A : \textbf{if } B2 \textbf{ then } S1 \textbf{ else } S2 \textbf{ fi}\}Q}$$

$$\Leftrightarrow P\left\{\begin{array}{l}\textbf{while } B1 \textbf{ assert } A:\\ \textbf{while } B1 \wedge B2 \textbf{ assert } A: S1 \textbf{ end};\\ \textbf{while } B1 \wedge B2 \textbf{ assert } A: S2 \textbf{ end}\\ \textbf{end}\end{array}\right\} Q$$

$T5b.$ $\dfrac{\text{true}}{P\{\textbf{if } B \textbf{ then } S \textbf{ else null}\}P}$
$\Rightarrow P\{\textbf{while } B \textbf{ assert } P: S \textbf{ end}\}P$

$T5c.$ $\dfrac{P \supset a \leqq b,\ I \text{ not assigned to in } S}{P\{I := a; \textbf{while } I \leqq b \textbf{ assert } A: S: I := I+1 \textbf{ end}\}Q}$
$\Rightarrow P\{S_a^I; I := a+1; \textbf{while } I \leqq b \textbf{ assert } A \wedge I \geqq a+1: S: I := I+1 \textbf{ end}\}Q$

$T5d.$ $\dfrac{\begin{array}{c}E1 = f(a),\ K = f(I-1)\{K := E2\}K = f(I)\\ K \text{ not free in } A, S, P, Q\end{array}}{P\{I := a; \textbf{while } I \leqq b \textbf{ assert } A: S; I := I+1 \textbf{ end}\}Q \Rightarrow}$
$$P\left\{\begin{array}{l} I := a; K := E1;\\ \textbf{while } I \leqq b \textbf{ assert } A \wedge K = f(I): S; I := I+1; K := E2 \textbf{ end}\end{array}\right\} Q$$

Most of these rules can be easily derived by first writing down the correctness statement for $P\{G\}Q$ and then the correctness statement for $P\{G'\}Q$ and figuring out the difference H so that the first correctness and H will imply the second correctness statement. For example, in rule $T3e$, the first correctness statement is

$$P \supset A' \wedge A, A' \wedge A'' \wedge B\{S\}A' \wedge A'', A' \wedge A'' \wedge \sim B \supset Q$$

and the premises for the second are

$$P \supset A', A' \wedge B\{S\}A', A' \wedge \sim B \supset Q.$$

Only the second premise does not follow from the preceding premises.

For rule $T5a$, we need no premises H since

$P\{\textbf{while } B1 \textbf{ assert } A: \textbf{if } B2 \textbf{ then } S1 \textbf{ else } S2 \textbf{ fi}\}Q$ iff
$P \supset A,\ A \wedge B1 \wedge B2\{S1\}A,\ A \wedge B1 \wedge \sim B2\{S2\}A,\ A \wedge \sim B1 \supset Q$
$P\{\textbf{while } B1 \textbf{ assert } A: \textbf{while } B1 \wedge B2 \textbf{ assert } A: S1 \textbf{ end}; \textbf{while } B1 \wedge \sim B2 \textbf{ assert } A: S2 \textbf{ end end}\}Q$ iff
$P \supset A,\ A \wedge B1 \supset A,\ A \wedge B1 \wedge B2\{S1\}A,\ A \wedge \sim(B1 \wedge B2) \supset A,\ A \wedge B1 \wedge \sim B2\{S2\}A,$
$A \wedge \sim(B1 \wedge \sim B2) \supset A,\ A \wedge \sim B1 \supset Q$

As for program schema, we must systematically generalize from specific programs transformations to find a larger body of transformations and we must learn how to use such transformations. The examples of the next section illustrate the methodology for constructing correct programs.

7. Examples of the use of derived rules, schema and transformations.

Example 1. The following problem was considered in Wirth [26]: find the product of two integers a and b (b nonnegative) by repeated addition, doubling, and halving only. An informal attack on the problem attributed to Dijkstra consisted of three steps:

1. Write down and prove the most obvious algorithm

$$b \geqq 0 \left\{ \begin{array}{l} x := a;\ y := b;\ z := 0 \\ \textbf{while } y \neq 0 \\ \quad \textbf{assert } y \geqq 0 \wedge x^*y + z = a^*b: \\ \quad y := y - 1;\ z := z + x \\ \quad \textbf{end} \end{array} \right\} z = a^*b$$

2. Notice that when y is even, the program may be speeded up by a different operation which still preserves the invariant:

insert at the beginning of the loop body
if even(y) **then** $y := y$ div 2; $x := x^*2$ **fi**

3. Applying the informal notion "if a relation is invariant over a statement, it remains so regardless of how often the statement is executed", replace the statement inserted above by a loop, getting

$$b \geqq 0 \left\{ \begin{array}{l} x := a;\ y := b;\ z := 0; \\ \textbf{while } y \neq 0 \\ \quad \textbf{assert } y \geqq 0 \wedge x^*y + z = a^*b: \\ \quad \textbf{while } \text{even}\,(y) \\ \quad\quad \textbf{assert } y > 0 \wedge x^*y + z = a^*b: \\ \quad\quad y := y \text{ div } 2;\ x := x^*2 \\ \quad\quad \textbf{end}; \\ \quad y := y - 1;\ z := z + x \\ \quad \textbf{end} \end{array} \right\}$$

More formally, the transformation which expresses step 2 is $T1$b. P and Q in the transformation are the context of the $y := y - 1$ where

P is $y \geqq 0 \wedge x^*y + z = a^*b \wedge y \neq 0$ which is $y > 0 \wedge x^*y + z = a^*b$,
S' is $y := y - 1$, S is **if** even (y) **then** $y := y$ **div** 2; $x := x^*2$ **fi**,
Q is $y \geqq 0 \wedge x^*y + (z + x) = a^*b$.

The premise requires proof that

(i) $y > 0 \wedge x^*y + z = a^*b \wedge \text{even}\,(y) \supset (y \text{ div } 2) > 0 \wedge (x^*2)^*(y \text{ div } 2) + z = a^*b$.

Transformation $T5$b shows that **while** even (y) **assert** P: $y := y$ div 2; $x := x^*2$ **end** also satisfies the premise of transformation and so may be inserted to give program 3.

Alternatively, using only the replacement theorem, we would replace $y := y - 1$ by preceding it with the **while** even (y)...statement. This requires

$P\{\textbf{while} \cdots ; y := y-1\}Q$ which includes (i) above and

(ii) $y>0 \wedge x^*y+z=a^*b \wedge \sim \text{even}(y) \supset (y-1) \geqq 0 \wedge x^*(y-1)+(z+x) = a^*b$ and

(iii) $y \geqq 0 \wedge x^*y+z=a^*b \wedge y \neq 0 \supset y>0 \wedge x^*y+z=a^*b.$

We can now compare proofs:

The correctness statement for 1 is

(iv) $(b \geqq 0 \supset b \geqq 0 \wedge a^*b+0=a^*b) \wedge$

(v) $(y \geqq 0 \wedge x^*y+z=a^*b \wedge y \neq 0 \supset (y-1) \geqq 0 \wedge x^*(y-1)+(z+x)=a^*b) \wedge$

(vi) $(y \geqq 0 \wedge x^*y+z=a^*b \wedge y=0 \supset z=a^*b).$

The correctness statement for 3 is

$$\text{(iv)} \wedge \text{(iii)} \wedge \text{(i)} \wedge \text{(ii)} \wedge \text{(vi)}.$$

Using specific transformations requires

$$\text{(iv)} \wedge \text{(v)} \wedge \text{(vi)} \wedge \text{(i)}.$$

Using the replacement theorem requires

$$\text{(iv)} \wedge \text{(v)} \wedge \text{(vi)} \wedge \text{(i)} \wedge \text{(ii)} \wedge \text{(iii)}.$$

Thus, the smallest proof in terms of number of verification conditions is the specific transformation proof because one condition is absorbed in the proof of transformation $T5$b.

However, the main point of the methodology is not only reduced proof size, but also flexibility to systematically modify an initial program until a more efficient, or otherwise desirable, program is obtained, while knowing that the program is correct after each modification. We now claim that the informal reasoning used in Wirth's example is supported by the above more formal reasoning.

Example 2. In § 5, we gave a schema for initializing a vector. Now suppose we also know about g that it is $g(i) = \textit{if } i=1 \text{ then } h1 \text{ else } h2(g(i-1), i)$. We will now show how to derive another correct schema using the more specific transformations of the last section. The two ideas for improvement of the efficiency of computing g are to initialize for $g(1)$ before the loop and to introduce an additional variable K which will hold the last computed value of g. We start with the schema

$$\begin{array}{l} 1 \leqq N \\ \left\{\begin{array}{l} I := 1; \\ \textbf{while } I \leqq N \textbf{ assert } A\colon V := \text{assn}(V, I, g(I)); I := I+1 \textbf{ end} \end{array}\right\} \\ 1 \leqq i \leqq N \supset \text{sel}(V, i) = g(i) \end{array} \tag{1}$$

where A is $(\forall i)(1 \leqq i < I \supset \text{sel}(V, i) = g(i)) \wedge 1 \leqq I \leqq N+1$.

Application of the loop unraveling transformation $T5$c and $T4$f to simplify $g(1)$ to $h1$ and $1+1$ to 2 gives

$$\begin{array}{l} 1 \leqq N \\ \left\{\begin{array}{l} V := \text{assn}(V, 1, h1); I := 2; \\ \textbf{while } I \leqq N \textbf{ assert } A \wedge I \geqq 2; \\ \quad V := \text{assn}(V, I, g(I)); I := I+1; \textbf{ end} \end{array}\right\} \end{array} \tag{2}$$

$(\forall i)(1 \leqq i \leqq N \supset \text{sel}(V, i) = g(i)).$

Now we apply the inductive variable insertion transformation $T5$d

$$1 \leqq N$$
$$(3) \qquad \left\{\begin{array}{l} V := \text{assn}(V, 1, h1);\ I := 2;\ K := g(I-1); \\ \textbf{while } I \leqq N \textbf{ assert } A \wedge I \geqq 2 \wedge K = g(I-1): \\ \quad V := \text{assn}(V, I, g(I));\ I := I+1;\ K := g(I-1) \textbf{ end} \end{array}\right\}$$

$(\forall i)(1 \leqq i \leqq N \supset \text{sel}(V, i) = g(i))$.

Applying transformations $T4$d and $T4$e we can transform the program fragment before the loop to

$$(4) \qquad V := \text{assn}(V, 1, h1);\ K := h1;\ I := 2$$

and then to

$$(5) \qquad K := h1;\ V := \text{assn}(V, 1, K);\ I := 2.$$

Similarly, the body of the loop can be transformed by $T4$d to

$$(6) \qquad V := \text{assn}(V, I, g(I));\ K := g(I);\ I := I+1$$

then by $T4$e to

$$(7) \qquad K := g(I);\ V := \text{assn}(V, I, K);\ I := I+1$$

and finally by $T4$f to

$$(8) \qquad K := h2(K, I);\ V := \text{assn}(V, I, K);\ I := I+1.$$

Altogether this gives the more efficient final program

$$1 \leqq N$$
$$\left\{\begin{array}{l} K := h1;\ V := \text{assn}(V, 1, K);\ I := 2; \\ \textbf{while } I \leqq N \textbf{ assert } A \wedge I \geqq 2 \wedge K = g(I-1): \\ \quad K := h2(K, I);\ V := \text{assn}(V, I, K);\ I := I+1 \textbf{ end} \end{array}\right\}$$

$(\forall i)(1 \leqq i \leqq N \supset \text{sel}(V, i) = g(i))$.

The premises used in each of the above rules are easily seen to be true:
(2) $1 \leqq N \wedge I = 1 \supset 1 \leqq N,\ g(1) = h1,\ 1+1 = 2$
(3) $g(I-1) = g(I-1),\ K = g(I-2)\{K := g(I-1)\}K = g(I-1)$, K not free in $A \wedge I \geqq 2$,
$V := \text{assn}(V, I, g(I));\ I := I+1,\ 1 \leqq N,\ (\forall i)(1 \leqq i \leqq N \supset \text{sel}(V, i) = g(i))$
(4) $g(2-1) = h1$, K not free in 2
(5) $\text{assn}(V, 1, h1) = \text{assn}(V, 1, K)^K_{h1}$, V not free in $h1$
(6) $g(I) = g(I-1)^I_{I+1}$, K not free in $I+1$
(7) $\text{assn}(V, I, g(I)) = \text{assn}(V, I, K)^K_{g(I)}$, V not free in $g(I)$
(8) $A \wedge I \geqq 2 \wedge K = g(I-1) \supset g(I) = h2(K, I)$.

The advantages of this methodology are that the steps which are taken throughout the program modification process are very small and easily proved since most of the proving is subsumed in the derivation of the transformations. Of course, the disadvantage is that the transformations must be proved, but this is worth the effort if the transformations can be used often. Another disadvantage is the large amount of writing, but this is potentially mechanized by a program which can store, retrieve, and apply schemas and transformations as well as perform

some of the simplification types of theorem proving which are characteristic of this approach.

8. Conclusions and directions for further research. The goal of a potentially powerful methodology for systematically constructing correct programs through partially proved schemas, correctness-preserving program transformations, incremental proofs, completion of partial program annotations, and appropriate mechanical assistance has been the theme of the last several years of research in program correctness. The present paper has attempted to bridge the gap between earlier work which laid the foundations of a theory of partial correctness and the proof theory needed to support the full methodology. The main ideas are the formal equivalence, and therefore interchangeability, of several versions of the inductive assertion method, the fact that programming knowledge can be expressed in partially proved schemas, the replacement theorem which supports preservation of correctness under generic and specific types of transformations, and extension of the basic systems with operators which augment given annotations.

Much more work needs to be done before a fully developed theory of program correctness will exist. The main needs are

(i) extending the present proof theory with similar results for proving termination, and

(ii) defining the semantics of more powerful programming language constructs and then studying their properties within the theory, and

(iii) expressing and organizing world knowledge about programs as gained through experience and the study of human factors in programming, which is the subject of structured programming research.

REFERENCES

[1] R. M. BURSTALL AND J. DARLINGTON, *Some transformations for developing recursive programs*, Proc. Internat. Conf. on Reliable Software, Los Angeles, Calif., 1975, pp. 465–472.

[2] S. COOK, *Axiomatic and interpretive semantics for an ALGOL fragment*, Tech. Rep. 79, Comput. Sci. Dept., Univ. of Toronto, 1975.

[3] J. DE BAKKER AND L. G. L. T. MEERTENS, *On the completeness of the inductive assertion method*, J. Comput. System Sci., 11(1975), pp. 323–357.

[4] E. W. DIJKSTRA, *Notes on structured programming*, Structured Programming, Academic Press, New York, 1972.

[5] ———, *Guarded commands, non-determinacy, and a calculus for the derivation of programs*, Comm. ACM, 18 (1975), pp. 453–457.

[6] J. DONAHUE, *The mathematical semantics of axiomatically defined programming language constructs*, Proc. IRIA Colloq. on Proving and Improving Programs, Arc et Senans, France, July 1975, pp. 353–370.

[7] B. ELSPAS, *The semiautomatic generation of inductive assertions for proving program correctness*, Rep. 2686, Stanford Research Inst., Menlo Park, Calif., July 1974.

[8] R. FLOYD, *Assigning meanings to programs*, Proc. Symp. on Appl. Math., vol. 19, J. Schwarz, ed., American Mathematical Society, Providence, R.I., 1967, pp. 19–32.

[9] S. GERHART, *Proof theory of partial correctness verification systems*, Tech. Rep. CS-76-3, Computer Sci. Dept., Duke Univ., 1976.

[10] ———, *Verification operator systems and their application to logical analysis of programs*, Proc. IRIA Colloq. on Proving and Improving Programs, Palo Alto, Calif., July 1975, pp. 209–224.

[11] ———, *Correctness-preserving program transformations*, Proc. 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, Calif., Jan. 1975, pp. 54–66.

[12] ———, *Knowledge about programs: A model and case study*, Proc. Internat. Conf. on Reliable Software, Los Angeles, Calif., 1975, pp. 88–95.

[13] C. A. R. HOARE, *An axiomatic basis for computer programming*, Comm. ACM, 12 (1969), pp. 576–580, 583.

[14] ———, *Procedures and parameters: An axiomatic approach*, Symp. on Semantics of Algorithmic Languages, E. Engeler, ed., Springer-Verlag, New York, 1971, pp. 102–106.

[15] C. A. R. HOARE AND N. WIRTH, *An axiomatic definition of the programming language PASCAL*, Acta Informatica, 2 (1973), pp. 335–355.

[16] C. A. R. HOARE AND P. LAUER, *Consistent and complementary definitions of the semantics of programming languages*, Ibid., 3 (1973), pp. 135–153.

[17] S. IGARASHI, R. LONDON AND D. LUCKHAM, *Automatic program verification I*, Ibid., 4 (1975), pp. 145–182.

[18] S. KATZ AND Z. MANNA, *A heuristic approach to program verification*, Proc. 3rd Internat. Conf. on Artificial Intelligence, Palo Alto, Calif., 1973, pp. 500–512.

[19] J. KING, *A program verifier*, Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, 1969.

[20] D. E. KNUTH, *Structured programming with go to statements*, Comput. Surveys, 6 (1974), pp. 247–260.

[21] Z. MANNA, *Mathematical theory of computation*, McGraw-Hill, New York, 1974.

[22] Z. MANNA AND N. DERSCHOWITZ, *On automating structured programming*, Proc. IRIA Colloq. on Proving and Improving Programs, Arc et Senans, France, July 1975, pp. 167–196.

[23] M. SINTZOFF AND A. VAN LAMSWEERDE, *Constructing correct and efficient concurrent programs*, Proc. Internat. Conf. on Reliable Software, Los Angeles, Calif., 1975, pp. 319–326.

[24] B. WEGBREIT, *The synthesis of loop predicates*, Comm. ACM, 17 (1974), pp. 102–112.

[25] N. WIRTH, *Systematic Programming*, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[26] ———, *On the composition of well-structured programs*, Comput. Surveys, 6 (1974), pp. 247–260.

[27] L. YELOWITZ AND A. DUNCAN, *Loop unravelling: A practical tool in proving program correctness*, Information Processing Lett., 4 (1975), pp. 70–72.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

CORRECT COMPUTATION RULES FOR RECURSIVE LANGUAGES*

PETER J. DOWNEY AND RAVI SETHI†

Abstract. This paper considers simple, LISP-like languages for the recursive definition of functions. We focus on the connections between formal *computation rules* for calculation with recursive definitions, and the mathematical semantics of such definitions. A computation rule is *correct* when it is capable of computing the least fixpoint of a recursive definition. We give necessary and sufficient conditions for the correctness of rules under (a) all possible interpretations and (b) particular interpretations.

Key words. computation rules, recursive programs, mathematical semantics of programming languages, correct rules

1. Introduction. The factorial function $f(n) \Leftarrow$ if $n = 0$ then 1 else $n * f(n-1)$; the Fibonacci sequence $f(n) \Leftarrow$ if $n = 0$ or 1 then 1 else $f(n-1) + f(n-2)$; BNF definitions like ⟨sequence⟩ : : = ⟨sequence⟩⟨item⟩|⟨item⟩ (or PBF as Ingerman suggests [9]), are all examples of recursive definitions. Recursive specifications are a natural way of viewing functions and programs. Gries [8] gives examples of programs constructed from recursive specifications that are easy to prove correct. Recursion and its implementation will be studied in this paper. While the examples and motivation will tend to be drawn from the area of programming languages, the results are equally applicable to formal languages. Downey [6] considers the use of recursive equations for defining languages and language functions.

ALGOL 60 is one of the first programming languages to admit recursive procedures. ALGOL provides two mechanisms for procedure evaluation—call by name and call by value [15, §§ 4.7.3.1, 4.7.3.2]. It is well known [14] that the two mechanisms are not equivalent; the difference lies in how the parameters or arguments of a procedure call are treated. Call by value first evaluates the actual parameters in the call, and then executes the procedures using the computed values. With call by name, on the other hand, formal parameters are uniformly replaced by the corresponding actual parameters in the procedure body.

Example 1.1. Consider the recursive definition of McCarthy's "91-function":

$$f(x) \Leftarrow \text{if } x > 100 \text{ then } x - 10 \text{ else } f(f(x+11)).$$

We will study the evaluation of the above function with the actual parameter 99 substituted for x. Since $99 < 100$, $f(99)$ will be given by $f(f(99+11)) = f(f(110))$. At this point we have two choices.

The first possibility is to view $f(f(110))$ as $f(a)$, where a is the numeric value of $f(110)$. In order to compute $f(f(110))$ we will therefore start by computing $a = f(110)$, which turns out to be 100. This strategy of evaluating all parameters before making a procedure call is "call by value".

* Received by the editors May 23, 1975, and in revised form December 12, 1975.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. The second author is now at Bell Laboratories, Murray Hill, New Jersey 07974.

The computation sequence for $f(99)$ under call by value is as follows:

$$f(99) \rightarrow f(f(110)) \rightarrow f(100) \rightarrow f(f(111)) \rightarrow f(101) \rightarrow 91.$$

The second possibility in computing $f(f(110))$, termed "call by name", is to uniformly substitute the string $f(110)$ for x in the body of the procedure:

$$f(f(110)) \rightarrow \text{if } f(110) > 100 \text{ then } f(110) - 10 \text{ else } f(f(f(110) + 11))$$

In order to perform the test $f(110) > 100$, we will evaluate $f(110)$, (still using call by name):

$$\rightarrow \text{if } 100 > 100 \text{ then } f(110) - 10 \text{ else } f(f(f(110) + 11))$$
$$= f(f(f(110) + 11))$$

At this stage call by name uniformly substitutes the string $f(f(110) + 11)$ for x in the procedure body. Eventually the computation halts with the value 91.

As in the above example, when faced with nested calls like $f(f(f(b)))$, call by value first determines $a = f(b)$ and then $f(f(a))$. Call by value thus leads to the evaluation of an "innermost" function call at each stage. Call by name requires the textual substitution of the actual parameter in the procedure body; in this case $f(f(b))$ is the actual parameter. Call by name thus leads to the evaluation of an "outermost" function call at each stage. The next example shows that outermost evaluation and innermost evaluation may sometimes lead to different results [14].

Example 1.2. Consider the recursive definition

$$f(x, y) \Leftarrow \text{if } x = 0 \text{ then } 1 \text{ else } f(x, f(x - y, y))$$

When started with $f(1, 0)$ innermost evaluation does not halt:

$$f(1, 0) \rightarrow f(0, f(1, 0)) \rightarrow f(0, f(0, f(1, 0))) \rightarrow \cdots.$$

Outermost evaluation computes the value 1.

Example 1.2 demonstrates that outermost evaluation may sometimes compute a value where innermost evaluation diverges. The difference stems from the fact that outermost evaluation computes parameters only as required, while innermost evaluation may embark on a fruitless sequence of evaluations. Outermost evaluation may be inefficient however. In Example 1.1, $f(110)$ will be computed twice by outermost evaluation: once in the test $f(110) > 100$, and again in the "else" part.

In Example 1.2 it is reasonable to suggest that the value or "meaning" of $f(1, 0)$ is 1. By diverging, the call by value rule does not yield an answer agreeing with this meaning, and is therefore incorrect in a well defined sense made precise in Definition 3.1 below.

The central question at this point is: when is a computation rule correct?

The correctness of a rule ϕ may be a trivial consequence of the definition, as in: the meaning of a program is the result of executing the program using computation rule ϕ. The alternative to such an operational definition, employed in this paper, is provided by the fixpoint semantics of programs following Scott [20].

Correctness of computation rules for procedures in which recursion is the only form of iteration has been studied by Vuillemin [23], who gives a condition for determining a class of computation rules called "safe" rules. In this paper we build on the work of Vuillemin. We give necessary and sufficient conditions for a computation rule to be correct. The correctness of safe rules is shown. A condition for correctness more general than the safety condition is given here.

By way of related work, Snyder [21] studies the classes of recursive schemata with call by name and call by value, using a model as in [2]. The class of functionals computed with call by value is properly contained in the class of functionals computed with call by name. The model permits side-effects to take place and is different from the one in this paper.

The work of Rosen [18], Courcelle and Vuillemin [4] and Courcelle, Vuillemin and Nivat [5] on "tree equivalence" of recursion schemes introduces the notion of equivalence between schemes under every interpretation. This "strongest possible" concept of equivalence forms the foundation of § 4 on universally correct computation rules.

2. Recursion schemes: Syntax and semantics. Here we define the general notion of programming language used to discuss the process of recursive definition. A language consists of a *syntax* of formal expressions and a *semantics*, which interprets formal expressions in some chosen universe.

Syntax. The notion of program that we will use is the same as that in [23]:

$$\begin{aligned}\langle\text{scheme}\rangle &::= F(X_1, \cdots, X_n) \Leftarrow \langle\text{term}\rangle\\ \langle\text{term}\rangle &::= A_1 | A_2 | \cdots\\ &\quad | X_1 | X_2 | \cdots | X_n\\ &\quad | G_1(\langle\text{term}\rangle_1, \cdots, \langle\text{term}\rangle_{p_1})\\ &\quad \vdots\\ &\quad | G_k(\langle\text{term}\rangle_1, \cdots, \langle\text{term}\rangle_{p_k})\\ &\quad | F(\langle\text{term}\rangle_1, \cdots, \langle\text{term}\rangle_n)\end{aligned}$$

In the above definition, $A_1, A_2, \cdots$, represent a set of *fixed (nullary) constants*; $G_1, G_2, \cdots, G_k$ represent *base functions* of $p_1, p_2, \cdots, p_k$ arguments, respectively; $X_1, X_2, \cdots, X_n$ represent a set of *formal parameters* of the *function variable* F. An example of a scheme is given in Fig. 2.1. We sometimes avoid subscripts by using $G, H, \cdots$ and $X, Y, \cdots$ for base functions and formal parameters, respectively.

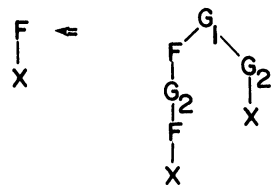


FIG. 2.1. *A scheme*

As in [23], we limit ourselves to a single recursive equation. Extension to systems of mutual recursions is straightforward. For a scheme $F(\bar{X}) \Leftarrow P$, we assume that $F, X_1, \cdots, X_n$ occur in P, and that P is of the form $G(P_1, \cdots, P_n)$, for some G.

A natural way of viewing terms is as labeled trees, as in [19]. Consider a term T. Some nodes in T will be labeled by function symbols F. Let x be such a node with $U_1, U_2, \cdots, U_n$ as subtrees. $T(x \leftarrow P[\bar{U}/\bar{X}])$ is the tree formed by replacing node x in T by the tree for the scheme P modified as follows: the formal parameters $X_1, X_2, \cdots, X_n$ of P are replaced uniformly by $U_1, U_2, \cdots, U_n$, respectively (notation: $P[\bar{U}/\bar{X}]$). A more precise definition may be found in [19]. Since the matching of actual parameters with formals always takes place we will write $T(x \leftarrow P)$ instead of $T(x \leftarrow P[\bar{U}/\bar{X}])$. The replacement can immediately be extended to sets. See [19] or [1] for details. We write $T \rightarrow T(S \leftarrow P)$ (read T is *macroexpanded* to $T(S \leftarrow P)$) if S is a *nonempty* set of nodes in T labeled with function symbols F. This notation allows us to refer to distinct *occurrences* of function "calls" $F(U_1, \cdots, U_n)$ in T.

Semantics. In order to define the meaning of a program scheme S, we will define an interpretation I of the base function symbols G of S over some domain of data D. The meaning of a scheme will then be a mapping from arguments in D to values in D. Following [13], [7] we take as domains complete posets rather than the more restrictive complete lattices [20].

A *complete poset* (cpo) is a system $(D, \leqq, \omega)$ where D is a set, $\leqq$ is a partial order on D and $\omega \in D$, subject to the axioms:

(i) (ω is a zero for D) $\forall x \in D,\ \omega \leqq x$,

(ii) (D is complete) for every linearly ordered subset (chain) $C \subseteq D$, there exists a *least upper bound* lub C such that

$$\forall x \in C,\ x \leqq \text{lub } C,$$

$$(\forall x \in C,\ x \leqq y) \text{ implies } (\text{lub } C \leqq y).$$

A map $g : D^k \rightarrow D$ is called *continuous* if $g(\text{lub } C_1, \cdots, \text{lub } C_k) = \text{lub}\{g(x_1, \cdots, x_k) | \bar{x} \in C_1 \times \cdots \times C_k\}$.

If D, D' are cpo's, $[D \rightarrow D']$ denotes the set of continuous functions from D to D'. This set forms a cpo $([D \rightarrow D'], \leqq, \Omega)$ where $f \leqq g$ iff $\forall x \in D,\ f(x) \leqq g(x)$ and $\Omega = \lambda x.\omega$. Similarly $(D \times D', \leqq \times \leqq', (\omega, \omega'))$ is a cpo under componentwise ordering. These observations allow us to synthesize complex function and product cpo's from given basic cpo's.

The most important fact about continuous functions is the Tarski–Scott theorem:

THEOREM 2.1 [22], [20]. *Any continuous function f from a* cpo D *to itself has a least fixed point x^* characterized by $x^* = \text{lub}_n f^n(\omega)$.*

DEFINITION. An *interpretation* I for a recursive scheme S in a cpo D is a map I assigning to each function symbol G of rank $k > 0$ a continuous function $g^k \in [D^k \rightarrow D]$, with the restriction that $g^k(\omega, \omega, \cdots, \omega) = \omega$; and assigning to each nullary symbol A an element a in D.

The interpretation extends to finite trees (terms) as follows:

$$I(T) = \lambda f.\lambda x. \begin{cases} a & \text{if } T = A, \\ x & \text{if } T = X, \\ g(IT_1, \cdots, IT_k) & \text{if } T = G(T_1, \cdots, T_k), \\ f(IT_1, \cdots, IT_k) & \text{if } T = F(T_1, \cdots, T_k). \end{cases}$$

Notation. Capital letters T will represent syntactic objects (trees), and the corresponding lower case letters t their interpretations. By $t[g]$ we denote the function obtained by substituting g for every function variable in T under I. Most of the time g will be the function Ω.

An *input assignment* (or simply *input*) $\bar{d}$ is an assignment of elements $d_1, \cdots, d_n$ in D to the variables $X_1, \cdots, X_n$. (We choose to conceptually separate "interpretation" from "inputs". An interpretation I applied to a term or a scheme yields a function. Further specifying an input yields a value.)

The scheme $F(\bar{X}) \Leftarrow P$ defines a fixpoint functional recurrence $f = p(f)$. Using the Tarski–Scott theorem, the meaning of a scheme P may now be defined as the least fixpoint f_p of $f = p(f)$, i.e., $f_p = \text{lub}_n\, p^n[\Omega]$. For details consult [17], [11]. A different view of the meaning of a scheme occurs in [12].

3. Computation rules; Correctness. The least fixpoint function f_p is referred to as the *denotational* or *mathematical* semantics of the scheme $F(\bar{X}) \Leftarrow P$. An alternative is available for assigning meaning to schemes: by formal *computation.* We regard schemes as rewriting systems that start with an initial tree of the form $F(\bar{X})$ and repeatedly macroexpand to determine new trees. The value of the scheme is then given by interpreting the macroexpansion sequence of trees. This approach is referred to as *operational* semantics.

A *computation rule* ϕ is an algorithm for selecting some (possibly empty) set of nodes, labeled with the function variable F, from each tree T. For a nonempty set S, macroexpanding T at the nodes in S gives a new tree T'. The choice of the set S may depend on the scheme $F(\bar{X}) \Leftarrow P$, the interpretation I and the input $\bar{d}$. For rule ϕ, the *computation sequence* $\{T_i\}$ *with starting term* T *by* $(P, I, \bar{d})$ is given by:

(i) $T_0 = T$,

(ii) $T_{i+1} = T_i$ if the set S of nodes in T_i selected by ϕ is empty,

(iii) $T_{i+1} = T_i(S \leftarrow P)$ if the set S as in (ii) is nonempty.

Given a computation sequence $\{T_i\}$ for ϕ, we write $T_i \underset{\phi}{\rightarrow} T_{i+1}$. Note that macroexpansion $T_i \rightarrow T_{i+1}$ occurs only when some nonempty set of nodes in T_i is selected for expansion, but $T_i \underset{\phi}{\rightarrow} T_{i+1}$ and $T_i = T_{i+1}$ is quite possible. The advantages of permitting $T_i = T_{i+1}$ are brought out by Example 3.1.

If $\{T_i\}$ is a computation sequence, then the interpreted functions $\{t_i[\Omega]\}$ form a chain [1], [23], whose lub we shall regard as the function "being computed". We will view computation rules as defining such a function. In order to get a value the function is applied to the input data. The *function computed by* ϕ *with starting term* T *under* (P, I) is defined as

$$\phi_p(T) = \lambda \bar{d}.\, \text{lub}_i\, \{t_i[\Omega]\}(\bar{d}),$$

where $\{t_i[\Omega]\}$ is the sequence of interpreted functions for the computation sequence $\{T_i\}$ for rule ϕ, with starting term T by $(P, I, \bar{d})$. The next example explores the influence of $(P, I, \bar{d})$ on a computation rule.

Example 3.1. We will illustrate the above definitions by specifying a computation rule corresponding to "call by value" applied to McCarthy's 91-function with input 99 as in Example 1.1. The function definition in Example 1.1 can be rewritten as the scheme $F(X) \Leftarrow G(X, FFHX)$, where $H(X)$ is interpreted to be $\lambda x.\ x+11$, and $G(X, Y)$ is interpreted to be

$$g = \lambda xy.\ \text{if } x > 100 \text{ then } x - 10 \text{ else } y.$$

The computation rule ϕ with starting term $T_0 = F(X)$ is given in Fig. 3.1. Let $T_0, T_1, \cdots, T_5$ be the terms in (i)–(vi) in Fig. 3.1. The computation rule selects the

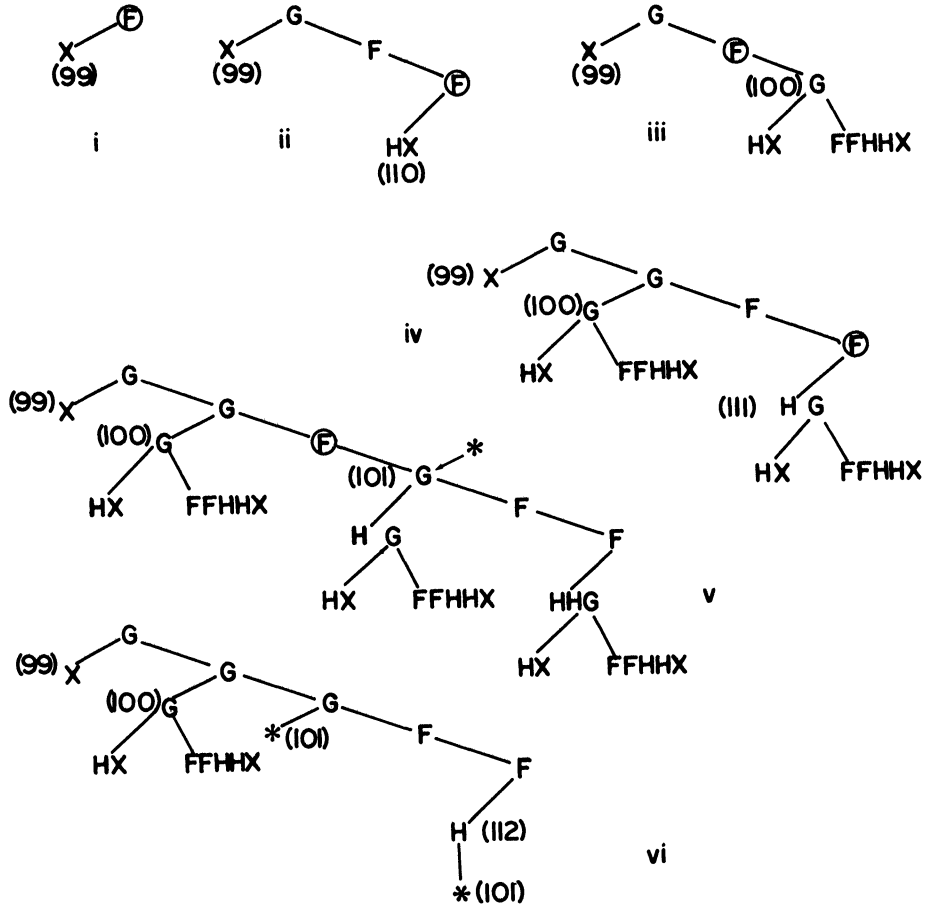


FIG. 3.1. *A sequence of terms modeling evaluation of McCarthy's* 91*-function with input* 99*, under call by value. Once the value of a node n, shown here in parentheses, can be determined by interpreting and supplying the input, no further expansion takes place in the subtree for n.*

circled nodes in T_0–T_4. In T_5 no nodes are selected, so the computation sequence is $T_0, T_1, T_2, T_3, T_4, T_5, T_5, T_5, \cdots$.

Let us relate the sequence of terms in Fig. 3.1 to the following sequence from Example 1.1:

$$f(99) \to f(f(110)) \to f(100) \to f(f(111)) \to f(101) \to 91.$$

The starting term $F(X)$ corresponds to $f(99)$. In T_1, $g(99, y)$ is y, so the computation rule seeks to evaluate $FFHX$. In T_2 when the term is interpreted we will find that $g(110, y) = 100$, so further expansion in the subtree for $G(HX, FFHX)$ is fruitless. In Fig. 3.1 once the value of a subtree can be determined by eventually interpreting and supplying input data, that subtree is ignored. In T_5 the value of the root can be so determined, so no more expansion takes place.

In the above example we saw that a computation rule may be sensitive to the input and the interpretation. In [1], [11], [23], input sensitivities are permitted through the notion of a "simplification rule".

We now turn to particular computation rules that will be used in the sequel.

Example 3.2. (a) The *full substitution rule* σ chooses to macroexpand all occurrences of F in any term. This simultaneous expansion is done "bottom-up" on a term; see [1, pp. 87–88] or [10] for details. The computation sequence of $F(X)$ by $F(X) \Leftarrow G(FHFX, HX)$ is

$$F(X) \underset{\sigma}{\to} G(FHFX, HX) \underset{\sigma}{\to} G(G(FHFHG(FHFX, HX), HHG(FHFX, HX)), HX) \underset{\sigma}{\to} \cdots.$$

(b) The *parallel outermost rule* π chooses to macroexpand all nodes labeled F with no ancestor nodes that are labeled F. For $F(X)$ with the scheme from (a), as in Fig. 3.2 we get

$$F(X) \underset{\pi}{\to} G(FHFX, HX) \underset{\pi}{\to} G(G(FHFHFX, HHFX), HX) \underset{\pi}{\to} \cdots.$$

(c) The *weird rule* ψ operates on a term T as follows: it traces the paths of T from root to leaf, looking for the outermost chain of uninterrupted occurrences of F's, and picks the lowest F in all such chains. In $HFFX$ the inner F is chosen. In $G(FX, HFFHFX)$, the first and third F's are chosen, etc. The computation sequence of FFX by $F(X) \Leftarrow G(X, FHX)$ is

$$FFX \underset{\psi}{\to} FG(X, FHX) \underset{\psi}{\to} G(G(X, FHX), FHG(X, FHX)).$$

At this point all chains of outermost F's are single nodes. Since the scheme does not introduce any such chains with more than one node, from this point on ψ proceeds as π would.

A number of other computation rules are defined in [23].

A general question of *validity* of rules now arises: for what rules ϕ is it true that $\phi_p(F(\bar{X})) = f_p$?

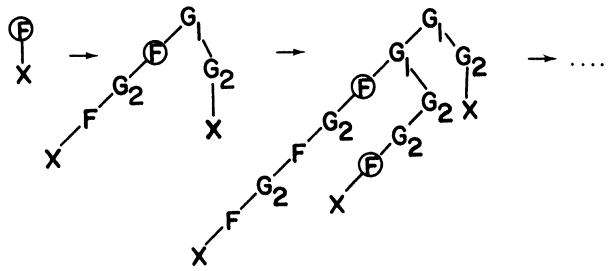


FIG. 3.2. Parallel outermost, π. *Expand all nodes labeled F with no ancestors labeled F. Such F's are circled. The scheme is as in Fig.* 2.1.

DEFINITION 3.1. A rule ϕ is said to be *correct for* $(P, I, \bar{d})$ if for every term T

$$\phi_p(T)(\bar{d}) = t[f_p](\bar{d}).$$

A rule ϕ is *correct for* (P, I) if for all $\bar{d}$, ϕ is correct for $(P, I, \bar{d})$.

Our purpose, as in [23], is to characterize correct rules. Fundamental facts are given by

THEOREM 3.1 [1]. *For every scheme P, interpretation I, and rule* ϕ,

$$\phi_p(T) \leqq t[f_p] \quad \textit{for all } T,$$

i.e., the least fixpoint function is at least as well defined as any computed function.

THEOREM 3.2 [23]. *The full substitution rule* σ *is correct for every scheme P and every interpretation I.*

Proof. Let $F(\bar{X}) \Leftarrow P$ be the scheme, and let $\bar{d}$ be arbitrary. For arbitrary term T, let $T = T_0, T_1, \cdots$ be the σ-computation sequence with starting term T by $(P, I, \bar{d})$.

Let $p^0[f] = f$ be the identity functional, $p^1 = p$ and p^i be the i-fold composition of p. Structural induction on R and the definition of σ establish

$$\text{if } R \underset{\sigma}{\rightarrow} S \text{ then } s[f] = r[p[f]] \tag{1}$$

and since $T_i \underset{\sigma}{\rightarrow} T_{i+1}$ we have $\forall i$

$$t_i[f] = t_0[p^i[f]]. \tag{2}$$

Then

$$\begin{aligned} \sigma_p(T_0) &= \text{lub}_i\, t_i[\Omega] \\ &= \text{lub}_i\, t_0[p^i[\Omega]] \qquad \text{(by (2))}, \\ &= t_0[\text{lub}_i\, p^i[\Omega]] \qquad \text{(by continuity)}, \\ &= t_0[f_p]. \end{aligned}$$

since $\text{lub}_i\, p^i[\Omega] = f_p$ by the Tarski–Scott theorem. Thus $\sigma_p(T) = t[f_p]$ for all T. □

The above result suggests that we look for rules ϕ that are correct under all interpretations I. Such rules are called *universally correct.*

DEFINITION 3.2. A rule ϕ is called *universally correct for P* if for all I, ϕ is correct for (P, I).

Section 4 below develops a characterization of all universally correct rules. In § 5 we examine rules which are correct under particular interpretations, or classes of interpretations.

Consider the set of all finite trees $L = \{T' | T \xrightarrow{*} T'\}$ obtainable from T by the macroexpansion relation $\xrightarrow{*}$ defined by a scheme P.

THEOREM 3.3 [23]. *The structure* $(L, \xrightarrow{*}, T)$ *is a lattice* (*the computation lattice of T by P*), *with zero T.*

Each computation rule ϕ defines a chain in this lattice. Note that the lattice is not generally complete, since L does not contain infinite trees as elements.

4. Universal correctness and syntactic dominance. Theorem 3.1 tells us that any computation rule ϕ is "sound", i.e., that it computes a function $\phi_p F(\bar{X})$ dominated by f_p. A correct ϕ is one which is also "adequate", i.e., which actually achieves f_p in the limit.

To show that a computation sequence $T_0, T_1, \cdots$ computes f_p, it is enough to show that the elements of $\{t_i[\Omega]\}$ "eventually dominate" some "yardstick" sequence $\{s_i[\Omega]\}$ which is known to have $\text{lub}_i\, s_i[\Omega] = f_p$ (i.e., to be correct). One such sequence is the computation sequence by full substitution $p^i[\Omega]$ (Theorem 3.2). A much more convenient yardstick to use, however, is the parallel outermost computation sequence. We shall show (Theorem 4.1) that the parallel outermost rule π is universally correct, and then show that it may be used to test whether an arbitrarily given rule ϕ is universally correct (Definition 4.2, Theorem 4.2). It turns out that any universally correct rule ϕ must "syntactically dominate" π (Definition 4.2).

Here we construct a special poset which plays the role of a Herbrand universe in the sequel [18], [3], [4]. Let H consist of all finite and infinite trees built up from symbols $\{G_i\}$, $\{A_i\}$, $\{X_i\}$ and a new nullary symbol $\perp$. For $U, V \in H$ define $U \leqq V$ if there is a set N of mutually independent nodes in V such that $U = V(N \leftarrow \perp)$. (Two nodes are independent when neither is an ancestor of the other). Then $(H, \leqq, \perp)$ forms a complete poset. The *Herbrand interpretation* I_H of a scheme assigns meaning $A_i \in H$ to the symbols A_i, $X_i \in H$ to X_i and to G_i the function $g_i(T_1, \cdots, T_k) =$ if $T_1 = T_2 = \cdots = T_k = \perp$ then $\perp$ else $G_i(T_1, \cdots, T_k)$ in $[H^k \to H]$.

DEFINITION 4.1. Let R be a finite or infinite tree over $\{G_i\} \cup \{A_i\} \cup \{X_i\} \cup \{F_i\}$. μR is the tree in H formed from R by replacing all outermost function variables by $\perp$, making them leaves.

Extend all interpretations I by setting $I(\perp) = \omega$.

LEMMA 4.1. *For all interpretations I, all trees R and all* $\bar{d}$,

$$I(R)[\Omega](\bar{d}) = I(\mu R)(\bar{d}).$$

Proof. The proof is immediate from the definition of interpretation. □

THEOREM 4.1. *Parallel outermost* π *is a correct computation rule, for all I, P and* $\bar{d}$; *i.e.,* π *is universally correct for all P.*

Proof. Let S_0 be a given starting term and let $S_0, S_1, \cdots$ be the full substitution computation sequence: $S_i \underset{\sigma}{\to} S_{i+1}$. Let $T_0 = S_0$, $T_1, T_2, \cdots$ be the parallel

outermost sequence: $T_i \underset{\pi}{\rightarrow} T_{i+1}$. We show that for each S_i there exists a T_j such that $\mu S_i \leqq \mu T_j$. It then follows from the above lemma that $\forall i\ \exists j. s_i[\Omega](\bar{d}) \leqq t_j[\Omega](\bar{d})$ and $\pi_p S_0(\bar{d}) = \text{lub}_j\, t_j[\Omega](\bar{d}) = s_0[f_p](\bar{d})$.

Let l be the depth of tree μS_i. If $S_0 \overset{j}{\underset{\pi}{\rightarrow}} T_j$, then the node of minimal depth labeled $\perp$ in μT_j has depth $\geqq j$. Consider T_{l+1}. We will show that $\mu S_i \leqq \mu T_{l+1}$.

Since the set of trees derived from S_0 under macroexpansion is a lattice under $\overset{*}{\rightarrow}$ (Theorem 3.3),[1] there is some R such that $S_i \overset{*}{\rightarrow} R$ and $T_{l+1} \overset{*}{\rightarrow} R$. Thus $\mu T_{l+1} \leqq \mu R$ and $\mu S_i \leqq \mu R$. As a consequence any node n in μS_i labeled with a letter in $\{G_i, A_i, X_i\}$ is of depth $\leqq l$ and so has the same label in μT_{l+1}. All other nodes of μS_i are labeled $\perp$. Hence $\mu S_i \leqq \mu T_{l+1}$. □

The next definition gives a criterion which we show characterizes all universally correct rules (Theorems 4.2, 4.3).

DEFINITION 4.2. Given a computation rule ϕ and a scheme P, *ϕ syntactically dominates* the parallel outermost rule π for P if

$$\forall T.\ T \underset{\pi}{\rightarrow} U \quad \text{implies} \quad \exists V.\ T \overset{*}{\underset{\phi}{\rightarrow}} V \quad \text{and} \quad U \overset{*}{\rightarrow} V.$$

Remark. A diagrammatic representation [19] of the above definition is given in Fig. 4.1. ϕ syntactically dominates π if every outermost occurrence of F in T eventually gets macroexpanded by ϕ.

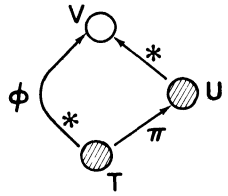


FIG. 4.1. *Computation rule ϕ* syntactically dominates *π if for all terms T and U, if $T \underset{\pi}{\rightarrow} U$, then there exists a term V such that $T \overset{*}{\underset{\phi}{\rightarrow}} V$ and $U \overset{*}{\rightarrow} V$.*

LEMMA 4.2. *$\forall T, U, V.\ T \underset{\pi}{\rightarrow} U\ \&\ T \overset{*}{\rightarrow} V$ implies $\exists W.\ V \underset{\pi}{\rightarrow} W\ \&\ U \overset{*}{\rightarrow} W$.*

Proof. In the language of [19], $\underset{\pi}{\rightarrow}$ commutes with $\rightarrow$.

For Fig. 4.2, it is enough to show that if $T \underset{\pi}{\rightarrow} U$ and $T \rightarrow V$, then there exists W such that $V \underset{\pi}{\rightarrow} W$ and $U \overset{*}{\rightarrow} W$, from which the result follows. This assertion is proved by structural induction on the term T.

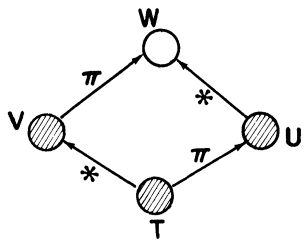


FIG. 4.2. *Parallel outermost π, has a useful property. If $T \underset{\pi}{\rightarrow} U$, then for all V such that $T \overset{*}{\rightarrow} V$, it must be true that if $V \underset{\pi}{\rightarrow} W$, then $U \overset{*}{\rightarrow} W$.*

[1] Actually, we only need the fact that $(L, \overset{*}{\rightarrow}, T)$ is a "Church–Rosser system" [19].

Basis. T is either A or X. From the definition of macroexpansion in § 2, at least one F-labeled node is expanded if $T \to V$. Since neither A nor X have such a node, $T \to V$ must be false and the assertion vacuously true.

Inductive step. There are two cases depending on whether the root of T is labeled F or G.

Case $T = G(T_1, \cdots, T_k)$. If $T \to U$, then U must be of the form $G(U_1, \cdots, U_k)$, where $T_i \underset{\pi}{\to} U_i$, from the properties of rule π. Also $V = G(V_1, \cdots, V_k)$ where $T_i \to V_i$ or $T_i = V_i$, $1 \leqq i \leqq k$. From the inductive hypothesis, for $1 \leqq i \leqq k$,

$$\exists W_i.\ V_i \underset{\pi}{\to} W_i \ \& \ \exists n_i.\ U_i \overset{n_i}{\longrightarrow} W_i.$$

But then $U = G(\bar{U}) \overset{n}{\to} G(\bar{W})$, where $n = \max_i n_i$, and $V = G(\bar{V}) \underset{\pi}{\to} G(\bar{W})$. Letting $W = G(\bar{W})$, this case follows.

Case $T = F(T_1, \cdots, T_n)$. Since $T \underset{\pi}{\to} U$ it must be true that $U = P[\bar{T}/\bar{X}]$. Let $T \to V$ by macroexpansion at some nonempty set S of nodes in T.

Subcase a. The outermost F in T is not in S. Then $V = F(V_1, \cdots, V_n)$, where $T_i \to V_i$ or $T_i = V_i$ for $1 \leqq i \leqq n$. In this case $U = P[\bar{T}/\bar{X}] \to P[\bar{V}/\bar{X}]$ and $V = F(\bar{V}) \underset{\pi}{\to} P[\bar{V}/\bar{X}]$. Choosing $W = P[\bar{V}/\bar{X}]$ proves this subcase.

Subcase b. The outermost F in T is in S. Then $V = P[\bar{V}/\bar{X}]$, where $T_i \to V_i$ or $T_i = V_i$, $1 \leqq i \leqq n$. Define W by $V \underset{\pi}{\to} W$. Then $V = P[\bar{V}/\bar{X}] \underset{\pi}{\to} W$, and $U = P[\bar{T}/\bar{X}] \to P[\bar{V}/\bar{X}] = V \underset{\pi}{\to} W$. Clearly $U \overset{*}{\to} W$, proving the subcase and the case $T = F(\bar{T})$.

By structural induction on T, the lemma is proved. □

We are now in a position to characterize universally correct computation rules.

THEOREM 4.2. *If ϕ syntactically dominates π for P, then ϕ is universally correct for P.*

Proof. Let $T_0, T_1, T_2, \cdots$ be the parallel outermost computation sequence. Refer to Fig. 4.3. From Lemma 4.2 and the hypothesis,

$$\forall T_i, V_i.\ T_i \underset{\pi}{\to} T_{i+1} \ \& \ T_i \overset{*}{\to} V_i$$

implies

$$\exists W_{i+1}.\ V_i \underset{\pi}{\to} W_{i+1} \ \& \ T_{i+1} \overset{*}{\to} W_{i+1}$$

and from syntactic dominance of π

$$\forall V_i, W_{i+1}.\ V_i \underset{\pi}{\to} W_{i+1} \quad \text{implies} \quad \exists V_{i+1}.\ V_i \overset{*}{\underset{\phi}{\to}} V_{i+1} \ \& \ W_{i+1} \overset{*}{\to} V_{i+1}.$$

Since $\forall T_i\ \exists V_j.\ T_i \overset{*}{\to} V_j$ where the $\{V_j\}$ are a subsequence of the ϕ-computation sequence $\{Z_i\}$, we conclude that $\text{lub}_i\ t_i[\Omega] \leqq \text{lub}_i\ z_i[\Omega]$ and $\phi_p T_0 = \pi_p T_0 = t[f_p]$. □

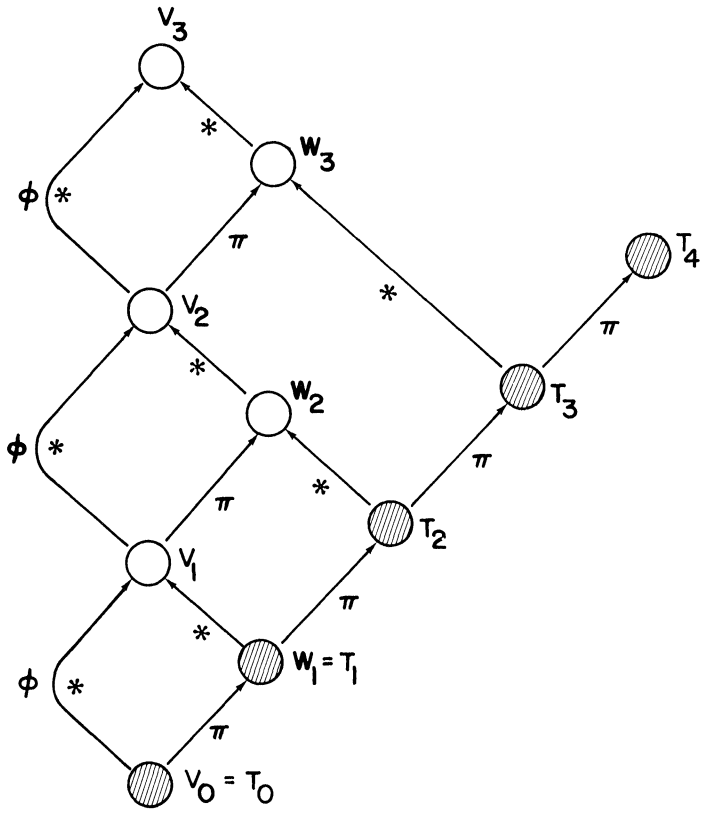


FIG. 4.3. $\{T_i\}$ *is the parallel outermost computation sequence. We construct the sequence* $\{V_i\}$ *which syntactically majorizes* $\{T_i\}$.

The next theorem proves the converse of Theorem 4.2. There is one issue we must confront, however. A certain small set of schemes P will be correctly computed by any rule whatever under all interpretations, e.g., $F(X) \Leftarrow G(F(X), F(X))$.

DEFINITION 4.3. A node n in a term T is *hopeless* if every leaf in the subtree of n in μT is $\perp$. Let $F(\bar{X}) \Leftarrow P$ be a scheme and let $F(\bar{X}) = T_0, T_1, \cdots$ be the π-computation by P. P is *hopeless* if the root of every μT_i is hopeless.

The hopeless P are exactly those for which $f_p = \Omega$ in every interpretation, since $f_p = \Omega$ in the Herbrand interpretation. It is trivial to decide whether n or P is hopeless [18].

THEOREM 4.3. *Let P be a scheme which is not hopeless. If ϕ is universally correct for P, then ϕ syntactically dominates π for P.*

Proof. Let $P = G(P_1, \cdots, P_k)$ be assumed not hopeless. Suppose ϕ does not syntactically dominate π for this P. Then we find a term U and an interpretation I in which $\phi_p U \neq u[f_p]$.

Since ϕ does not syntactically dominate π, there exists a term U in which some outermost F occurrence (at, say, node n) is never expanded by ϕ in the sequence $U = U_0 \underset{\phi}{\rightarrow} U_1 \underset{\phi}{\rightarrow} U_2 \cdots$.

Then the terms $\mu U_0 \leqq \mu U_1 \leqq \cdots$ all have $\perp$ rooted at node n. Let I_H be the Herbrand interpretation of the scheme. Now $I_H U_0 \leqq I_H U_1 \leqq \cdots$ all have $\perp$ rooted at some ancestor of node n (some ancestor of n might be hopeless), and so does $\text{lub}_i\, I_H U_i = I_H U^*$.

Let the subtree at node n in U_0 be $F(\bar{V})$. The notion of hopeless P is defined with starting term $F(\bar{X})$. By structural induction on $\bar{V}$ and the fact that P is not hopeless,

$$\forall \bar{V} \exists i.\ F(\bar{V}) \xrightarrow[\pi]{i} W, \quad \text{where } I_H W \neq \perp.$$

Consider the π sequence from $U = T_0 \underset{\pi}{\rightarrow} T_1 \underset{\pi}{\rightarrow} \cdots$. The F at n is eventually macroexpanded in this sequence (say at T_j).

Thus the trees $I_H T_j \leqq I_H T_{j+1} \leqq \cdots$ have a limit tree $I_H T^*$ with a non-$\perp$ tree rooted at n. Since $I_H T^* \neq I_H U^*$, rule ϕ cannot be correct. □

Syntactic dominance is a necessary and sufficient criterion for universal correctness. We give here a sequence of two sufficient criteria for universal correctness. Call a term *constant* if it has no node labeled F.

COROLLARY 4.1. *Let P be given. Let the rule ϕ be such that*

(i) *for all base functions G and terms $\bar{T} = (T_1, \cdots, T_k)$, $\bar{S} = (S_1, \cdots, S_k)$, $G(\bar{T}) \underset{\phi}{\rightarrow} G(\bar{S})$ implies $T_i \overset{+}{\underset{\phi}{\rightarrow}} S_i$;*

(ii) *for all terms $F(\bar{T})$ there exists a base function G and terms $\bar{S}$ such that $F(\bar{T}) \overset{*}{\underset{\phi}{\rightarrow}} G(\bar{S})$;*

(iii) *for all nonconstant terms T, there exists $S \neq T$ such that $T \underset{\phi}{\rightarrow} S$.*

Then ϕ syntactically dominates π, the parallel outermost rule, for P.

Proof. We must show that, for all T, if $T \underset{\pi}{\rightarrow} U$ then $\exists V.\ T \overset{*}{\underset{\phi}{\rightarrow}} V$ and $U \overset{*}{\rightarrow} V$. The proof is by structural induction on T.

Case T is constant. Then T cannot be macroexpanded, so $T = U$ and we choose $V = U$.

Case $T = F(\bar{T})$. Then by (ii), $T = F(\bar{T}) \overset{*}{\underset{\phi}{\rightarrow}} G(\bar{S})$. Since only the outermost F in T is expanded by π, if $F(\bar{T}) \underset{\pi}{\rightarrow} U$ then $U \overset{*}{\rightarrow} G(\bar{S})$. Let $V = G(\bar{S})$.

Case $T = G(\bar{T})$ and T is not constant. Since $T \underset{\pi}{\rightarrow} U$ it follows that $U = G(\bar{U})$ and for $1 \leqq i \leqq k$, $T_i \underset{\pi}{\rightarrow} U_i$. The inductive hypothesis applied to T_i implies $\exists V_i$ and integers $j(i)$ such that $T_i \overset{j(i)}{\underset{\phi}{\longrightarrow}} V_i$ and $U_i \overset{*}{\rightarrow} V_i$. Let l be the largest of the $j(i)$. We will construct terms $\bar{S}$ such that $G(\bar{T}) \overset{l}{\underset{\phi}{\rightarrow}} G(\bar{S})$ and $G(\bar{V}) \overset{*}{\rightarrow} G(\bar{S})$.

Since $G(\bar{T})$ is not constant, by (iii) there exist terms $\bar{S}$ such that $G(\bar{T}) \overset{l}{\underset{\phi}{\rightarrow}} G(\bar{S})$ via the sequence $G(\bar{T}) = G(\bar{T}^0) \underset{\phi}{\rightarrow} G(\bar{T}^1) \underset{\phi}{\rightarrow} \cdots \underset{\phi}{\rightarrow} G(\bar{T}^l) = G(\bar{S})$. From hypothesis (i), we know that $T_i = T_i^0 \overset{+}{\underset{\phi}{\rightarrow}} T_i^1 \overset{+}{\underset{\phi}{\rightarrow}} \cdots \overset{+}{\underset{\phi}{\rightarrow}} T_i^l$. Recall that $T_i \overset{j(i)}{\underset{\phi}{\rightarrow}} V_i$ and $j(i) \leqq l$. It follows that $V_i \overset{*}{\rightarrow} T_i^l$ for all i, $1 \leqq i \leqq k$.

Clearly $G(\bar{V}) \overset{*}{\rightarrow} G(\bar{S})$. Also $G(\bar{U}) \overset{*}{\rightarrow} G(\bar{V})$. Thus $G(\bar{U}) \overset{*}{\rightarrow} G(\bar{S})$. Since $G(\bar{T}) \underset{\pi}{\rightarrow} G(\bar{U})$, and $G(\bar{T}) \overset{*}{\underset{\phi}{\rightarrow}} G(\bar{S})$, we have the result with $V = G(\bar{S})$. □

Remark. In the hypothesis $T_i \xrightarrow[\phi]{+} S_i$, the "+" is crucial. Replacing + with "*" makes the statement false. Consider the rule *lo* which expands the leftmost outermost occurrence of F, e.g.,

$$G(F(X), F(X)) \underset{lo}{\rightarrow} G(G(X, F(X)), F(X)) \underset{lo}{\rightarrow} G(G(X, G(X, F(X))), F(X)) \rightarrow \cdots.$$

Then *lo* satisfies (i) with * for +. Yet it is known that *lo* is not universally correct [23].

Example 4.1. Consider the weird rule ψ of Example 3.2(c). It is easy to see from Corollary 4.1 that ψ syntactically dominates π, and so is universally correct. Note that ψ is not a safe rule (Definition 5.6, below) for most interpretations, and thus cannot be proved universally correct by the methods of Vuillemin [23].

Corollary 4.1 gives a "top-down" criterion for the performance of ϕ: condition (i) requires that if $T \underset{\phi}{\rightarrow} U$, then the subterms of T derive the subterms of U under ϕ. By contrast the next corollary gives a "bottom-up" criterion: the action of ϕ on subterms is required to extend to larger, containing terms.

COROLLARY 4.2. *Let P be given. Let the rule ϕ be such that:*

(i) *for all base functions G and terms $\bar{T} = (T_1, \cdots, T_k)$, $T_i \underset{\phi}{\rightarrow} S_i$ for $1 \leqq i \leqq k$ implies there exist terms $\bar{W} = (W_1, \cdots, W_k)$ with $S_i \xrightarrow{*} W_i$ and $G(\bar{T}) \underset{\phi}{\xrightarrow{*}} G(\bar{W})$;*

(ii) *for all terms $\bar{T}$ there exists a base function G and terms $\bar{S}$ such that $F(\bar{T}) \underset{\phi}{\xrightarrow{*}} G(\bar{S})$.*

Then ϕ syntactically dominates π for P.

Proof. We use a straightforward structural induction on T such that $T \underset{\pi}{\rightarrow} U$. Only the case $T = G(\bar{T})$ differs from Corollary 4.1. Details are omitted. □

Remark. The criterion of Corollary 4.1 implies that of Corollary 4.2. We show that Corollary 4.1(i) implies Corollary 4.2(i). Assume $G(\bar{T}) \underset{\phi}{\rightarrow} G(\bar{S})$ implies $T_i \underset{\phi}{\xrightarrow{+}} S_i$. If now $T_i \underset{\phi}{\rightarrow} R_i$, then $R_i \xrightarrow{*} S_i$ and $G(\bar{T}) \underset{\phi}{\xrightarrow{*}} G(\bar{S})$, showing 4.2(i) with $\bar{W} = \bar{S}$.

The next example shows that the second criterion (Corollary 4.2) is strictly weaker than the first (Corollary 4.1).

Example 4.2. Consider a modified parallel outermost rule which operates as follows: on nonconstant terms of the form $T = G(\bar{T})$, θ chooses for expansion the leftmost F among those closest (in path length) to the root. (This is quite different from picking the leftmost outermost F). On other terms, θ behaves like π.

Now θ satisfies Corollary 4.2(i) since $T_i \underset{\theta}{\rightarrow} S_i$ implies $G(\bar{T}) \underset{\theta}{\xrightarrow{*}} G(\bar{S})$. However $G(F(\bar{U}), F(\bar{V})) \underset{\theta}{\rightarrow} G(P[\bar{U}/\bar{X}], F(\bar{V}))$ does not imply $F(\bar{V}) \underset{\theta}{\xrightarrow{+}} F(\bar{V})$ (which is false). Thus θ does not satisfy Corollary 4.1(i).

The next example shows that syntactic dominance is strictly weaker than the criterion of Corollary 4.2.

Example 4.3. Consider the mixed rule μ which operates as follows: on terms of the form $G(\cdots)$, μ performs like π, the parallel outermost rule; but on terms of the form $F(\cdots)$, μ performs like σ, the full substitution rule.

Let the scheme be $FX \Leftarrow G(X, FFX)$. Let $G(T_1, T_2) = G(X, FFX)$. Clearly $T_2 = FFX \underset{\mu}{\rightarrow} G(G(X, FFX), FFG(X, FFX)) = S_2$. We need to show that there is no $\bar{W}$ such that $G(\bar{T}) \overset{*}{\underset{\mu}{\rightarrow}} G(\bar{W})$ and $S_2 \overset{*}{\rightarrow} W_2$. Consider the μ-sequence from $G(\bar{T})$: $G(\bar{T}) \underset{\mu}{\rightarrow} G(X, G(FX, FFX)) \underset{\mu}{\rightarrow} \cdots$. It is characterized by the property: no G is below any F in any term. Now consider any term R such that $S_2 \overset{*}{\rightarrow} R$. Any such R is characterized by the property: at least one F dominates a G. Hence there is no $\bar{W}$, $G(\bar{T}) \overset{*}{\underset{\mu}{\rightarrow}} G(\bar{W})$ with $S_2 \overset{*}{\rightarrow} W_2$. So μ does not satisfy 4.2(i). Yet μ obviously syntactically dominates π.

We have shown that for any computation rule to be correct for a given program under all interpretations and for all inputs, it must syntactically dominate the parallel outermost rule π. Common computation rules, like call by name and call by value, depend on both the interpretation and the input data, choosing not to macroexpand function symbols under some circumstances (Example 3.1). We will study such computation rules in the following section.

5. Security: An interpretation-dependent correctness criterion. Requiring a rule ϕ to be correct for all interpretations imposes a severe restriction on ϕ, as we have seen from Theorem 4.3. One is in fact usually interested in ϕ's which are correct for one particular interpretation I and input values $\bar{d}$ restricted to a certain set. Thus we need a correctness criterion which is semantic, i.e., which uses the properties of the interpretation I and input data $\bar{d}$ in the test for correctness.

One such criterion is given in Definition 5.5. Theorem 5.1 shows that rules meeting this criterion for $(I, \bar{d})$ exactly characterize the correct rules for $(I, \bar{d})$. Theorem 5.2 shows the correctness of Vuillemin's "safety" criterion [23].

These results require us to think carefully about outermost occurrences of F's and whether they are expanded. Given a tree T we shall think of nodes as existing quite apart from their labels; indeed they can exist whether or not they have any label at all in that tree. For a precise treatment, see Rosen [19]. We need the following definitions.

DEFINITION 5.1. For a term R, let $\tau(R)$ be the set of nodes of R that are labeled with outermost occurrences of F.

DEFINITION 5.2. Let $R \overset{*}{\underset{\phi}{\rightarrow}} S$. Let $\rho(R, S)$ be the subset of nodes in $\tau(R)$ which remain unexpanded by ϕ in S.

Note that no copies are made of elements of $\tau(R)$ since they are all outer F's.

DEFINITION 5.3. Define

$$\rho(R) = \bigcap_{R \overset{*}{\underset{\phi}{\rightarrow}} S} \rho(R, S),$$

the set of outermost nodes of R which are never expanded under ϕ. Note that if $R_0 \underset{\phi}{\rightarrow} R_1 \underset{\phi}{\rightarrow} R_2 \underset{\phi}{\rightarrow} \cdots$, then $\rho(R_0) \subseteq \rho(R_1) \subseteq \rho(R_2) \subseteq \cdots$.

A node n *dominates* node m if $n = m$ or n is an ancestor of m.

DEFINITION 5.4. Let R be a term, let N be a subset of independent nodes of R. By $R \underset{\pi[N]}{\longrightarrow} S$ is meant that S is obtained from R by parallel outermost restricted to the forest dominated by nodes in N.

Given a rule ϕ and a starting term R_0, two distinct cases are possible. Either there is an infinite sequence $R_0 \underset{\phi}{\rightarrow} R_1 \underset{\phi}{\rightarrow} R_2 \underset{\phi}{\rightarrow} \cdots$ of ϕ-successors (π is such a rule), or else for some R_k, R_k has no ϕ-successor (ϕ "chooses" the empty set of nodes). Then $\tau(R_k) = \rho(R_k)$. In the latter case, it is true that $R_k \underset{\phi}{\rightarrow} R_k$. Note this does *not* imply that $\underset{\phi}{\rightarrow}$ is reflexive: it is reflexive only on "stuck" terms. From these observations we see that each R_0 defines an unique infinite sequence $\{R_j\}$ by $R_0 \overset{j}{\underset{\phi}{\rightarrow}} R_j$.

The main semantic criterion is now definable.

DEFINITION 5.5. Let ϕ be a rule and let $(P, I, \bar{d})$ be a triple of program P, interpretation I and input $\bar{d}$. For each starting term R_0 and all j, define a sequence $\{S_j\}$ by:

$$\text{if } R_0 \overset{j}{\underset{\phi}{\rightarrow}} R_j \text{ then } R_j \xrightarrow[\pi[\rho(R_j)]]{j} S_j.$$

Rule ϕ is called *secure for* $(P, I, \bar{d})$ if for all terms R_0

$$\text{lub}_j\, s_j[\Omega](\bar{d}) \leqq \text{lub}_j\, r_j[\Omega](\bar{d}).$$

In other words, the limit of the terms formed by expanding the subtrees under the outer F's in R_j that will never be expanded contains no more information than the limit of the sequence $\{R_j\}$.

Two useful lemmas follow immediately from the above definitions, and will be stated without proof.

LEMMA 5.1. *Let R, S be terms and let M, N be sets of nodes such that $M \subseteq \tau R$ and $N \subseteq \tau S$. If $R \overset{*}{\rightarrow} S$, $R \xrightarrow[\pi[M]]{i} R'$, $S \xrightarrow[\pi[N]]{i} S'$ and $M \subseteq N$, then $R' \overset{*}{\rightarrow} S'$.*

LEMMA 5.2. *Let R, S, T be terms and $M \subseteq \tau R$ a set of nodes. If $R \xrightarrow[\pi[M]]{i} S$, $R \xrightarrow[\pi[M]]{j} T$ and $i < j$, then $S \xrightarrow[\pi[M]]{+} T$.*

We are now ready for the main result of this section.

THEOREM 5.1. *Given program P, interpretation I and input $\bar{d}$, a rule ϕ is correct for $(P, I, \bar{d})$ if and only if it is secure for $(P, I, \bar{d})$.*

Proof (secure $\Rightarrow$ correct). Let $R_0 = T_0$ be the starting term, and define sequences $\{R_j\}$, $\{S_j\}$, and $\{T_j\}$ by $R_0 \overset{j}{\underset{\phi}{\rightarrow}} R_j$, $R_j \xrightarrow[\pi[\rho R_j]]{j} S_j$ and $T_0 \overset{j}{\underset{\pi}{\rightarrow}} T_j$.

CLAIM. *For every T_j there is an $S_{i(j)}$ such that $T_j \overset{*}{\rightarrow} S_{i(j)}$.*

The claim is proved by induction on j.

Basis $j = 1$. Note that $R_0 = T_0$. By definition there is a term $R_{i(1)}$, $i(1) \geqq 1$, such that $R_0 \underset{\phi}{\rightarrow} R_{i(1)}$ and all nodes of $\tau R_0 - \rho R_0$ have been expanded in $R_{i(1)}$. Let S be such that $R_{i(1)} \xrightarrow[\pi[\rho R_0]]{} S$. All nodes in τR_0 have been expanded in S, so $T_1 \overset{*}{\rightarrow} S$.

Since $i(1) \geqq 1$ then $R_{i(1)} \xrightarrow[\pi[\rho R_{i(1)}]]{+} S_{i(1)}$. Since $R_{i(1)} \xrightarrow[\pi[\rho R_0]]{} S$ and $\rho R_0 \subseteq \rho R_{i(1)}$, it follows that $S \overset{*}{\rightarrow} S_{i(1)}$ and so $T_1 \overset{*}{\rightarrow} S_{i(1)}$.

Induction step. Refer to Fig. 5.1. By the induction hypothesis $T_j \overset{*}{\rightarrow} S_{i(j)}$. Consider the nodes in $\tau R_{i(j)} - \rho R_{i(j)}$. From the definition of $\rho R_{i(j)}$ there exists a term $R_{i(j+1)}$ such that $R_{i(j)} \overset{+}{\underset{\phi}{\rightarrow}} R_{i(j+1)}$ and all nodes of $\tau R_{i(j)} - \rho R_{i(j)}$ have been expanded in $R_{i(j+1)}$.

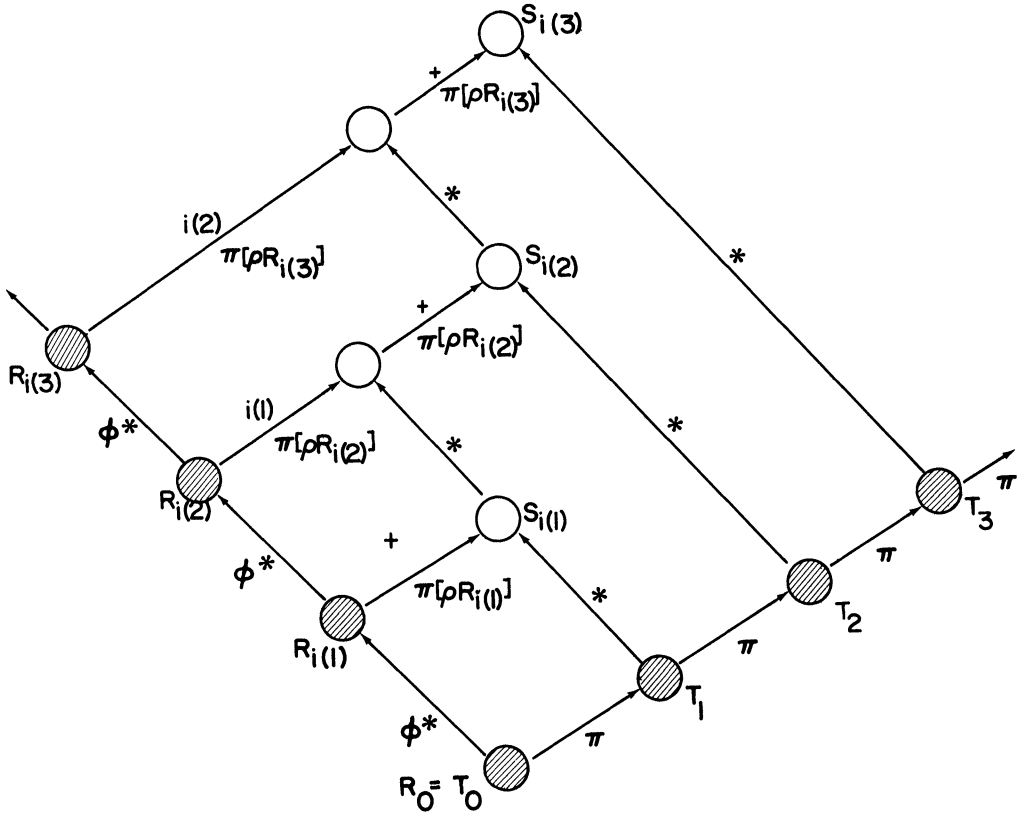


FIG. 5.1. $\{T_i\}$ *is the parallel outermost computation sequence, and* $\{R_i\}$ *is the sequence for* ϕ. *We determine a new sequence* $\{S_i\}$ *which syntactically majorizes* $\{T_i\}$, *but from security of* ϕ, *contains no more information than* $\{R_i\}$.

Consider $S_{i(j+1)}$ such that $R_{i(j+1)} \xrightarrow[\pi[\rho R_{i(j+1)}]]{i(j+1)} S_{i(j+1)}$. We will show that $T_{j+1} \xrightarrow{*} S_{i(j+1)}$.

Define S such that $R_{i(j+1)} \xrightarrow[\pi[\rho R_{i(j+1)}]]{i(j)} S$. Since $R_{i(j)} \xrightarrow{*} R_{i(j+1)}$, $R_{i(j)} \xrightarrow[\pi[\rho R_{i(j)}]]{i(j)} S_{i(j)}$ and $\rho R_{i(j)} \subseteq \rho R_{i(j+1)}$, then by Lemma 5.1

$$S_{i(j)} \xrightarrow{*} S. \tag{3}$$

Since $R_{i(j+1)} \xrightarrow[\pi[\rho R_{i(j+1)}]]{i(j)} S$, $R_{i(j+1)} \xrightarrow[\pi[\rho R_{i(j+1)}]]{i(j+1)} S_{i(j+1)}$ and $i(j+1) > i(j)$, then by Lemma 5.2

$$S \xrightarrow[\pi[\rho R_{i(j+1)}]]{+} S_{i(j+1)}. \tag{4}$$

From the induction hypothesis $T_j \xrightarrow{*} S_{i(j)}$, so all F-labeled nodes of τT_j are either expanded in $S_{i(j)}$ or are labeled with outer F's. We will show that all the nodes of τT_j have been expanded in $S_{i(j+1)}$.

Let n be any node of τT_j still labeled F in $S_{i(j)}$. There are two cases.

Case a. n is in the forest dominated by $\rho R_{i(j)}$. Then since $S_{i(j)} \xrightarrow{*} S \xrightarrow[\pi[\rho R_{i(j+1)}]]{+} S_{i(j+1)}$ by (3) and (4), since n is in the forest dominated by $\rho R_{i(j+1)}$, and since n is an outermost F-labeled node in $S_{i(j)}$, then the F at n has been expanded in $S_{i(j+1)}$.

Case b. n is not in the forest dominated by $\rho R_{i(j)}$. Since $R_{i(j)} \xrightarrow[\pi[\rho R_{i(j)}]]{i(j)} S_{i(j)}$ and the expansions in deriving $S_{i(j)}$ from $R_{i(j)}$ are confined to the forest dominated by $\rho R_{i(j)}$, then n must be an outer F-labeled node in $R_{i(j)}$. Clearly $n \notin \rho R_{i(j)}$. Thus this F at n is expanded in $R_{i(j+1)}$, and hence in $S_{i(j+1)}$.

In either case, all τT_j are expanded in $S_{i(j+1)}$ and $T_j \xrightarrow{*} S_{i(j+1)}$, so $T_{j+1} \xrightarrow{*} S_{i(j+1)}$ as required. Induction completes the claim.

From the claim and the security hypothesis, it follows that

$$\text{lub}_j\, t_j[\Omega](\bar{d}) \leqq \text{lub}_j\, s_{i(j)}[\Omega](\bar{d}) \leqq \text{lub}_i\, s_i[\Omega](\bar{d})$$

and

$$\text{lub}_j\, s_j[\Omega](\bar{d}) \leqq \text{lub}_j\, r_j[\Omega](\bar{d}).$$

Since π is a correct rule, it follows that $\text{lub}_j\, r_j[\Omega](\bar{d}) = r_0[f_p](\bar{d})$ and ϕ is correct.

(correct $\Rightarrow$ secure). Suppose the security condition is violated. Then for some R_0 and $\bar{d}$, $\text{lub}_j\, r_j[\Omega](\bar{d}) \not\leqq \text{lub}_j\, s_j[\Omega](\bar{d})$. Since π is a correct rule $\text{lub}_j\, s_j[\Omega](\bar{d}) \leqq \text{lub}_j\, t_j[\Omega](\bar{d})$ (it is obvious that $S_j \xrightarrow{*} S_{j+1}$), and so $\phi_p R_0(\bar{d}) \not\leqq r_0[f_p](\bar{d})$ and ϕ is incorrect. □

Though the security criterion is necessary and sufficient for correctness, it may be easier in a given instance to apply a simpler yet sufficient test. We give a sequence of progressively weaker criteria, each of which is sufficient to imply correctness.

We will need a way to denote functional substitution in interpreted terms. Let T be a term, I an interpretation, and let N_1 and N_2 be disjoint sets of nodes of T labeled by F. Then $t[f/N_1]$ means: under interpretation I, substitute the function f for each occurrence of a function variable at nodes N_1. Then $t[f/N_1][g/N_2]$ means: substitute the functions f and g for the F-labeled nodes of N_1 and N_2, respectively. As an abbreviation, when N_2 consists of all F-labeled nodes not in N_1, we write $t[f/N_1][g]$.

Let ϕ be a rule and let T be a term. By $\varepsilon(T)$ we mean the set of *all* F-labeled nodes of T chosen for expansion in one step of ϕ (not just the outermost F's).

COROLLARY 5.1. *Let ϕ be a rule, P be a program and I an interpretation. Of the criteria listed below,* (i) *implies* (ii) *implies* (iii). *Each criterion implies ϕ is correct for (P, I) and all $\bar{d}$.*

(i) $\forall T.\ t[\Omega/\tau(T) - \rho(T)][f_p] \leqq t[\Omega]$

(ii) $\forall T.\ \exists S.\ T \xrightarrow[\phi]{*} S\ \&\ s[\Omega/\tau(S) - \rho(T)][f_p] \leqq s[\Omega]$

(iii) *ϕ is secure for (P, I) and all $\bar{d}$.*

Proof. The second assertion in the statement of the corollary follows from the first and Theorem 5.1.

(i) $\Rightarrow$ (ii). Let $T = S$.

(ii) $\Rightarrow$ (iii). Assume ϕ satisfies (ii) and let $T = R_0 \xrightarrow[\phi]{i} R_i$ under scheme $F(\bar{X}) \Leftarrow P$.

Define $\{S_i\}$ by $R_i \xrightarrow[\pi[\rho R_i]]{i} S_i$. To establish security of ϕ we must show that for arbitrary $\bar{d}$,

$$\text{lub}_i\, s_i[\Omega](\bar{d}) \leqq \text{lub}_i\, r_i[\Omega](\bar{d}). \tag{5}$$

Let $\rho R_i = \{n_1, \cdots, n_m\}$. Let T_j be the tree rooted at n_j in R_i. Then S_i is R_i with T_j' rooted at n_j, $1 \leqq j \leqq m$, where $T_j \xrightarrow[\pi]{i} T_j'$. Then $t_j[f_p] = t_j'[f_p]$, $1 \leqq j \leqq m$, since π is correct. It then follows that $r_i[\Omega/\tau R_i - \rho R_i][f_p] = s_i[\Omega/\tau R_i - \rho R_i][f_p]$.

By monotonicity and the above identity:

$$s_i[\Omega] \leqq s_i[\Omega/\tau R_i - \rho R_i][f_p] = r_i[\Omega/\tau R_i - \rho R_i][f_p]. \tag{6}$$

From (ii), we know $\forall i\ \exists k.\ R_i \xrightarrow[\phi]{*} R_k$ and

$$r_k[\Omega/\tau R_k - \rho R_i][f_p] \leqq r_k[\Omega]. \tag{7}$$

Now in the computation $R_i \xrightarrow[\phi]{*} R_k$, nodes in the forest dominated by ρR_i are never expanded, so

$$r_i[\Omega/\tau R_i - \rho R_i][f_p] \leqq r_k[\Omega/\tau R_k - \rho R_i][f_p]. \tag{8}$$

Together (6), (8) and (7) yield $\forall i\ \exists k.\ s_i[\Omega] \leqq r_k[\Omega]$. This last inequality yields (5). □

A criterion different in flavor from those of Corollary 5.1(i) and (ii) has been given in Vuillemin [23].

DEFINITION 5.6. Rule ϕ is called *safe for* $(P, I, \bar{d})$ if for every term R

$$r[\Omega/\varepsilon R][f_p](\bar{d}) \leqq r[\Omega](\bar{d}).$$

Informally, when a term T is interpreted to give $t[\Omega]$, all occurrences of F in T are replaced by Ω. Thus the subtree under an F-labeled node in T has no effect on $t[\Omega]$. In testing if a rule ϕ is safe, we have to replace all nodes chosen for expansion in T by Ω, with all unchosen nodes being replaced by f_p. Some of the chosen nodes may be under nodes in $\rho(T)$. Since F's at nodes in $\rho(T)$ are never expanded, the subtree under a node in $\rho(T)$ has no effect on the function computed. In testing for security, no distinction is made between chosen and unchosen F's below a node in $\rho(T)$.

Security for all $(I, \bar{d})$ does not imply safety. Consider the weird rule ψ of Example 3.2(c). Rule ψ satisfies Corollary 5.1(i) since $\rho(T)$ is empty for all T (Example 4.1). However, as we shall now see, ψ is not safe. In the term $T = F(F(X))$ ψ selects the inner F for expansion, under all $(P, I, \bar{d})$. With scheme $F(X) \Leftarrow G(A, F(X))$ and I_H we have

$$t[\Omega/\varepsilon(T)][f_p](\bar{d}) = f_p(\perp) = G(A, G(A, \cdots)) \neq t[\Omega](\bar{d}) = \perp.$$

We will show in Theorem 5.2 that all safe rules are correct. From Theorem 5.1 and the fact that rule ϕ is secure but not safe, safety is strictly subsumed by security. As with the tests in Corollary 5.1, safety may be easier to test than security.

In proving Theorem 5.2 we will again be interested in outermost F-labeled nodes that are never expanded. However, we will also need to talk about nodes in

the subterm below an F that is never expanded. The following definitions generalize the concept of $\tau(\cdot)$ and $\rho(\cdot)$.

DEFINITION. For term T, let $\tau_i(R)$ be the set of F-labeled nodes x of R that have exactly i F-labeled nodes (including x) on the path from x to the root of R.

Note that $\tau_1(R)$ is $\tau(R)$ in Theorem 5.1.

DEFINITION. Let $\rho_0(R)$ be the set consisting of the root of R. Let $\rho_i(R)$ be the subset of $\tau_i(R)$ dominated by the set of nodes $\rho_{i-1}(R)$ such that no element of $\rho_i(R)$ will ever be expanded under ϕ with starting term R.

Note that for $i>0$, nodes of $\rho_i(R)$ are never expanded. Moreover, none of their ancestors are ever expanded. Note that $\rho_1(R)$ was called $\rho(R)$ in Theorem 5.1.

LEMMA 5.3. *Let* $R \underset{\phi}{\rightarrow} R'$. *Then* $\forall i.\ \rho_i(R) \subseteq \rho_i(R')$.

Proof. The proof is immediate from the definition of $\rho_i(\cdot)$. □

Another notion we need to define for Theorem 5.2 is the notion of "copies". Consider the scheme $F(X) \Leftarrow P = G_1(FG_2FX, G_2X)$ from Fig. 2.1. The formal parameter X appears in two places in P. Thus when macroexpansion under the scheme $F(X) \Leftarrow P$ takes place, the actual parameters for X will be substituted in two places; resulting in "copies". For brevity we will define the notion of "copies" only for certain nodes in certain terms. In the process we will keep track of exactly where parameters are being substituted by tracing the path from the root of a term to a given node x in a term.

DEFINITION. Let x be a node in term R. Let $y_0, y_1, \cdots, y_{k+1} = x$ be the path from the root y_0 of R to x. The *trace of* x *in* R, written $\mathcal{T}Rx$ is the sequence $(H_0)_{j_0}(H_1)_{j_1} \cdots (H_k)_{j_k}$, where for all i, $0 \leqq i \leqq k$, H_i is the function label (a base function or function variable) of node y_i, and y_{i+1} is the j_ith ordered direct descendent of y_i.

For example in $G_1(FG_2FX, G_2\underline{X})$, the trace of the underlined X is $(G_1)_2(G_2)_1$. Given a scheme $F(\bar{X}) \Leftarrow P$, we will use the notion of trace in P to identify all the leaves at which a formal parameter X_i appears in P.

DEFINITION. Let $F(\bar{X}) \Leftarrow P$ be a scheme. The *trace set* of X_i in P, denoted $\mathcal{T}P(X_i)$ is given by:

$$\mathcal{T}P(X_i) = \{\mathcal{T}Px \mid x \text{ in } P \text{ has label } X_i\}.$$

THEOREM 5.2. *Given scheme* P, *interpretation* I *and input* $\bar{d}$, *rule* ϕ *is correct for* $(P, I, \bar{d})$ *if* ϕ *is safe for* $(P, I, \bar{d})$.

Proof. Let $R_0 = T_0$ be the starting term and let sequences $\{R_i\}$ and $\{T_j\}$ be defined by $R_0 \overset{i}{\underset{\phi}{\rightarrow}} R_i$ and $T_0 \overset{j}{\underset{\pi}{\rightarrow}} T_j$. Moreover, let $\{V_j\}$ be given by $F(\bar{X}) \overset{j}{\underset{\pi}{\rightarrow}} V_j$.

We will be interested in a particular subsequence of the $\{R_i\}$ sequence given by the following: (i) $R_{i(0)} = R_0$, and (ii) $R_{i(j+1)}$ is the first term in $\{R_i\}$ such that all elements of $\bigcup_{k=1}^{j+1} (\tau_k R_{i(j)} - \rho_k R_{i(j)})$ have been expanded in $R_{i(j+1)}$.

From the $\{R_{i(j)}\}$ sequence we will construct a sequence $\{S_j\}$ as follows:

$$S_j = R_{i(j)}\Big(\bigcup_{k=1}^{j} \rho_k R_{i(j)} \leftarrow V_j \Big).$$

As in Theorem 5.1, we will show that the $\{T_j\}$ sequence is dominated by the $\{S_j\}$ sequence. From safety we will show that the $\{S_j\}$ sequence has no more

"semantic information" than the $\{R_{i(j)}\}$ subsequence. Then $\{T_j\}$ has no more "information" than $\{R_{i(j)}\}$. Since π is a correct rule, and $\{R_{i(j)}\}$ is a subsequence of $\{R_i\}$, it will follow that ϕ is correct for $(P, I, \bar{d})$.

Consider terms $R_{i(j)}$ and S_j. Let the scheme in question be $F(\bar{X}) \Leftarrow V_j$. Consider a node x in $R_{i(j)}$ such that all F-labeled proper ancestors of x are in the set $\bigcup_{i=1}^{j} \rho_k R_{i(j)}$. Node y in S_j is a *copy* of x in $R_{i(j)}$ if $\mathcal{T}S_j y$ can be formed from $\mathcal{T}R_{i(j)}x$ by replacing all occurrences of $(F)_h$, for some h, in $\mathcal{T}R_{i(j)}x$ by a string from $\mathcal{T}V_j(X_h)$. The notion of copy for all other nodes x in $R_{i(j)}$ is undefined.

We are now in a position to set up an inductive proof of the theorem. By induction we prove the following:

CLAIM. *For all* j

(i) $S_j \overset{*}{\rightarrow} S_{j+1}$, *and*

(ii) $T_j \overset{*}{\rightarrow} S_j$.

Basis $j = 0$. By definition $R_{i(0)} = R_0 = T_0$ and $V_0 = F(\bar{X})$. Clearly $S_0 = R_{i(0)}$, so $T_0 \overset{*}{\rightarrow} S_0$. $S_0 \overset{*}{\rightarrow} S_1$ is clear since $S_1 = S_0(\rho_1 R_0 \leftarrow V_1)$.

Inductive step $j > 0$. Let $R_{i(j+1)}$ be the first term in the expansion of $R_{i(j)}$ by ϕ in which all elements of $\bigcup_{k=1}^{j+1} (\tau_k R_{i(j)} - \rho_k R_{i(j)})$ have been expanded. We will verify parts (i) and (ii) of the inductive hypothesis.

(i) By definition $R_{i(j)} \overset{*}{\rightarrow} R_{i(j+1)}$. Moreover, since $\forall k \geqq 0$, $\rho_k R_{i(j)} \subseteq \rho_k R_{i(j+1)}$, it follows that $S_j \overset{*}{\underset{\phi}{\rightarrow}} S_{j+1}$.

(ii) Consider node $z \in \tau_1 T_j$, and let y in S_j be such that $\mathcal{T}S_j y = \mathcal{T}T_j z$. We will consider three cases.

Case 1. y in S_j is a copy of a node x in $\bigcup_{k=1}^{j+1} (\tau_k R_{i(j)} - \rho_k R_{i(j)})$. By the definition of $R_{i(j+1)}$, x has been macroexpanded in $R_{i(j+1)}$. Let $S = R_{i(j+1)}(\bigcup_{k=1}^{j} \rho_k R_{i(j)} \rightarrow V_j)$. Since nodes in $\rho_k R_{i(j)}$ are never expanded under ϕ, it is easy to see that $S_j \overset{*}{\rightarrow} S$. Since x has been macroexpanded in $R_{i(j+1)}$, node y in S_j has been macroexpanded in S. Since $\rho_k R_{i(j)} \subseteq \rho_k R_{i(j+1)}$ it follows that $S \overset{*}{\rightarrow} S_{j+1}$, so y has been macroexpanded in S_{j+1}.

Case 2. y in S_j is a copy of a node x in $\bigcup_{k=1}^{j+1} \rho_k R_{i(j)}$. If $x \in \bigcup_{k=1}^{j} \rho_k R_{i(j)}$, then from the definition of S_j, y has already been macroexpanded in S_j, and hence S_{j+1}. Therefore consider the case when $x \in \rho_{j+1} R_{i(j)}$. In this case let $S = R_{i(j)}(\bigcup_{k=1}^{j+1} \rho_k R_{i(j)} \leftarrow V_j)$. Clearly y has been macroexpanded in S. It is easy to see that $S_j \overset{*}{\rightarrow} S \overset{*}{\rightarrow} S_{j+1}$, so y is macroexpanded in S_{j+1}.

Case 3. The remaining case occurs when y in S_j is not a copy of any node in $R_{i(j)}$. Outside the forest dominated by nodes in $\rho_1 R_{i(j)}$, terms $R_{i(j)}$ and S_j are identical, by construction. If y occurs outside the forest dominated by $\rho_1 R_{i(j)}$, then y must be in $\tau_1 R_{i(j)}$. But then y is macroexpanded in $R_{i(j+1)}$ and hence S_{j+1}.

For each node u in $\rho_1 R_{i(j)}$ there is a unique copy in S_j. Thus y in S_j must have a lowest proper ancestor w in S_j such that w is a copy of a node x in $R_{i(j)}$.

Consider $\mathcal{T}S_j w$ and $\mathcal{T}S_j y$. Since w is a proper ancestor of y, $\mathcal{T}S_j w$ must be a proper prefix of $\mathcal{T}S_j y$. Let $\mathcal{T}S_j y = \mathcal{T}S_j w.\xi$, where ξ is a string of base functions with subscripts.

We will first show that node x in F must be in $\bigcup_{k=1}^{j} \rho_k R_{i(j)}$. Since y is an outermost F in S_j, $\mathcal{T}S_j y$ and hence $\mathcal{T}S_j w$ have no function symbols. Thus all proper ancestors of x with F labels must be in $\bigcup_{k=1}^{j} \rho_k R_{i(j)}$. If x has label F, and is not in

$\cup_{k=1}^{j} \rho_k R_{i(j)}$, then node w will be labeled F, which is not possible. If x is labeled with a base function, then all direct descendants of x in $R_{i(j)}$ have copies in S_j, contradicting the choice of w in S_j. Thus x is in $\cup_{k=1}^{j} \rho_k R_{i(j)}$.

Since x is macroexpanded under V_j in constructing S_j from $R_{i(j)}$, ξ is either a proper prefix of a string in $\mathcal{T}V_j(X_h)$, for some h, or for some h, a string η in $\mathcal{T}V_j(X_h)$ is a prefix of ξ. But if η is a prefix of ξ, then let $\mathcal{T}S_j w \cdot \eta$ be the trace of node u in S_j. From the definition of copy, u must be a copy of some node v in $R_{i(j)}$, contradicting the choice of w. Thus ξ must be a proper prefix of a string in $\mathcal{T}V_j(X_h)$, for some h.

Consider V_j. Since y is in $\tau_1 S_j$, it follows that the node with trace ξ in V_j is an element of $\tau_1 V_j$. But by definition of π, since $V_j \underset{\pi}{\rightarrow} V_{j+1}$, the node with trace ξ in V_{j+1} has been macroexpanded.

S_{j+1} is formed from $R_{i(j+1)}$ by macroexpansion under V_{j+1}. Since $y \in \tau_1 S_j$ and $S_j \overset{*}{\rightarrow} S_{j+1}$, the F at node y has been macroexpanded in S_{j+1}.

The inductive hypothesis has therefore been verified.

The claim demonstrated the existence of a chain $\{S_j\}$ such that $\forall j.\ T_j \overset{*}{\rightarrow} S_j$. Thus

$$\text{lub}_j\, t_j[\Omega] \leqq \text{lub}_j s_j[\Omega]. \tag{9}$$

Consider $S_j = R_{i(j)}(\cup_{k=1}^{j} \rho_k R_{i(j)} \rightarrow V_j)$. Since elements of $\rho_k R_{i(j)}$ are never chosen for expansion under ϕ

$$\forall k.\ \rho_k R_{i(j)} \subseteq \bar{\varepsilon} R_{i(j)}$$

where $\bar{\varepsilon} R_{i(j)}$ are all the F-labeled nodes not chosen by ϕ in $R_{i(j)}$. Thus $S_j \overset{*}{\rightarrow} S_j' = R_{i(j)}(\bar{\varepsilon} R_{i(j)} \leftarrow V_j)$. Moreover

$$\begin{aligned} s_j[\Omega] \leqq s_j'[\Omega] &= r_{i(j)}[v_j[\Omega]/\bar{\varepsilon} R_{i(j)}][\Omega] \\ &\leqq r_{i(j)}[f_p/\bar{\varepsilon} R_{i(j)}][\Omega] \end{aligned}$$

by monotonicity and the fact that $v_j[\Omega] \leqq f_p$. From the safety hypothesis, $r_{i(j)}[f_p/\bar{\varepsilon} R_{i(j)}][\Omega](\bar{d}) \leqq r_{i(j)}[\Omega](\bar{d})$. Thus $s_j[\Omega](\bar{d}) \leqq r_{i(j)}[\Omega](\bar{d})$ and

$$\text{lub}_j\, s_j[\Omega](\bar{d}) \leqq \text{lub}_j\, r_{i(j)}[\Omega](\bar{d}) \leqq \text{lub}_j\, r_j[\Omega](\bar{d}). \tag{10}$$

Together, inequalities (9) and (10) yield $\text{lub}_j\, t_j[\Omega](\bar{d}) \leqq \text{lub}_j\, r_j[\Omega](\bar{d})$, and since π is universally correct, ϕ must be correct for $(P, I, \bar{d})$. □

In [23] the safety condition is applied to show that a number of particular computation rules are correct.

6. Conclusions. In this paper we have been concerned with the relationship between recursive schemes, computation rules and interpretations. In § 4 we considered C_U, the class of *all* interpretations, and sought computation rules that were correct for all interpretations in C_U. In § 5 we considered the class C_I which consists of the single interpretation I. Again we sought computation rules that were correct for all input data under all interpretations in C_I. The same question can be asked about other interesting classes of interpretations C.

A related question to that of correctness of a computation rule is equivalence between recursive schemes with respect to a class of interpretations. For the class C_U this question has been studied in [18]. Again, equivalence for interesting classes of interpretations (if decidable) might be a worthwhile question.

Snyder [21] considers the classes of functions that can be computed using certain computation rules. While the rules studied were not all fixpoint rules, he showed that all functions computed by "normal" evaluation [11] can be computed by "call by value". His model permits side-effects and iteration, so the idea is to call repeatedly a recursive procedure. Initially the procedure is called with undefined arguments and returns with an indication of the next argument it would like. The model is partially interpreted in that if-then-else constructions are permitted. The interesting point is that, with side-effects, call by name is strictly more powerful than call by value: there are functions that can be computed by call by name but not by call by value.

Within the context of the model in this paper consider the following question: given a recursive scheme S is there a recursive scheme S' such that the function computed by computation rule ϕ for S' is the same as that computed by a correct computation rule for S? The possibility of adding interpreted base functions exists.

By way of an example consider the recursion scheme: $F(X, Y) \Leftarrow G_1(X, F(G_2X, F(X, Y)))$, which, suitably interpreted, was used in Example 1.2 to show that parallel outermost converges while parallel innermost diverges. Let us add the binary base function $H(X, Y)$ where $h(x, y)$ is lub $\{x, y\}$. For the scheme in Fig. 6.1 parallel innermost computes the same function as parallel outermost does with the original scheme.

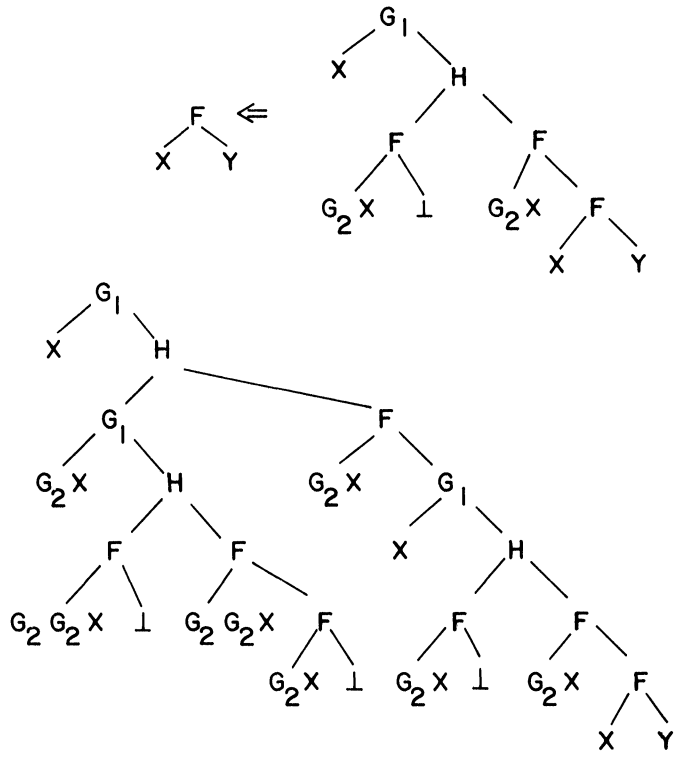


FIG. 6.1. *A modified recursive scheme, and the term after two steps of parallel innermost evaluation from* $F(X, Y)$

Acknowledgments. The impetus for this paper came from an informal 4 o'clock seminar with Ed Robertson, Joel Seiferas and Ray Zahar. Their incisive questions led to some of the answers in this paper. Barry Rosen simplified Corollaries 4.1 and 4.2.

REFERENCES

[1] J. M. CADIOU, *Recursive definitions of partial functions and their computations*, Ph.D. thesis, Computer Sci. Dept., Stanford Univ., Stanford, Calif., 1972.

[2] R. L. CONSTABLE AND D. GRIES, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[3] B. COURCELLE, *Recursive schemes, algebraic trees and deterministic languages*, Proc. 15th Ann. Symp. on Switching and Automata Theory, New Orleans, 1974.

[4] B. COURCELLE AND J. VUILLEMIN, *Semantics and axiomatics of a simple recursive language*, Proc. 6th Ann. ACM Symp. on Computing, Seattle, 1974, pp. 13–26.

[5] B. COURCELLE, J. VUILLEMIN AND M. NIVAT, in preparation.

[6] P. J. DOWNEY, *Formal languages and recursion schemes*, Ph.D. thesis, Center for Research in Computing Technology, Harvard Univ., Cambridge, Mass., 1974.

[7] H. EGLI AND R. CONSTABLE, *Computability concepts for programming language semantics*, Proc. 7th Ann. ACM Symp. on Computing, Albuquerque, May 1975, pp. 98–106.

[8] D. GRIES, *Recursion as a programming tool*, unpublished.

[9] P. Z. INGERMAN, *Pānini–Backus form suggested*, Comm. ACM, 7 (1967), p. 137.

[10] Z. MANNA, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[11] Z. MANNA, S. NESS AND J. VUILLEMIN, *Inductive methods for proving properties of programs*, Comm. ACM, 16 (1973), pp. 491–502.

[12] Z. MANNA AND A. SHAMIR, *The optimal fixedpoint of recursive programs*, Proc. 7th Ann. ACM Symp. on Computing, Albuquerque, May 1975, pp. 194–206.

[13] G. MARKOWSKY AND B. K. ROSEN, *Bases for chain-complete posets*, 16th Ann. Symp. on Foundations of Computer Science, Berkeley, Calif., Oct. 1975, pp. 34–47.

[14] J. H. MORRIS, *Lambda calculus models of programming languages*, MAC-TR-57, MIT Project MAC, Cambridge, Mass., 1968.

[15] P. NAUR, ed., *Revised report on the algorithmic language, ALGOL 60*, Comm. ACM, 6 (1963), pp. 1–17.

[16] M. NIVAT, *On the interpretation of recursive program schemes*, Proc. Convegno d'Informatica Teorica, Istituto di Alta Matematica di Roma, Feb. 1973.

[17] J. REYNOLDS, *Notes on a lattice-theoretic approach to the theory of computation*, Dept. of Systems and Inf. Science Rep., Syracuse Univ., Syracuse, N.Y., Oct. 1972.

[18] B. K. ROSEN, *Program equivalence and context-free grammars*, J. Comput. System Sci., 11 (1975), pp. 358–374.

[19] ———, *Tree-manipulation systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20 (1973), pp. 160–187.

[20] D. SCOTT, *Outline of a mathematical theory of computation*, Oxford Mon. PRG-2, Oxford University Press, Oxford, England, 1970.

[21] L. SNYDER, *An analysis of parameter evaluation for recursive procedures*, Thesis, Dept. of Computer Sci., Carnegie–Mellon Univ., Pittsburgh, 1973.

[22] A. TARSKI, *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math., 5 (1955), pp. 285–309.

[23] J. VUILLEMIN, *Correct and optimal implementations of recursion in a simple programming language*, J. Comput. System Sci., 9 (1974), pp. 332–354.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

GENERALIZED PROGRAM SCHEMAS*

ASHOK K. CHANDRA†

Abstract. A unified approach towards program schemas is described in order to explicate the notions of uninterpreted computer programs and the data structures used by such programs.

Key words. data structure, first order formalism, generalized schema, Herbrand schema, nonoracle schema, program schema, uninterpreted program

1. Introduction. The aim of this study is three-fold: (i) to study the class of computable functionals on uninterpreted domains, and the machines (or program schemas) on which such functionals can be computed, (ii) to unify the notion of the interpretation for a program schema with the notion of the data structures which the schema uses, and (iii) to unify a large number of the classes of program schemas considered by earlier researchers so as to clarify the various notions associated with computations on uninterpreted domains and to examine various conjectures regarding the power of "all" schemas.

Regarding the first point, it is reasonable that the study of computable functionals (on uninterpreted domains) will go along lines similar to the study of partial recursive functions (on interpreted domains); i.e., various classes of computing machines will be proposed and several such classes will be found to be equivalent, and maximal with respect to these classes, leading to the belief that these classes compute "all" the computable functionals. This is analogous to the role played by Turing machines for the computable functions. We can, however, lay down certain criteria of computability for functionals. We propose the following (a different set of criteria would lead to a different notion of computability): as for machines on interpreted domains, the requirements that the machines computing functionals be finite and that they operate in a stepwise and nonrandom fashion (without recourse to analog devices or randomizing mechanisms such as throwing dice) are reasonable—Rogers [13]. In addition, we propose the following (which are unique to machines computing functionals):

1. *Finite first order*—only first order functions and predicates (called base functions and predicates) are allowed, and each machine may use only a finite number of these.
2. *Totality*—all base functions and predicates are total.
3. *Characterizability*—the computation should be completely characterized by an interpretation for the base functions and predicates (and the inputs, if any).
4. *Well-foundedness*—the computation of a machine on isomorphic interpretations must be the same.
5. *Nonoracularity*—in one step of the computation the machine should be able to "look-at" at most a finite number of elements in the domain of the interpretation.

Our formalism uses a slightly stronger notion of well-foundedness than 4 above.

* Received by the editors May 6, 1975.

† Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. This research was supported in part by the Advance Research Projects Agency of the Office of the Secretary of Defense under Contract SD-183.

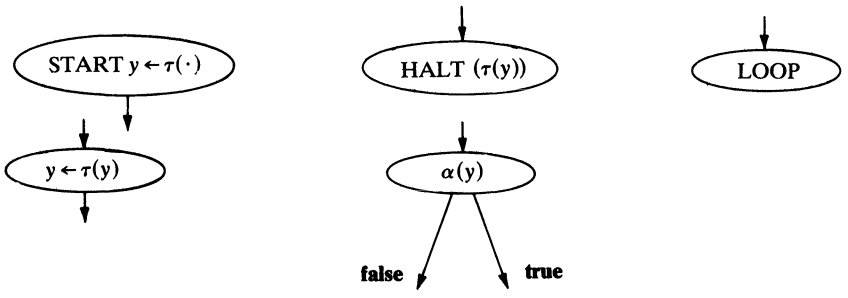
Regarding the second aim of this study, we note that when Ianov [6] first considered program schemas he treated them as models for computer programs where the entire data space of the program could be represented by a single variable, which could be changed by applying functions, or tested by predicates (which were considered uninterpreted). This was not an adequate model for several purposes because (i) the data space of real programs is often subdivided into individual parts that can be changed independently, and (ii) for proving properties about programs the basic operations frequently cannot be considered uninterpreted. Extensions of both have been considered. Using the one-variable philosophy of Ianov, however, we can represent the subdivision of the data space by partially interpreted functions and predicates, which suffice to handle both (i) and (ii) above. This is the approach we will use.

With reference to the third aim of this study, we observe that several researchers have proposed classes of schemas that are claimed to be maximal for "all" schemas. These include the effective functionals of Strong [16] and the flowchart schemas with arrays of Constable and Gries [4]. While these classes are quite stable in that the addition of a large number of features such as counters, pushdown stacks, queues, etc., does not increase the power of the schemas, we question the philosophical significance of the statement that these classes are maximal for the uninterpreted schemas. In our model the uninterpreted schemas form a strictly more powerful class, but the flowchart schemas with arrays are maximal with respect to the uninterpreted Herbrand schemas, which was conjectured by Chandra and Manna [3].

2. Definition of generalized schemas.

DEFINITION. The *vocabulary* $V \cup V'$ where $V = \{f_i^j : i, j \geqq 0\} \cup \{p_i^j : i, j \geqq 0\}$ and $V' = \{g_i^j : i, j \geqq 0\} \cup \{q_i^j : i, j \geqq 0\}$ consists of a denumerable set of ranked function symbols f_i^j, g_i^j, and predicate symbols p_i^j, q_i^j, of rank j. For convenience in describing schemas we will, however, frequently use function and predicate symbols other than those in $V \cup V'$.

DEFINITION. A *schema* S is a triple $\langle F, \phi, P \rangle$ of a finite flowchart F, a formula ϕ of first order predicate calculus with equality, and a finite set $P \subset V$. The flowchart F has one variable y, and statements consist of the following:



where $\tau(\cdot)$ represents a term not containing y, $\tau(y)$ represents an arbitrary term,

and $\alpha(y)$ represents an arbitrary atomic formula. All statements in F are considered to be labeled with distinct labels; terms and atomic formulas use symbols from the vocabulary $V \cup V'$. F has a unique start statement, but all function and predicate symbols used in F need not be in P. $\Sigma(S)$ denotes the set of function and predicate symbols that are used in F, ϕ, or in P. Note that the schemas have no inputs, but one can use zero-ary function symbols instead of inputs. For convenience we will use ALGOL-like notation instead of strict flowchart notation.

DEFINITION. An interpretation I is said to be an *interpretation for a schema* S if it specifies at least the base functions and predicates in $\Sigma(S)$, and ϕ is true in I ($I \models \phi$). Dom(I) denotes the domain of I, and $\Sigma(I)$ the set of function and predicate symbols in I.

DEFINITION. Given a schema S and an interpretation I for S, the path Path(S, I) of the computation of S on I is the finite or infinite sequence $s_1, s_2 \cdots$ of statements executed during the computation of S on I, except that if in the ith step, say, a test is made, then s_i is $\langle t, b \rangle$ where t is the test statement, and b is "true" or "false" corresponding to the exit taken in the computation. Val(S, I) is the output of the computation of S on I if it halts, and is undefined otherwise.

DEFINITION. Given an interpretation I and a set $P \subset \Sigma(I)$, we define the *subinterpretation* I/P *of* I *with respect to* P as follows: Dom(I/P) is the smallest subset of Dom(I) closed under the functions of P; and the values of the functions and predicates of P are the same in I/P as in I. Note—if P contains no zero-ary function, then Dom(I/P) is empty.

DEFINITION. A schema $S = \langle F, \phi, P \rangle$ is said to be *well-founded* if for every two interpretations I_1, I_2 for S (i.e., $I_1 \models \phi$ and $I_2 \models \phi$) such that there is an isomorphism θ from I_1/P to I_2/P, we have

(i) Path(S, I_1) = Path(S, I_2), and

(ii) if the computations halt, then Val(S, I_2) = θ(Val(S, I_1)).

Note that if $S = \langle F, \phi, P \rangle$ is well-founded and its computation on I halts, then Val(S, I) $\in$ Dom(I/P). The significance of a set P that makes S well-founded is that for any interpretation for S, knowledge of merely the functions and predicates P is sufficient to characterize the computation. We will only be interested in schemas that are well-founded, and in this paper, all schemas considered are well-founded.

It follows from the definition that

(a) given any F and ϕ, if Q is the set of function and predicate symbols in F, then $\langle F, \phi, Q \rangle$ is well-founded,

(b) if $\langle F, \phi, P \rangle$ is well-founded, and Q is any set such that $P \subset Q$, then $\langle F, \phi, Q \rangle$ is also well-founded, and

(c) if ϕ is "false," then $\langle F, \phi, P \rangle$ is well-founded for all F and P.

It is not partially decidable whether a schema S is well-founded. This follows directly from the fact that the divergence problem for schemas is not partially decidable (§ 6). The undecidability of well-foundedness should not shock us unduly. The corresponding problem for the usual program schemas referred to in the literature [1]–[8], [10]–[12], [14] (*conventional schemas*), too, is not partially

decidable. For, consider a conventional schema S with a statement HALT(f_0^0) where f_0^0 is not used in the rest of S. Now we ask if the output of S can be determined if we give an interpretation for S, but refuse to specify the value of f_0^0. This is not partially decidable.

The correspondence between conventional schemas and generalized schemas can be represented as follows:

Conventional schema	*Generalized schema*
The total data space	The variable y
Functions and predicates	The set P
Interpretation	I/P
The structure of the data space, and totally interpreted features (like counters)	Predicates and functions other than those in P, related by the formula ϕ.

We obtain subclasses of the generalized schemas by restricting the kinds of flowcharts and the formulas ϕ allowed. Section 4 describes how features like counters and pushdown stacks can be represented.

Example. Consider the schema $S_1 = \langle F_1, \phi_1, P_1 \rangle$. There are two zero-ary functions a_0, a_1, and two binary functions f_+, $f.$. The formula ϕ_1 is:

$$
\begin{aligned}
& a_0 \neq a_1 \\
\wedge\, & \forall x \forall y && f_+(x, y) = f_+(y, x) \wedge f.(x, y) = f.(y, x) \\
\wedge\, & \forall x \forall y \forall z && f_+(f_+(x, y), z) = f_+(x, f_+(y, z)) \wedge f.(f.(x, y), z) = f.(x, f.(y, z)) \\
\wedge\, & \forall x && f_+(x, a_0) = x \wedge f.(x, a_1) = x \\
\wedge\, & \forall x \exists y && f_+(x, y) = a_0 \\
\wedge\, & \forall x && (x \neq a_0) \rightarrow \exists y f.(x, y) = a_1 \\
\wedge\, & \forall x \forall y \forall z && f.(x, f_+(y, z)) = f_+(f.(x, y), f.(x, z)),
\end{aligned}
$$

the flowchart F_1 is:

START $y \leftarrow a_1$;

while $y \neq a_0$ **do** $y \leftarrow f_+(y, a_1)$;

HALT (a_0),

and the set P_1 is $\{a_1, f_+\}$.

An interpretation for the schema S_1 is a field. The schema halts if and only if the characteristic of the field is finite. Note that a_0 is not in P_1, but the schema is well-founded.

3. Properties of schemas.

DEFINITION. A schema S *halts* (*diverges*) if its computation halts (diverges) on every interpretation for S.

DEFINITION. A schema S is *free* if for every path through S beginning at the start statement, there is an interpretation for S for which the computation follows that path.

DEFINITION. We say that two schemas $S_1 = \langle F_1, \phi_1, P_1\rangle$ and $S_2 = \langle F_2, \phi_2, P_2\rangle$ are *compatible* if $P_1 = P_2$.

DEFINITION. $S_2 = \langle F_2, \phi_2, P\rangle$ is a *generalization* of $S_1 = \langle F_1, \phi_1, P\rangle$ (notation $S_2 \underset{\text{gen}}{\geqq} S_1$) if: $\forall I_1$ for S_1 $\exists I_2$ for S_2 and $\exists$ an isomorphism θ: $(I_1/P) \leftrightarrow (I_2/P)$ such that if S_1 halts on I_1, then S_2 also halts on I_2 and $\text{Val}(S_2, I_2) = \theta(\text{Val}(S_1, I_1))$; and if $\text{Val}(S_1, I_1)$ is undefined, then $\text{Val}(S_2, I_2)$ is also undefined.

DEFINITION. $S_2 = \langle F_2, \phi_2, P\rangle$ *includes* (is at least as defined as) $S_1 = \langle F_1, \phi_1, P\rangle$ (notation $S_2 \geqq S_1$) if:

(i) $\forall I_1$ for S_1, $\exists I_2$ for S_2 and $\exists$ an isomorphism $\theta : (I_1/P) \leftrightarrow (I_2/P)$ such that if S_1 halts on I_1, then S_2 also halts on I_2, and $\text{Val}(S_2, I_2) = \theta(\text{Val}(S_1, I_1))$, and

(ii) $\forall I_2$ for S_2, $\exists I_1$ for S_1 and $\exists$ an isomorphism $\theta : (I_1/P) \leftrightarrow (I_2/P)$ such that if S_1 halts on I_1, then S_2 also halts on I_2, and $\text{Val}(S_2, I_2) = \theta(\text{Val}(S_1, I_1))$.

It can be checked that $\underset{\text{gen}}{\geqq}$ and $\geqq$ are reflexive and transitive. We also write $S_1 \underset{\text{gen}}{\leqq} S_2$ for $S_2 \underset{\text{gen}}{\geqq} S_1$, $S_1 \leqq S_2$ for $S_2 \geqq S_1$, etc.

DEFINITION. We say that two compatible schemas S_1 and S_2 are *equivalent* ($S_1 \equiv S_2$) if $S_1 \underset{\text{gen}}{\geqq} S_2$, and $S_2 \underset{\text{gen}}{\geqq} S_1$.

Comment. $S_1 \equiv S_2$ if and only if $S_1 \geqq S_2$, and $S_2 \geqq S_1$.

Consider the schema $S_2 = \langle F_2, \phi_2, P_2\rangle$ where ϕ_2 is

$$\forall x \; f_+(x, a_1) = a_1 \leftrightarrow x = a_0,$$

F_2 is

START $y \leftarrow a_1$;

while $y \neq a_0$ **do** $y \leftarrow f_+(y, a_1)$;

HALT(a_0),

and P_2 is $\{a_1, f_+\}$. If S_1 is the schema of the example in § 2, then $S_2 \underset{\text{gen}}{\geqq} S_1$, but not $S_1 \underset{\text{gen}}{\geqq} S_2$, since the characteristic of a field must be a prime or infinity.

When we say that a conventional schema is uninterpreted, we mean that any interpretation over its base functions is an interpretation for the schema. We say it is uninterpreted even though its structural features are interpreted, e.g., the operation of pushing a value into a stack, or of incrementing a counter, is well defined. We would like to make this notion concrete, and apply it to our generalized schemas.

DEFINITION. A schema $S = \langle F, \phi, P\rangle$ is said to be *uninterpreted* if it is well-founded and for every interpretation I where $P \subset \Sigma(I)$, there is an interpretation I' for S whose subinterpretation over P is isomorphic to I/P, i.e., $\forall I$ such that $P \subset \Sigma(I)$, $\exists I'$ for S such that $\exists$ an isomorphism $\theta : (I/P) \leftrightarrow (I'/P)$.

Consider the schema $S_3 = \langle F_3, \phi_3, P_3 \rangle$ where F_3 is

START $y \leftarrow f(a)$;

while $p(y)$ **do** $y \leftarrow f(y)$;

HALT($g(a)$),

ϕ_3 is

$$g(a) = f(a)$$

and P_3 is $\{a, f, p\}$. Then S_3 is uninterpreted, but $S_3' = \langle F_3, \phi_3, \{a, f, g, p\} \rangle$ is not. Note that both S_3, S_3' are well-founded, but $\langle F_3, \phi_3, \{a, g, p\} \rangle$ is not.

If H is a Herbrand interpretation corresponding to an interpretation I, we write $I \overset{h}{\rightarrow} H$; i.e., H is a free interpretation with $\Sigma(H) = \Sigma(I)$ and such that $p_i^j(\tau_1(\cdot), \cdots, \tau_j(\cdot))$ is true in H if and only if it is true in I.

DEFINITION. A schema $S = \langle F, \phi, P \rangle$ is called a *Herbrand schema* if it is well-founded, and

(a) $\forall I$ for S, $\exists H$ for S, such that $(I/P) \overset{h}{\rightarrow} (H/P)$,

(b) $\forall H$ for S, $\forall I_1$ such that $(I_1/P) \overset{h}{\rightarrow} (H/P)$, $\exists I$ for S, such that $\exists$ an isomorphism $\theta : (I_1/P) \leftrightarrow (I/P)$, and

(c) $\forall I, H$ for S, if $(I/P) \overset{h}{\rightarrow} (H/P)$ then Path(S, I) = Path(S, H), and Val(S, I) corresponds to Val(S, H) in the homomorphism $(I/P) \overset{h}{\rightarrow} (H/P)$.

The notion of a Herbrand schema (Chandra and Manna [3]) is that of a schema which cannot distinguish between Herbrand and non-Herbrand interpretations.

The property of being uninterpreted is independent of being a Herbrand schema. Consider, for example:

F_4 is

START $y \leftarrow a$; **if** $p(a, a_1)$ **then** HALT(y) **else** LOOP,

ϕ_4 is

$$\forall x \forall y\, p(x, y) \leftrightarrow (x = y),$$

F_5 is

START $y \leftarrow a$; **if** $q(y)$ **then** HALT(y) **else** LOOP,

ϕ_5 is

$$\forall x\, q(x) \leftrightarrow \neg q(f(x)).$$

$S_4 = \langle F_4, \phi_4, \{a, a_1\} \rangle$ and $S_4' = \langle F_4, \phi_4, \{a, a_1, p\} \rangle$ are both non-Herbrand, but S_4 is uninterpreted whereas S_4' is not. $S_5 = \langle F_5, \phi_5, \{a, q\} \rangle$ and $S_5' = \langle F_5, \phi_5, \{a, f, q\} \rangle$ are both Herbrand schemas, but S_5 is uninterpreted and S_5' is not.

Given a class $\mathcal{I}$ of interpretations, a schema S is said to *halt on* $\mathcal{I}$ if S halts on every interpretation I for S, where $I \in \mathcal{I}$; and similarly for divergence and freedom. And we say that $S_2 \underset{\text{gen}}{\geqq} S_1$ *on* $\mathcal{I}$ if $\forall I_1$ for S_1 such that $I_1 \in \mathcal{I}$, $\exists I_2$ for S_2, $I_2 \in \mathcal{I}$, and $\exists$ an isomorphism $\theta : (I_1/P) \leftrightarrow (I_2/P)$ such that either both schemas diverge, or $\text{Val}(S_2, I_2) = \theta(\text{Val}(S_1, I_1))$—compare with the definition of $S_2 \underset{\text{gen}}{\geqq} S_1$. And similarly for inclusion and equivalence.

Given a set P of function and predicate symbols, let $\mathcal{H}_P$ be the class of interpretations H such that (H/P) is a Herbrand (free) interpretation. The following is a slightly more general version of a similar theorem for conventional schemas [2].

THEOREM 1 (Theorem of Herbrand schemas). *For Herbrand schemas* $S_1 = \langle F_1, \phi_1, P\rangle$ *and* $S_2 = \langle F_2, \phi_2, P\rangle$,

(a) S_1 *halts if and only if it halts on* $\mathcal{H}_P$,

(b) S_1 *diverges if and only if it diverges on* $\mathcal{H}_P$,

(c) $S_1 \equiv S_2$ *if and only if* $S_1 \equiv S_2$ *on* $\mathcal{H}_P$,

(d) $S_1 \leqq S_2$ *if and only if* $S_1 \leqq S_2$ *on* $\mathcal{H}_P$,

(e) $S_1 \underset{\text{gen}}{\leqq} S_2$ *if and only if* $S_1 \underset{\text{gen}}{\leqq} S_2$ *on* $\mathcal{H}_P$, *and*

(f) S_1 *is free if and only if* S_1 *is free on* $\mathcal{H}_P$.

Proof. We prove only part (e). The other parts follow likewise.

Only if. Suppose $S_1 \underset{\text{gen}}{\leqq} S_2$ and $H_1 \in \mathcal{H}_P$, H_1 for S_1. Then $\exists H_2$ for S_2 such that (H_1/P) and (H_2/P) are isomorphic and the outputs of S_1 on H_1 and of S_2 on H_2 correspond. But (H_2/P) is free, and hence by the definition of $\mathcal{H}_P$, $H_2 \in \mathcal{H}_P$. This gives the desired result, i.e., that $S_1 \underset{\text{gen}}{\leqq} S_2$ implies that $S_1 \underset{\text{gen}}{\leqq} S_2$ on $\mathcal{H}_P$.

If. Suppose $S_1 \underset{\text{gen}}{\leqq} S_2$ on $\mathcal{H}_P$ and I_1 is an interpretation for S_1. As S_1 is a Herbrand schema, $\exists H_1$ for S_1 such that $(I_1/P) \overset{\text{h}}{\rightarrow} (H_1/P)$, and $\text{Val}(S_1, I_1)$ corresponds to (the term) $\text{Val}(S_1, H_1)$, or both diverge. As $S_1 \underset{\text{gen}}{\leqq} S_2$ on $\mathcal{H}_P$, $\exists H_2 \in \mathcal{H}_P$, H_2/P isomorphic to H_1/P, H_2 for S_2 such that $\text{Val}(S_1, H_1)$ corresponds to $\text{Val}(S_2, H_2)$ in the isomorphism. By properties (b), (c) in the definition of Herbrand schemas, $\exists I_2$ for S_2 such that $\exists$ an isomorphism $\theta : (I_1/P) \leftrightarrow (I_2/P)$, and $\text{Val}(S_2, I_2)$ corresponds to $\text{Val}(S_2, H_2)$, i.e., $\text{Val}(S_2, I_2) = \theta(\text{Val}(S_1, I_1))$, or both are undefined. □

It may be noted that we defined $\underset{\text{gen}}{\geqq}$, $\geqq$, uninterpreted schemas, and Herbrand schemas using the notion of isomorphism instead of equality, which would have served equally well in our formalism. We anticipate, however, the use of these notions in other contexts. For example, if a schema were constrained such that its interpretations could only be over the natural numbers, we would still consider it uninterpreted. And if another schema always took interpretations on the domain of, say, lists, we might still consider the two to be equivalent if they always computed corresponding values.

4. Translating conventional schemas into generalized schemas. Conventional flowchart schemas are assumed to have function and predicate symbols in V; we reserve the symbols in V' for control purposes. Given a conventional

schema S^* having function and predicate symbols in P, we translate it into a well-founded generalized schema $S = \langle F, \phi, P \rangle$ where for each statement in S^* there is one statement in S. Usually S^* is uninterpreted, in which case S will also be uninterpreted (in fact $\langle F, \phi, Q \rangle$ will be uninterpreted for every finite $Q \subset V$, where $P \subset Q$) and if S^* is a Herbrand schema (for definition see Chandra [2]) so is S. We give here only a few examples of the translation, and it may be checked that the notions of halting, divergence, inclusion, equivalence, freedom, etc., for conventional schemas correspond to those for the generalized schemas. For translation of other features, e.g., arrays, queues, see [2].

4.1. One-variable schemas. Given a conventional flowchart schema S^* with one variable y (S^* may have equality tests but no Boolean variable, counter, etc.), the corresponding generalized schema is $S = \langle F, \text{true}, P \rangle$ where F is identical to the flowchart of S^*, and P is the set of function and predicate symbols in S^*.

4.2. *n*-variable schemas. If S^* has n variables $y_1, \cdots, y_n$ and predicate and function symbols P, to construct $S = \langle F, \phi, P \rangle$, we add $n+1$ new functions *comb*, $v_1, \cdots, v_n$ from V'. ϕ is

$$\begin{aligned} \forall x_1 \cdots \forall x_n \quad & v_1(comb(x_1, \cdots, x_n)) = x_1 \\ & \wedge \cdots \\ & \wedge v_n(comb(x_1, \cdots, x_n)) = x_n. \end{aligned}$$

To construct F we first define the translation $T(\tau(y_1, \cdots, y_n))$ of a term $\tau(y_1, \cdots, y_n)$ as follows:

(i) $T(\tau(\cdot)) = \tau(\cdot)$, $\tau(\cdot)$ has no variables,
(ii) $T(y_i) = v_i(y)$,
(iii) $T(f(\tau_1, \cdots, \tau_k)) = f(T(\tau_1), \cdots, T(\tau_k))$.

Any assignment $\langle y_1, \cdots, y_n \rangle \leftarrow \langle \tau_1, \cdots, \tau_n \rangle$ in S^* is replaced in F by $y \leftarrow comb(T(\tau_1), \cdots, T(\tau_n))$, any test $p(\tau_1, \cdots, \tau_k)$ (resp. $\tau_1 = \tau_2$) in S^* is replaced by $p(T(\tau_1), \cdots, T(\tau_k))$ (resp. $T(\tau_1) = T(\tau_2)$), and HALT(τ) is replaced by HALT($T(\tau)$). Note—we assume all variables in S^* are initialized at the beginning.

4.3. Counters and stacks. If S^* has n variables, m counters $c_1, \cdots, c_m$, and k stacks $s_1, \cdots, s_k$, $S = \langle F, \phi, P \rangle$ has $n+m+k+8$ new functions from V'. Let l denote $n+m+k$. ϕ is

$$\begin{aligned} \forall x_1 \cdots \forall x_l \quad & v_1(comb(x_1, \cdots, x_l)) = x_1 \\ & \wedge \cdots \\ & \wedge v_l(comb(x_1, \cdots, x_l)) = x_l \\ \wedge \forall x \; & plusone(x) \neq x \\ & \wedge minusone(plusone(x)) = x \\ & \wedge minusone(zero) = zero \end{aligned}$$

$$\wedge \forall s \forall x\, push(s, x) \neq \Lambda$$
$$\wedge\, top(push(s, x)) = x$$
$$\wedge\, pop(push(s, x)) = s.$$

Assignment to variables is handled as before. Any assignment $c_i \leftarrow c_i + 1$ is replaced by $y \leftarrow comb(v_1(y), \cdots, plusone(v_{n+i}(y)), \cdots, v_l(y))$, and similarly for $c_i \leftarrow c_i - 1$. Any test $c_i = 0$ is replaced by $v_{n+i}(y) = zero$. If a term τ is pushed into a stack s_i, the corresponding statement is $y \leftarrow comb(v_1(y), \cdots, push(v_{n+m+i}(y), T(\tau)), \cdots, v_l(y))$, and similarly for popping stacks and testing for emptiness (we assume an empty stack is never popped).

The class of generalized schemas corresponding to the conventional flowchart schemas with counters, stacks and equality will be called $\mathscr{C}(c, s, e)$.

5. Maximal schemas. In this section we consider the power of uninterpreted schemas.

DEFINITION. We say a formula ψ is *over* a set P if it contains no function or predicate symbol other than those in P.

DEFINITION. A schema $S = \langle F, \phi, P\rangle$ is a *nonoracle schema* if

(a) for every path in F from the start statement to a test statement, there is a quantifier free formula $\psi(\cdot)$ over P such that for every interpretation I for S, if the computation of S on I follows this path, the test yields a true outcome if and only if $\psi(\cdot)$ is true in I, and

(b) for every path in F from the start statement to a halt statement, there is a quantifier free formula $\psi(x)$ over P such that for every interpretation I for S, if the computation of S on I follows this path, for all elements v in I, the output is v if and only if $\psi(v)$ is true.

LEMMA 2. *Every well-founded schema is a nonoracle schema.*

Proof. Given a well-founded schema $S = \langle F, \phi, P\rangle$ and a path in F from the start statement to a test statement, we can represent the conjunction of all tests (tests $\alpha(y)$ are changed to $\alpha'(\cdot)$ by substituting the value of y) executed along this path (or their negations if the false exit is taken by the path) by a formula ϕ_1. Then for every interpretation I on which the computation of S follows this path, I satisfies $\phi \wedge \phi_1$, and the computation on every interpretation I where $\Sigma(S) \subset \Sigma(I)$ and I satisfies $\phi \wedge \phi_1$, follows this path. Also, the test can be represented by an atomic formula α over $\Sigma(S)$ with no variable. By the well-foundedness of S, whenever $I_1 \models \phi \wedge \phi_1$, $I_2 \models \phi \wedge \phi_1$, (I_1/P) isomorphic to (I_2/P) we have $I_1 \models \alpha$ if and only if $I_2 \models \alpha$, and hence by minor modifications of Lemma 4 in Shoenfield [15, § 5.5], there is a quantifier free formula $\psi(\cdot)$ over P such that $\phi \wedge \phi_1 \rightarrow (\alpha \leftrightarrow \psi(\cdot))$ is valid, i.e., $\phi \wedge \phi_1 \models \alpha \leftrightarrow \psi(\cdot)$.

If, on the other hand, the given path in F leads to a halt statement, then the output is some (constant) term $\tau(\cdot)$. If we now introduce a new zero-ary function a_0 into interpretations for the schema, whenever $I_1 \models \phi \wedge \phi_1$, $I_2 \models \phi \wedge \phi_1$, $(I_1/P \cup \{a_0\})$ isomorphic to $(I_2/P \cup \{a_0\})$, we have $I_1 \models a_0 = \tau(\cdot)$ if and only if $I_2 \models a_0 = \tau(\cdot)$ by the well-foundedness of S, and hence there is a quantifier-free formula $\psi(a_0)$ over $P \cup \{a_0\}$ (we call it $\psi(a_0)$ instead of ψ for convenience) such that $\phi \wedge \phi_1 \rightarrow (a_0 = \tau(\cdot) \leftrightarrow \psi(a_0))$ is valid. But a_0 doesn't appear in $\phi \wedge \phi_1$, and

hence $\phi \wedge \phi_1 \rightarrow \forall x(x = \tau(\cdot) \leftrightarrow \psi(x))$ is valid, i.e., $\phi \wedge \phi_1 \models \forall x(x = \tau(\cdot) \leftrightarrow \psi(x))$, which is the desired result. □

THEOREM 3 (Theorem of maximal schemas). *Every uninterpreted schema can be effectively translated into an equivalent schema in* $\mathscr{C}(c, s, e)$.

Note—every schema in $\mathscr{C}(c, s, e)$ is uninterpreted. Also, it is not partially decidable if a given schema is uninterpreted.

Outline of proof. Schemas in $\mathscr{C}(c, s, e)$ can simulate Turing machine computations. Given an uninterpreted schema $S = \langle F, \phi, P \rangle$, we construct an equivalent schema $S' \in \mathscr{C}(c, s, e)$ as follows. S' simulates the computation of S, and keeps track of the path followed. Given a path in F from the start statement to a test, the path predicate is denoted by ϕ_1 (as in the proof of Lemma 2), and the test can be represented by a variable-free atomic formula α over $\Sigma(S)$. By Lemma 2, there is a quantifier-free formula $\psi(\cdot)$ over P such that $\phi \wedge \phi_1 \rightarrow (\alpha \leftrightarrow \psi(\cdot))$ is valid. S' enumerates the valid formulas to determine an appropriate $\psi(\cdot)$. Using its pushdown stack, it then checks to see whether or not $\psi(\cdot)$ is true in the interpretation, and thereby knows which exit the schema S would take from this test.

On the other hand, if the given path in S reaches a halt statement, we again use ϕ_1 for the path predicate; and $\tau(\cdot)$ (over $\Sigma(S)$ for the output term. By Lemma 2, there is a quantifier-free formula $\psi(x)$ over P such that $\phi \wedge \phi_1 \rightarrow \forall x(x = \tau(x) \leftrightarrow \psi(x))$ is valid. S' enumerates the valid formulas to determine an appropriate $\psi(x)$. It then enumerates all terms $\tau(\cdot)$ over P and checks if $\psi(\tau(\cdot))$ is true in the interpretation. It outputs the first term for which it is true. □

Let $\mathscr{C}(c, s)$ denote the class of generalized schemas corresponding to conventional schemas with counters and stacks but no equality tests. Every schema in $\mathscr{C}(c, s)$ is an uninterpreted Herbrand schema.

THEOREM 4 (Theorem of maximal Herbrand schemas). *Every uninterpreted Herbrand schema can be effectively translated into an equivalent schema in* $\mathscr{C}(c, s)$.

Outline of proof. Schemas in $\mathscr{C}(c, s)$ can simulate Turing machine computations. We construct an $S' \in \mathscr{C}(c, s)$ equivalent to a given uninterpreted Herbrand schema $S = \langle F, \phi, P \rangle$ as in the proof of Theorem 3. In the case of a path to a test statement, S' determines if $\psi(\cdot)$ is true in a Herbrand interpretation corresponding to the given interpretation. It therefore has to make no equality tests since $\tau_1 = \tau_2$ in a Herbrand interpretation if and only if τ_1 and τ_2 are identical. Since S is a Herbrand schema, the outcome of the test on the given interpretation is the same as that on a corresponding Herbrand interpretation.

The case for a path to a halt statement is similar. □

6. Decision problems. We consider the following decision problems (and their complements) for generalized schemas:

(a) the halting problem—to decide if a given schema halts on all interpretations for it;
(b) the divergence problem—to decide if a given schema diverges on all interpretations for it;

(c) the generalization problem—given two schemas S_1, S_2, to decide if $S_2 \underset{\text{gen}}{\geqq} S_1$;

(d) the inclusion problem—given two schemas S_1, S_2, to decide if $S_2 \geqq S_1$;

(e) the equivalence problem—given two schemas S_1, S_2, to decide if $S_2 \equiv S_1$.

THEOREM 5. *The halting problem is partially decidable, its complement is not. All the problems* (b)–(c) *and their complements are not partially decidable.*

This is the same as for conventional schemas, except that for conventional schemas the complement of the divergence problem is partially decidable.

Proof. The undecidability of the halting problem follows from the undecidability of the halting problem for conventional schemas [7]. For partial decidability, Manna [8] shows how to construct a first order formula ψ_F for any flowchart F such that the computation of F halts on every interpretation if and only if ψ_F is valid. A similar proof shows that a schema $S = \langle F, \phi, P \rangle$ halts on every interpretation for it if and only if $\phi \rightarrow \psi_F$ is valid, i.e., it is partially decidable.

That the divergence problem is not partially decidable follows from the corresponding result for conventional schemas [7]. Its complement is also not partially decidable because $\langle F, \phi, \{a\} \rangle$ where F is "START $y \leftarrow a$; HALT (y)" does not always diverge if and only if ϕ is satisfiable.

For uninterpreted schemas, $S_1 \underset{\text{gen}}{\geqq} S_2$ if and only if $S_1 \equiv S_2$, hence parts (c), (d), (e) follow from the corresponding results for conventional schemas [7]. □

Acknowledgment. The author would like to thank Zohar Manna for his inspiration and guidance. Thanks are also due to Paul C. Eklof for his help in the proof of Lemma 2.

REFERENCES

[1] S. BROWN, D. GRIES AND T. SZYMANSKI, *Program schemes with pushdown stores*, this Journal, 1 (1972), pp. 242–268.

[2] A. K. CHANDRA, *On the properties and applications of program schemas*, Ph.D. thesis, Comput. Sci. Rep. CS-336, AIM-188, Stanford Univ., Stanford, Calif., 1973.

[3] A. K. CHANDRA AND Z. MANNA, *Program schemas with equality*, 4th Ann. ACM Symp. on Theory of Computing, Denver, May 1972, pp. 52–64.

[4] R. L. CONSTABLE AND D. GRIES, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[5] S. J. GARLAND AND D. C. LUCKHAM, *Program schemes, recursion schemes, and formal languages*, J. Comput. System Sci., 7 (1973), pp. 119–160.

[6] I. IANOV, *The logical schemes of algorithms*, Problems in Cybernetics, 1, Pergamon Press, New York, 1960, pp. 82–140.

[7] D. C. LUCKHAM, D. M. R. PARK AND M. S. PATERSON, *On formalized computer programs*, J. Comput. System Sci., 4 (1970), pp. 220–249.

[8] Z. MANNA, *Properties of programs and first-order predicate calculus*, J. Assoc. Comput. Mach., 16 (1969), pp. 244–255.

[9] J. MCCARTHY, *Towards a mathematical science of computation*, Proc. IFIP, 1962, pp. 21–28.

[10] M. S. PATERSON, *Equivalence problems in a model of computation*, Ph.D. thesis, Univ. of Cambridge, England, 1967; also Artificial Intelligence Memo. no. 1, MIT, 1970.

[11] M. S. PATERSON AND C. E. HEWITT, *Comparative schematology*, Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, June 1970, pp. 119–127.
[12] D. PLAISTED, *Flowchart schemata with counters*, 4th Ann. ACM Symp. on Theory of Computing, Denver, May 1972, pp. 44–51.
[13] H. ROGERS, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.
[14] J. D. RUTLEDGE, *On Ianov's program schemata*, J. Assoc. Comput. Mach., 11 (1964), pp. 1–9.
[15] J. R. SHOENFIELD, *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.
[16] H. R. STRONG, *High level languages of maximum power*, Proc. IEEE Conf. on Switching and Automata Theory, 1971, pp. 1–4.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

THE THEORETICAL ASPECTS OF THE OPTIMAL FIXEDPOINT*

ZOHAR MANNA† AND ADI SHAMIR‡

Abstract. In this paper we define a new type of fixedpoint of recursive definitions and investigate some of its properties. This *optimal fixedpoint* (which always uniquely exists) contains, in some sense, the maximal amount of "interesting" information which can be extracted from the recursive definition, and it may be strictly more defined than the program's least fixedpoint. This fixedpoint can be the basis for assigning a new semantics to recursive programs.

Key words. recursive definitions, fixedpoints, optimal fixedpoints

Introduction. Recursive definitions are usually considered from two different points of view, namely:

(i) as an algorithm for computing a function, by repeated substitutions of the function definition for its name;

(ii) as a functional equation, expressing the required relations between values of the defined function for various arguments. A function that satisfies these relations (a solution of the equation) is called a *fixedpoint.*

The functional equation represented by a recursive definition may have many fixedpoints, all of which satisfy the relations dictated by the definition. There is no a priori preferred solution and therefore, if the definition has more than one fixedpoint, one of them must be chosen. A number of works describing a *least (defined) fixedpoint* approach towards the semantics of recursive definitions have been published recently (e.g., Scott [8]). Researchers in the field have chosen the least fixedpoint as the "best solution" for three reasons:

(i) It uniquely exists for a wide class of practically applicable recursive definitions.

(ii) The classical stack implementation technique computes this fixedpoint for any recursive definition.

(iii) There is a powerful method (computational induction) for proving properties of this fixedpoint.

However, as a mathematical model for extracting information from an implicit functional equation, the selection of the least defined solution seems a poor choice; for many recursive definitions, the least fixedpoint does not reveal all the useful information embedded in the definition. In general, the more defined the solution, the more valuable it is. On the other hand, this argument should be applied with caution, as there are inherently underdefined recursive definitions. Consider the extreme example $F(x) \Leftarrow F(x)$, for which any partial function is a solution. A randomly chosen total function is by no means superior to the totally undefined least fixedpoint in this case.

The *optimal fixedpoint,* defined in this paper, tries to remedy this situation. It is intended to supply the maximally defined solution relevant to the given

* Received by the editors December 31, 1975.

† Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. Now at Artificial Intelligence Laboratory, Stanford University, Stanford, California 94305.

‡ Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. Now at Computer Science Department, University of Warwick, Coventry, CV4 7AL, England.

recursive definition. Consider, for example, the following recursive definition for solving the discrete form of the Laplace equation, where $F(x, y)$ maps pairs of integers in $[-100, 100]\times[-100, 100]$ into reals:

$$F(x, y) \Leftarrow \textbf{if } x<-100 \vee x>100 \vee y<-100 \vee y>100$$
$$\textbf{then } x^2+y^2$$
$$\textbf{else } \tfrac{1}{4}[F(x-1, y)+F(x+1, y)+F(x, y-1)+F(x, y+1)].$$

This concise organization of knowledge is defined enough to have a unique total fixedpoint (which is our optimal fixedpoint), but its least fixedpoint is totally undefined inside the square $[-100,100]\times[-100,100]$.

While the notion of the optimal fixedpoint is theoretically well-defined, its computation aspects contain many pitfalls, since the optimal fixedpoints of certain recursive definitions are noncomputable partial functions. We do not pursue in this paper the practical aspects of the optimal fixedpoint approach; in Manna and Shamir [4], [5], and in more detail in Shamir [9], we suggest several techniques directed toward the computation of the optimal fixedpoint.

In § 1 of this paper, a few structural properties of the set of all fixedpoints of recursive definitions are proven. The optimal fixedpoint is then introduced (in § 2) as the formalization of our intuitive notion of the "best solution" of recursive definitions. The existence of a unique optimal fixedpoint for any recursive definition, as well as some of its properties, are established. In § 3 we consider the computability (from the point of view of recursive function theory) of the optimal fixedpoint of recursive definitions.

An informal exposition of the main ideas and philosophies of the optimal fixedpoint approach is contained in [5]. A more complete investigation of the various fixedpoints (including the optimal fixedpoint) of recursive definitions appears in [9]. Results which are somewhat related to this work have been obtained by Myhill [6], who investigated ways in which *total functions* can be defined by systems of formulas.

1. Some structural properties of the set of fixedpoints. In this part we introduce our terminology and prove those structural properties of the set of fixedpoints of recursive definitions which are needed in § 2.

1.1. Basic definitions. Let D^+ be a domain of defined values D to which the "undefined element" ω is added. The identity relation over D^+ is denoted by $\equiv$. The set of all mappings of $(D^+)^n$ into D^+ is called the set of *partial functions* of n arguments over D, and is denoted by $\mathrm{PF}(D, n)$.

The binary relation "less defined or equal", $\sqsubseteq$, over various domains plays a fundamental role in the theory.

DEFINITIONS.

(a) For $x, y \in D^+$, $x \sqsubseteq y$ if $x \equiv \omega$ or $x \equiv y$.

(b) For $\bar{x}, \bar{y} \in (D^+)^n$, $\bar{x} \sqsubseteq \bar{y}$ if $x_i \sqsubseteq y_i$ for all $1 \leqq i \leqq n$.

(c) For $f_1, f_2 \in \mathrm{PF}(D, n)$, $f_1 \sqsubseteq f_2$ if $f_1(\bar{x}) \sqsubseteq f_2(\bar{x})$ for every $\bar{x} \in (D^+)^n$.

(d) A function $f \in \mathrm{PF}(D, n)$ is *monotonic* if $\bar{x} \sqsubseteq \bar{y} \Rightarrow f(\bar{x}) \sqsubseteq f(\bar{y})$.

The relation $\sqsubseteq$ is a partial ordering of $\mathrm{PF}(D, n)$. We shall henceforth use the

standard terminology concerning partially ordered sets. In particular:

DEFINITIONS. For any subset S of $\mathrm{PF}(D, n)$:

(a) $f \in S$ is the *least element* of S if $f \sqsubseteq g$ for any $g \in S$.

(b) $f \in S$ is a *minimal element* of S if there is no $g \in S$ which satisfies $g \sqsubset f$.

(c) $f \in \mathrm{PF}(D, n)$ is an *upper bound* of S if $g \sqsubseteq f$ for all $g \in S$.

(d) $f \in \mathrm{PF}(D, n)$ is the *least upper bound* (lub) of S if f is the least element in the set of upper bounds of S.

The notions of the *greatest element*, a *maximal element*, a *lower bound* and the *greatest lower bound* (glb) of S are dually defined.

DEFINITIONS.

(a) $f, g \in \mathrm{PF}(D, n)$ are *consistent* if $f(\bar{x}) \neq \omega$ and $g(\bar{x}) \neq \omega \Rightarrow f(\bar{x}) \equiv g(\bar{x})$ for every $\bar{x} \in (D^+)^n$.

(b) A subset S of $\mathrm{PF}(D, n)$ is *consistent* if every two functions $f, g \in S$ are consistent.

From the definition it follows that:

(i) A subset S of $\mathrm{PF}(D, n)$ has a lub, denoted by **lub** S, if and only if S is consistent.

(ii) Every nonempty subset S of $\mathrm{PF}(D, n)$ has a glb, which is denoted by **glb** S.

DEFINITIONS.

(a) A *functional* is a mapping of $\mathrm{PF}(D, n)$ into $\mathrm{PF}(D, n)$.

(b) A functional τ over $\mathrm{PF}(D, n)$ is *monotonic* if $f \sqsubseteq g \Rightarrow \tau[f] \sqsubseteq \tau[g]$ for every $f, g \in \mathrm{PF}(D, n)$.

(c) A *recursive definition* is of the form $F(\bar{x}) \Leftarrow \tau[F](\bar{x})$, where τ is a functional and F is a function variable.

All the functionals we shall deal with in this paper will be monotonic over $\mathrm{PF}(D, n)$. In practice, there are many types of functionals which are monotonic only over a certain subset S of $\mathrm{PF}(D, n)$. The theory developed in this paper can be applied to any such restricted functional, provided that S satisfies the following two conditions:

(i) any consistent subset of S has a lub in S, and

(ii) any nonempty subset of S has a glb in S.

For simplicity, we do not consider in this part functions over multiple domains (e.g., $D_1^+ \times \cdots \times D_n^+ \to D^+$) or systems of functionals (e.g., $(\tau_1, \cdots, \tau_k)$). However, all the results can be extended easily to the more general cases.

1.2. Fixedpoints, pre-fixedpoints, and post-fixedpoints.

DEFINITION. A function $f \in \mathrm{PF}(D, n)$ is a *fixedpoint*, *pre-fixedpoint*, or *post-fixedpoint* of τ if $f \equiv \tau[f]$, $f \sqsubseteq \tau[f]$, or $\tau[f] \sqsubseteq f$, respectively. The sets of all fixedpoints, pre-fixedpoints, or post-fixedpoints of τ are denoted by $\mathrm{FXP}(\tau)$, $\mathrm{PRE}(\tau)$ or $\mathrm{POST}(\tau)$, respectively.

Clearly $\mathrm{FXP}(\tau) = \mathrm{PRE}(\tau) \cap \mathrm{POST}(\tau)$. A few useful properties of these sets for a monotonic functional τ are:

(i) $\mathrm{FXP}(\tau)$, $\mathrm{PRE}(\tau)$, and $\mathrm{POST}(\tau)$ are closed under the application of τ.

(ii) If $S \subseteq \mathrm{PRE}(\tau)$ is consistent, then **lub** $S \in \mathrm{PRE}(\tau)$.

(iii) If $S \subseteq \mathrm{POST}(\tau)$ is nonempty, then **glb** $S \in \mathrm{POST}(\tau)$.

The most important property of pre- and post-fixedpoints is that they enable

us to uniformly approach a fixedpoint of τ, either by monotonically ascending or by monotonically descending to it. The theoretical background of this process is contained in the theorem:

THEOREM 1 (Hitchcock and Park). *Let* $(S, \preccurlyeq)$ *be a partially ordered set, with a least element* Ω, *and such that any totally ordered subset has a* lub. *Then for any monotonic mapping* $\tau: S \to S$, *the set of fixedpoints of* τ *contains a least element.*

A formal proof, using a transfinite sequence of approximations $\tau^{(\lambda)}(\Omega)$ which converges to the least fixedpoint of τ, appears in Hitchcock and Park [1]. An immediate corollary of Theorem 1 is:

THEOREM 2. *For monotonic functional* τ:

(a) $\mathrm{FXP}(\tau)$ *contains a least element, denoted by* $\mathbf{lfxp}(\tau)$.

(b) *If* $f \in \mathrm{PRE}(\tau)$, *then the set* $\{f' \in \mathrm{FXP}(\tau) | f \sqsubseteq f'\}$ *contains a least element.*

(c) *If* $f \in \mathrm{POST}(\tau)$, *then the set* $\{f' \in \mathrm{FXP}(\tau) | f' \sqsubseteq f\}$ *contains a greatest element.*

Proof.

(a) This is immediate by Theorem 1, taking $\mathrm{PF}(D, n)$ as S, $\sqsubseteq$ as $\preccurlyeq$, and the totally undefined function as Ω.

(b) Define $S_f \equiv \{f' \in \mathrm{PF}(D, n) | f \sqsubseteq f'\}$. S_f is partially ordered by $\sqsubseteq$, and contains f as its least element. Since any totally ordered subset S of S_f is consistent, **lub** S exists. Furthermore, **lub** $S \in S_f$ since $f \sqsubseteq$ **lub** S.

The given monotonic functional τ maps $\mathrm{PF}(D, n)$ into $\mathrm{PF}(D, n)$. It is easy to show that τ maps S_f into itself. Therefore, we may consider the monotonic functional τ' mapping S_f into S_f, which is the restriction of τ to S_f. Theorem 1 ensures the existence of a least fixedpoint for τ', which is exactly the fixedpoint required.

(c) Using the reverse order, i.e., $f_1 \preccurlyeq f_2$ iff $f_2 \sqsubseteq f_1$, a proof dual to the proof of part (b) can be obtained. Q.E.D.

DEFINITION. A fixedpoint f of τ is FXP-*consistent* if for any $f' \in \mathrm{FXP}(\tau)$, f and f' are consistent. The set of all FXP-consistent fixedpoints of τ is denoted by $\mathrm{FXPC}(\tau)$.

From the definition, it follows that for any monotonic functional τ:

(i) Since $\mathbf{lfxp}(\tau)$ is FXP-consistent, $\mathrm{FXPC}(\tau)$ is nonempty.

(ii) Since any two FXP-consistent fixedpoints are consistent, $\mathrm{FXPC}(\tau)$ is consistent, and thus **lub**$\mathrm{FXPC}(\tau)$ exists.

THEOREM 3. *For a monotonic functional* τ, $\mathrm{FXPC}(\tau)$ *contains a greatest element.*

Proof. We know that $f_1 \equiv$ **lub** $\mathrm{FXPC}(\tau)$ exists. As a lub of fixedpoints, $f_1 \in \mathrm{PRE}(\tau)$. Thus, by Theorem 2(b), the set $\{f' \in \mathrm{FXP}(\tau) | f_1 \sqsubseteq f'\}$ contains a least element, say f_2. We show now that $f_2 \in \mathrm{FXPC}(\tau)$, implying that f_2 is the greatest function in $\mathrm{FXPC}(\tau)$.

Let g be any fixedpoint of τ. We would like to prove that f_2 and g are consistent, by showing the existence of a function f_3 such that $f_2 \sqsubseteq f_3$ and $g \sqsubseteq f_3$. The set of fixedpoints $S = \mathrm{FXPC}(\tau) \cup \{g\}$ is consistent by the definition of $\mathrm{FXPC}(\tau)$, and therefore by Theorem 2(b) again there exists some $f_3 \in \mathrm{FXP}(\tau)$ such that **lub** $S \sqsubseteq f_3$. Thus, $g \sqsubseteq f_3$ and **lub** $\mathrm{FXPC}(\tau) \sqsubseteq f_3$. Since f_2 was defined as the least fixedpoint such that **lub** $\mathrm{FXPC}(\tau) \sqsubseteq f_2$, we have $f_2 \sqsubseteq f_3$. Q.E.D.

1.3. Maximal fixedpoints.

DEFINITION. A fixedpoint f of a functional τ is said to be *maximal* if there is no other fixedpoint g which satisfies $f \sqsubset g$. The set of all maximal fixedpoints of τ is denoted by $\mathrm{MAX}(\tau)$.

Unlike the case of minimal fixedpoints, a monotonic functional may have any number of maximal fixedpoints. $\mathrm{MAX}(\tau)$ "covers" $\mathrm{FXP}(\tau)$ in the sense of:

THEOREM 4. *For monotonic functional τ, if $f \in \mathrm{PRE}(\tau)$, then $f \sqsubseteq g$ for some $g \in \mathrm{MAX}(\tau)$.*

In other words, if $f(\bar{d}) \equiv c$ for some $f \in \mathrm{PRE}(\tau)$, $\bar{d} \in (D^+)^n$ and $c \in D$, then there must exist $g \in \mathrm{MAX}(\tau)$ such that $g(\bar{d}) \equiv c$.

Proof. Let $S_f = \{f' \in \mathrm{FXP}(\tau) | f \sqsubseteq f'\}$. By Theorem 2(b), S_f contains at least one element—the least fixedpoint which is more defined than f.

We now show that S_f contains an upper bound for any totally ordered subset. Let S be such a subset. Since it is totally ordered, it is in particular consistent and thus **lub** S exists. Furthermore, as a lub of fixedpoints, **lub** S is a pre-fixedpoint. Using Theorem 2(b) once more, there is a fixedpoint f_1 which is more defined than **lub** S, i.e., which is an upper bound of S. By the definition of S and S_f, $f_1 \in S_f$ and thus S has an upper bound in S_f.

We have thus shown that S_f is nonempty and contains an upper bound for any totally ordered subset in it. By Zorn's lemma, any partially ordered set having these two properties contains a maximal element. This maximal element g is clearly a maximal fixedpoint of τ, and $f \sqsubseteq g$ by the definition of S_f. Q.E.D.

As a result of Theorem 4, we obtain a

COROLLARY. *For any monotonic functional τ, $\mathrm{MAX}(\tau)$ is nonempty.*

Proof. The proof follows by the fact that $\mathrm{PRE}(\tau)$ is nonempty, since the totally undefined function Ω is always in $\mathrm{PRE}(\tau)$. Q.E.D.

We also have

THEOREM 5. *For a monotonic functional τ, if $f \in \mathrm{PRE}(\tau)$ and $g \in \mathrm{MAX}(\tau)$, then either $f \sqsubseteq g$ or f and g are not consistent.*

Proof. By contradiction, suppose $f \not\sqsubseteq g$, and f and g are consistent. Then $f_1 \equiv$ **lub** $\{f, g\}$ exists and $g \sqsubset f_1 \in \mathrm{PRE}(\tau)$. Thus by Theorem 2(b) there is a fixedpoint f_2 such that $f_1 \sqsubseteq f_2$. Therefore, $g \sqsubset f_2$, which contradicts the maximality of g. Q.E.D.

From Theorem 5 we obtain a

COROLLARY. *Any two distinct maximal fixedpoints of τ are not consistent.*

Proof. If $f, g \in \mathrm{MAX}(\tau)$, then in particular $f \in \mathrm{PRE}(\tau)$ and we can thus apply Theorem 5. The possibility $f \sqsubseteq g$ is ruled out by the maximality of f, and thus f and g are nonconsistent. Q.E.D.

2. The optimal fixedpoint.

2.1. Definition and properties. By its definition, an FXP-consistent fixedpoint is a function which agrees in value with every other fixedpoint of τ for any argument. In particular, if such a fixedpoint has a defined value c at argument $\bar{d}$, then there can be no fixedpoint of τ which has a different defined value c' at $\bar{d}$. This value c is then said to be *weakly defined* by τ at $\bar{d}$ (it is not "strongly defined", however, since there may be fixedpoints that are not defined at all at $\bar{d}$). A fixedpoint which is not FXP-consistent, on the other hand, represents some

random selection of values from the many which are possible. It is in this sense that we may say that a recursive definition really "well defines" only its FXP-consistent solutions.

Among these "genuine" solutions of τ, the more defined the solution, the more informative it is. Motivated by this quality criterion, we introduce our main definition:

DEFINITION: The *optimal fixedpoint* of a monotonic functional τ is its greatest FXP-consistent fixedpoint. It is denoted by **opt** (τ).

Note that Theorem 3 guarantees the existence of the (uniquely defined) optimal fixedpoint of any monotonic functional. Using the properties of MAX(τ), we can characterize the optimal fixedpoint from a different point of view.

DEFINITION. Since MAX(τ) is nonempty, **glb** MAX(τ) always exists, and is denoted by **lmax** (τ).

As a glb of fixedpoints, **lmax** $(\tau) \in$ POST(τ), but it is not necessarily a fixedpoint. For example, consider the following functional over PF$(N, 1)$:[1]

$$\tau[F](x): \quad \textbf{if } x = 0 \quad \textbf{then } F(x) \quad \textbf{else } 0 \cdot F(x-1).$$

The fixedpoints of τ are the totally undefined function Ω, and all the functions f_i, $i = 0, 1, \cdots$, defined as:

$$f_i(x) \equiv \begin{cases} i & \text{if } x \equiv 0, \\ 0 & \text{otherwise.} \end{cases}$$

It is clear that MAX$(\tau) = \{f_0, f_1, \cdots\}$. The glb of this set of functions is:

$$\textbf{lmax}\,(\tau)(x) \equiv \begin{cases} \omega & \text{if } x \equiv 0, \\ 0 & \text{otherwise.} \end{cases}$$

This function is not a fixedpoint of τ, but is a post-fixedpoint of τ. It descends to the fixedpoint Ω by repeatedly applying τ to it.

However, we show now that the function **lmax** (τ) is closely related to **opt** (τ):

THEOREM 6. *For a monotonic functional* τ, **opt** (τ) *is the greatest element of the set* $\{f' \in \text{FXP}(\tau) | f' \sqsubseteq \textbf{lmax}\,(\tau)\}$.

Proof. Let us denote by f_1 the greatest element in the set. By Theorem 2(c), the function f_1 must exist since **lmax** $(\tau) \in$ POST (τ). We now have to show that **opt** $(\tau) \sqsubseteq f_1$ and $f_1 \sqsubseteq$ **opt**(τ).

To show **opt** $(\tau) \sqsubseteq f_1$, we note that by definition, **opt** (τ) is consistent with any maximal fixedpoint f of τ. By Theorem 5, it follows that **opt** $(\tau) \sqsubseteq f$. Thus, **opt** (τ) is a lower bound of MAX(τ), and therefore **opt** $(\tau) \sqsubseteq$ **lmax** $(\tau) \equiv$ **glb** MAX(τ). Since f_1 is the greatest element of $\{f' \in \text{FXP}(\tau) | f' \sqsubseteq \textbf{lmax}\,(\tau)\}$ we obtain **opt** $(\tau) \sqsubseteq f_1$.

We now show that $f_1 \sqsubseteq$ **opt**(τ). By the definition of **opt**(τ), it suffices to show that $f_1 \in$ FXPC(τ). Let f be any fixedpoint of τ. Theorem 4 implies that there exists some $f_2 \in$ MAX(τ) such that $f \sqsubseteq f_2$. By the definition of f_1, it follows that $f_1 \sqsubseteq f_2$. Thus, f_2 is an upper bound of f and f_1, which implies that they are consistent. Since this holds for any $f \in$ FXP(τ), $f_1 \in$ FXCP(τ). Q.E.D.

---

[1] N denotes the set of natural numbers.

The original definition of **opt**(τ) and Theorem 6 suggests that **opt** (τ) can be "reached" both from below (by ascending from **lfxp**(τ) as high as possible in FXPC(τ)), or from above (by descending from MAX(τ)). This situation is illustrated by the schematic diagram of Fig. 1. In our graphical representation, the set $\{f' \in \text{FXP}(\tau) | f \sqsubseteq f'\}$ is shown as an upper cone (Fig. 2A), and the set $\{f' \in \text{FXP}(\tau) | f' \sqsubseteq f\}$ is shown as a lower cone (Fig. 2B).

The following properties of **opt**(τ), for a monotonic functional τ, are immediate consequences of its definition and Theorem 6:

(a) If **lfxp**(τ) is a total function, then **opt**(τ) $\equiv$ **lfxp**(τ).

(b) **opt**(τ) $\in$ MAX(τ) if and only if τ has a unique maximal fixedpoint.

It is clear that a necessary condition for **opt**(τ)($\bar{d}$) $\equiv c$ for some $\bar{d} \in (D^+)^n$ and $c \in D$ is:

(i) $f(\bar{d}) \equiv \omega$ or $f(\bar{d}) \equiv c$ for all $f \in \text{FXP}(\tau)$, and

(ii) $f(\bar{d}) \equiv c$ for at least one $f \in \text{FXP}(\tau)$.

However, this condition is not sufficient, as demonstrated in the previous example:

$$\tau[F](x): \quad \textbf{if } x = 0 \quad \textbf{then } F(x) \quad \textbf{else } 0 \cdot F(x-1).$$

All the fixedpoints of τ are either undefined or defined as 0 at $x \equiv 1$ and there are fixedpoints which are defined at $x \equiv 1$, while **opt**(τ)(1) $\equiv \omega$.

2.2. Examples. In this section we illustrate the theory presented in this part with two functionals. These functionals are monotonic only over the subset MON(N, 1) of all monotonic functions in PF(N, 1). Since MON(N, 1) satisfies the

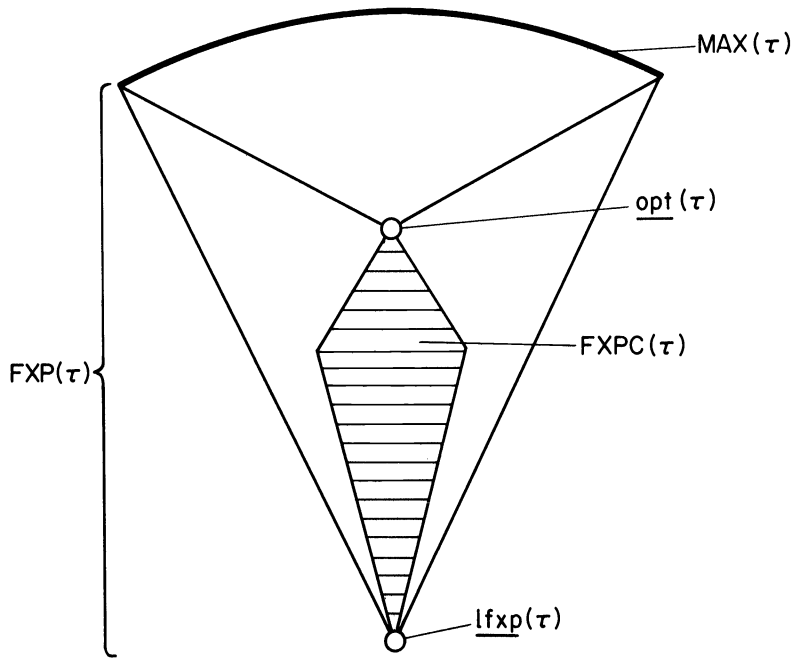


FIG. 1. *The fixedpoints of a recursive program*

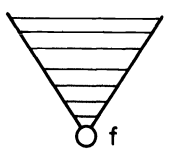


FIG. 2A

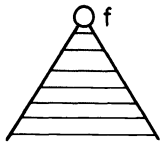


FIG. 2B

two conditions mentioned at the end of § 1.1, we may restrict the discussion to the domain MON(N, 1) rather than PF(N, 1).

Example 1. Consider first the monotonic functional τ_1 over MON(N, 1):

$$\tau_1[F](x): \quad \textbf{if } x=0 \quad \textbf{then } 1 \quad \textbf{else } F(F(x-1)).$$

The least fixedpoint of this functional is

$$\textbf{lfxp}(\tau_1) \equiv \begin{cases} 1 & \text{if } x \equiv 0, \\ \omega & \text{otherwise.} \end{cases}$$

We would like to show that $\textbf{opt}(\tau_1) \equiv \textbf{lfxp}(\tau_1)$. For this purpose, it suffices to find two fixedpoints $f_1, f_2 \in \text{FXP}(\tau_1)$ whose values disagree for any positive x. Two such functions are, for example:

$$f_1(x) \equiv \begin{cases} 1 & \text{if } x \in N, \\ \omega & \text{if } x \equiv \omega, \end{cases}$$

and

$$f_2(x) \equiv \begin{cases} x+1 & \text{if } x \in N, \\ \omega & \text{if } x \equiv \omega. \end{cases}$$

Thus both $\textbf{opt}(\tau_1)$ and $\textbf{lmax}(\tau_1)$ cannot be defined for any positive integer x; since $f(\omega) \equiv \omega$ for any $f \in \text{FXP}(\tau_1)$, we finally obtain that $\textbf{opt}(\tau_1) \equiv \textbf{lmax}(\tau_1) \equiv \textbf{lfxp}(\tau_1)$.

Since $\textbf{lfxp}(\tau_1)$ and $\textbf{opt}(\tau_1)$ are the least and greatest elements of $\text{FXPC}(\tau_1)$, $\textbf{lfxp}(\tau_1)$ is clearly the only element of $\text{FXPC}(\tau_1)$.

The functions f_1 and f_2 above are maximal, since they cannot be extended at $x \equiv \omega$. It is quite an instructive exercise to characterize all the maximal fixedpoints of τ_1. For example, it can be easily shown that any maximal fixedpoint other than f_2 is a total, ultimately periodic function over N.

Example 2. Let us consider now the functional τ_2, defined over the same domain:

$$\tau_2[F](x): \quad \textbf{if } x=0 \quad \textbf{then } 1 \quad \textbf{else } 2F(F(x-1)).$$

One can easily show that $\textbf{lfxp}(\tau_2) \equiv \textbf{lfxp}(\tau_1)$. The fixedpoint $\textbf{opt}(\tau_2)$ cannot be obtained by the technique used in the previous example, since no appropriate fixedpoints f_1 and f_2 can be found. As a matter of fact, this functional has exactly

three fixedpoints:

$$f_1(x) \equiv \begin{cases} 1 & \text{if } x \equiv 0, \\ \omega & \text{otherwise.} \end{cases}$$

$$f_2(x) \equiv \begin{cases} 1 & \text{if } x \equiv 0, \\ 0 & \text{if } x \equiv 1, \\ 2 & \text{if } x \equiv 2, \\ 4 & \text{if } x \equiv 3, \\ \omega & \text{otherwise.} \end{cases}$$

$$f_3(x) \equiv \begin{cases} 1 & \text{if } x \equiv 0, \\ 0 & \text{if } x \equiv 3i+1, \\ 2 & \text{if } x \equiv 3i+2, \\ 4 & \text{if } x \equiv 3i+3, \\ \omega & \text{if } x \equiv \omega. \end{cases} \quad i = 0, 1, 2, \cdots,$$

These fixedpoints are related by $f_1 \sqsubseteq f_2 \sqsubseteq f_3$, and therefore

$$\mathbf{lfxp}(\tau_2) \equiv f_1,$$

$$\mathbf{opt}(\tau_2) \equiv \mathbf{lmax}(\tau_2) \equiv f_3,$$

$$\mathrm{MAX}(\tau_2) \equiv \{f_3\},$$

$$\mathrm{FXPC}(\tau_2) \equiv \mathrm{FXP}(\tau_2) \equiv \{f_1, f_2, f_3\}.$$

3. The computability of optimal fixedpoints. In this part we state several results concerning the computability of optimal fixedpoints over the natural numbers. In our constructions we shall use systems of functionals $\bar{\tau} = (\tau_1, \cdots, \tau_k)$, where each τ_i is a monotonic functional mapping any k-tuple $(f_1, \cdots, f_k)$ of partial functions into a partial function $\tau_i[f_1, \cdots, f_k]$. Thus, $\bar{\tau}$ maps any k-tuple $(f_1, \cdots, f_k)$ of partial functions into the k-tuple $(\tau_1[f_1, \cdots, f_k], \cdots, \tau_k[f_1, \cdots, f_k])$; it represents a system of recursive definitions of the form

$$F_1(\bar{x}) \Leftarrow \tau_1[F_1, \cdots, F_k](\bar{x})$$
$$\vdots$$
$$F_k(\bar{x}) \Leftarrow \tau_k[F_1, \cdots, F_k](\bar{x}).$$

A fixedpoint of $\bar{\tau}$ is now defined as a k-tuple $(f_1 \cdots, f_k)$ mapped by $\bar{\tau}$ to itself. We shall be interested in the computability of the function f_1 appearing as the first element in such a tuple (this function is usually called the *main function*; the others are called the *auxiliary functions*). All the definitions and results contained in §§ 1 and 2 of the paper can be extended easily to this general case.

We first show that the collection of optimal fixedpoints of recursive definitions over the natural numbers contains (as main functions) all the partial computable functions:

THEOREM 7. *Any partial recursive function φ_i over the natural numbers is the optimal fixedpoint of some effectively constructable system of recursive definitions.*

Proof. Any partial recursive function can be computed by a counter machine with two counters (cf. Hopcroft and Ullman [2, p. 98]). Such a machine can be simulated by a system of recursive definitions in the following way.

The input value is stored in variable x_0, and with each counter $c_i (i = 1, 2)$ is associated a variable x_i. The main recursive definition which initializes the counters is

$$F_1(x) \Leftarrow F_2(x, 0, 0).$$

The function variables $F_2, \cdots, F_k$ correspond to the states $q_2, \cdots, q_k$ of the counter machine. The ith ($i \geqq 2$) recursive definition is either of the form

$$F_i(x_0, x_1, x_2) \Leftarrow \textbf{if } x_0 = 0 \textbf{ then } x_1 \textbf{ else } F_m(x_0'', x_1'', x_2''),$$

or of the form (for $j = 1, 2$)

$$F_i(x_0, x_1, x_2) \Leftarrow \textbf{if } x_j = 0 \textbf{ then } F_n(x_0', x_1', x_2') \textbf{ else } F_m(x_0'', x_1'', x_2''),$$

where the indexes n, m are chosen according to the state to which the counter machine transits when it is in state q_i, and counter c_j has the respective value (zero or nonzero). Each transformed variable x' or x'' stands for either $x+1$ or $x-1$, according to the operation done on the counter or the input value upon transition.

The evaluation of the least fixedpoint of this system of recursive definitions is done by repeatedly replacing a term $F_i(x_0, x_1, x_2)$ by the appropriate term $F_n(x_0', x_1', x_2')$ or $F_m(x_0'', x_1'', x_2'')$, thus simulating the state transitions of the counter machine. The process stops if and when a term $F_i(x_0, x_1, x_2)$ is replaced by the term x_1 (according to a definition of the first type), and the current value of x_1 is taken as the result of computation.

Due to the simple nature of these recursive definitions, their optimal fixedpoint coincides with their least fixedpoint (the main function in which is φ_i). To show this, define for any natural number c the following k-tuple of functions $(f_1^c, \cdots, f_k^c)$:

$$f_1^c(x) \equiv \begin{cases} c & \text{if evaluation of } F_1(x) \text{ is nonterminating,} \\ y & \text{if evaluation of } F_1(x) \text{ terminates with value } y, \end{cases}$$

and similarly, for $i \geqq 2$:

$$f_i^c(x_0, x_1, x_2) \equiv \begin{cases} c & \text{if evaluation of } F_i(x_0, x_1, x_2) \text{ is nonterminating,} \\ y & \text{if evaluation of } F_i(x_0, x_1, x_2) \text{ terminates with value } y. \end{cases}$$

For any c, the k-tuple $(f_1^c, \cdots, f_k^c)$ so defined is a fixedpoint of the system. It is a maximal fixedpoint by its totality. The optimal fixedpoint $(f_1, \cdots, f_k)$ is less defined than $(f_1^c, \cdots, f_k^c)$ for all c, and thus $f_1(x)$ cannot be defined if the evaluation of $F_1(x)$ is nonterminating. Q.E.D.

Theorem 7 shows that any function which can be defined as the main function in the least fixedpoint of an effective recursive definition (i.e., any partial recursive function) can also be defined as the main function in the optimal fixedpoint of a (perhaps different) effective recursive definition. The converse, however, is not

true. To show this, it suffices to consider the following simple functional over the natural numbers:

$$\tau[F](x): \quad \textbf{if } F(x)=1 \textbf{ then } h(x) \textbf{ else } 0,$$

where $h(x)$ is the halting function, defined as:

$$h(x) \equiv \begin{cases} 1 & \text{if } \varphi_x(x) \text{ is defined,} \\ \omega & \text{if } \varphi_x(x) \text{ is undefined.} \end{cases}$$

The function $h(x)$ is computable, as are all the other base functions which appear in the definition. In order to find the optimal fixedpoint of τ, we analyze the possible values of $F(x)$ for any x (there is absolutely no relation between values of F for different arguments x). The value of $F(x)$ can always be ω or 0, as a direct substitution shows. The value 1 is possible only if $h(x) \equiv 1$. Any maximal fixedpoint of τ is a composition of values 0 and 1 (only if legal) for the various arguments x. The optimal fixedpoint is then defined as 0 whenever only 0 is a possible value, while it is ω whenever both 0 and 1 are possible values. Thus

$$\textbf{opt}(\tau)(x) \equiv \begin{cases} \omega & \text{if } \varphi_x(x) \text{ is defined,} \\ 0 & \text{if } \varphi_x(x) \text{ is undefined,} \end{cases}$$

and this "inverted halting function" is noncomputable.

In order to see how noncomputable an optimal fixedpoint may be, we prove

THEOREM 8. *Let $f(x_1, \cdots, x_n)$ be a total predicate over the natural numbers,*[2] *which is the main function in the optimal fixedpoint of some system of recursive definitions $(\tau_3, \cdots, \tau_k)$. Then there is a system of recursive definitions $(\tau_1, \tau_2, \tau_3, \cdots, \tau_k)$ such that*

$$\textbf{opt}(\tau_1)(x_2, \cdots, x_n) \equiv (\exists\, x_1 \in N)[f(x_1, \cdots, x_n)].$$

Proof. The two additional recursive definitions τ_1 and τ_2 are given by:

$$F_1(x_2, \cdots, x_n) \Leftarrow F_2(0, x_2, \cdots, x_n),$$

$$F_2(x_1, x_2, \cdots, x_n) \Leftarrow \textbf{if } F_3(x_1, x_2, \cdots, x_n) > 0 \textbf{ then } 1 \textbf{ else } 2 \cdot F_2(x_1+1, x_2, \cdots, x_n).$$

The first definition simply initializes the search conducted by the second definition for a value of x_1 for which $F_3(x_1, x_2, \cdots, x_n)$ is nonzero (**true**). Such a sequential search is legal, because we assume that in the optimal fixedpoint $F_3(x_1, x_2, \cdots, x_n)$ represents a total function. If this search is successful, $F_2(0, x_2, \cdots, x_n)$ (which is the value returned by the main definition τ_1) is 2 to the power of the first such x_1 found, and this value is clearly nonzero.

If no such value x_1 can be found, we claim that the only two possible values of fixedpoints for $F_2(0, x_2, \cdots, x_n)$ are ω and 0. The fact that these are possible values is shown by direct evaluation. Suppose now that there is some other possible defined value c. This value should satisfy $c \equiv 2^{x_1} \cdot F_2(x_1+1, \cdots, x_n)$ for any natural number x_1. If $c > 0$, this cannot hold if x_1 is sufficiently large, no matter

[2] We assume that the truth value **false**/**true** of the predicate is determined by a zero/nonzero value of f.

what the value of $F_2(x_1+1, \cdots, x_n)$ is. Thus by the definition of the optimal fixedpoint, $\mathbf{opt}(\tau_1)(x_2, \cdots, x_n) \equiv 0$ in this case. Q.E.D.

We can now prove

THEOREM 9. *Any (total) predicate $f(x_1, \cdots, x_n)$ in the arithmetic hierarchy of predicates over natural numbers can be defined as the main function in the optimal fixedpoint of some system of recursive definitions.*

Proof. Any such predicate f can be expressed by (see, for example, Rogers [7])

$$f(x_{i+1}, \cdots, x_k) : (\exists\, x_i)(\sim\exists\, x_{i-1}) \cdots (\sim\exists\, x_1)[\varphi_j(x_1, \cdots, x_i, x_{i+1}, \cdots, x_k)],$$

or by

$$f(x_{i+1}, \cdots, x_k) : (\sim\exists\, x_i)(\sim\exists\, x_{i-1}) \cdots (\sim\exists\, x_1)[\varphi_j(x_1, \cdots, x_i, x_{i+1}, \cdots, x_k)],$$

where

$$\varphi_j(x_1, \cdots, x_k) \quad \text{is a recursive predicate.}$$

These two forms can be constructed in the following way. First a system which defines the recursive function $\varphi_j(x_1, \cdots, x_k)$ is constructed (by its totality, one need not use the method described in Theorem 7—any system of recursive definitions which yields φ_j as least fixedpoint also yields it as optimal fixedpoint). Then the pair of recursive definitions described in Theorem 8 is added for each existential quantifier, from right to left. The only change one should make in each pair in order to handle the negation sign is to change the predicate $F_3(x_1, \cdots, x_n) > 0$ into $F_3(x_1, \cdots, x_n) = 0$; thus we search for values which *do not* satisfy the previous existential condition. Finally, if a form of the second type above should be constructed, the following main recursive definition is added:

$$F_0(\bar{x}) \Leftarrow \mathbf{if}\ F_1(\bar{x}) > 0\ \mathbf{then}\ 0\ \mathbf{else}\ 1,$$

and the resultant predicate $F_1(\bar{x})$ is thus inverted in $F_0(\bar{x})$.

The proof that the procedure described above constructs a system of recursive definitions yielding the predicate $f(\bar{x})$ as the main function in the optimal fixedpoint is a straightforward generalization (by induction) of Theorem 8. Q.E.D.

Once we have constructed recursive definitions for all the predicates in the arithmetic hierarchy, we can also construct recursive definitions for all the partial functions whose graph[3] is a predicate of the arithmetic hierarchy.

THEOREM 10. *If $f(\bar{x})$ is a partial function with graph $g(\bar{x}, y)$ in the arithmetic hierarchy, then there exists a system of recursive definitions such that the main function in its optimal fixedpoint is $f(\bar{x})$.*

---

[3] The graph $g(\bar{x}, y)$ of a partial function $f(\bar{x})$ is a predicate defined by:

$$g(\bar{x}, y) \equiv \begin{cases} \mathbf{true} & \text{if } f(x) \equiv y,\ y \not\equiv \omega, \\ \mathbf{false} & \text{if } f(\bar{x}) \not\equiv y,\ y \not\equiv \omega, \\ \omega & \text{if } y \equiv \omega. \end{cases}$$

In particular, if $f(\bar{x})$ is undefined then $g(\bar{x}, y)$ is **false** for all $y \not\equiv \omega$.

Proof. By Theorem 9, there exists a system of recursive definitions $(\tau_3, \cdots, \tau_n)$ for which the main function in the optimal fixedpoint is the (total) function $g(\bar{x}, y)$. The following two recursive definitions τ_1 and τ_2 are added to the system (τ_1 serves as the main definition):

$$F_1(\bar{x}) \Leftarrow F_2(\bar{x}, 0)$$

$$F_2(\bar{x}, y) \Leftarrow \textbf{if } F_3(\bar{x}, y) > 0 \textbf{ then } y \textbf{ else } F_2(\bar{x}, y+1).$$

The proof that $F_1(\bar{x})$ really yields the desired partial function is a mixture of elements from the proofs of Theorems 7 and 8. The recursive definition τ_2 conducts a search (initialized by 0) for a value y which satisfies $F_3(\bar{x}, y) > 0$ (i.e., for which $g(\bar{x}, y)$ is true). If a value y is found, it is taken as the result of computation. Otherwise, due to the simple form of τ_2, any constant value c can serve as a value for a fixedpoint, and thus the main function in the optimal fixedpoint is undefined. Q.E.D.

REFERENCES

[1] P. HITCHCOCK AND D. PARK, *Induction rules and termination proofs*, IRIA Conf. on Automata, Languages and Programming Theory, July 1972, pp. 183–190.

[2] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

[3] Z. MANNA, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[4] Z. MANNA AND A. SHAMIR, *The optimal fixedpoint of recursive programs*, Proc. Symp. on Theory of Computing, Albuquerque, N.M., May 1975.

[5] ———, *A new approach to recursive programs*, CMU 10th Anniversary, Academic Press, New York, to appear (1976).

[6] J. MYHILL, *Finitely representable functions*, Constructivity in Mathematics, A. Heyting, ed., North-Holland, Amsterdam, pp. 195–207

[7] H. ROGERS, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[8] D. SCOTT, *Outline of a Mathematical Theory of Computation*, 4th Ann. Princeton Conf. on Information Sciences and Systems, 1970, pp. 169–176.

[9] A. SHAMIR, *Fixedpoints of recursive programs*, Ph.D. thesis, Weizmann Institute, Rehovot, Israel, to appear, 1976.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

COMPUTATIONAL COMPLEXITY OF MULTIPLE RECURSIVE SCHEMATA*

STEVEN S. MUCHNICK†

Abstract. The computational complexity properties of a hierarchy of classes of subrecursive schemata are investigated. The schemata are derived from the multiple recursive operators of Péter. Concrete complexity measures based on specific computation rules and an underlying random-access stored-program machine (RASP) model are defined and the complexity properties induced by certain structural features are studied.

It is shown to be undecidable whether two schemata have identical complexity. Upper bounds for the complexity of schemata are then given in terms of a hierarchy of multiple recursive function classes and lower bounds are given which demonstrate that multiple recursion is a thoroughly unfeasible computational tool in practice. Specifically, it is shown that a pure multiple recursive schema capable of defining nonprimitive recursive functions must have at least exponential complexity in terms of its arguments for all interpretations. We also show it undecidable whether a schema has the minimal complexity of the class to which it belongs.

Finally, we raise a number of open questions arising from this work.

Key words. computational complexity, program schema, multiple recursion, program structure, computation rule, unsolvable problems, recursive functions, recursive operators

1. Introduction.

The study of the relationship between the structural and quantitative aspects of algorithms has been one of the prime concerns of the theory of computation since its inception because it holds such great promise for deepening our insight into the nature of computation. Efforts in this area have converged from at least three distinct viewpoints. Recursive function theory has contributed the study of subrecursive hierarchies, computer software the study of programming languages and the schematic languages derived from them, and computer hardware the study of concrete and abstract machines leading to the theory of computational complexity. The quantitative work has been drawn together in recent years by the development of computational complexity theory and even more recently structural aspects of complexity theory have been emphasized in the efforts to delimit the class of natural computational complexity measures and to extend complexity theory to operators and schemata.

This paper contains a contribution to this field, namely an investigation of the computational complexity properties of a hierarchy of classes of subrecursive schemata. Concrete complexity measures based on specific computation rules and an underlying RASP machine model are defined and the complexity properties induced by certain structural features are studied. Along with some basic undecidability results, we give upper bounds for the complexity of schemata in terms of the Péter hierarchy of multiple recursive functions and lower bounds which demonstrate that multiple recursion is a thoroughly unfeasible computational tool in practice.

---

* Received by the editors February 28, 1975, and in revised form February 21, 1976.

† Department of Computer Science, University of Kansas, Lawrence, Kansas 66045. The research reported here was supported in part by the National Science Foundation under Grant GJ-579 and University of Kansas General Research Grant 3758-5038.

From the study of the quantitative aspects of computation on real and abstract machines (in particular the time and memory space used) has come the basis for the abstract theory of computational complexity. As axiomatized by Blum [2], it has produced many extremely interesting results, but has so far contributed relatively little to our understanding of real computation. Among the interesting results we might cite is the phenomenon of almost everywhere speed-up, i.e., that for any complexity measure there exist functions with no almost everywhere best algorithms. Unfortunately, Schnorr in a recent paper has shown that speed-up is irrelevant to practical programming because if a function has sufficiently large almost everywhere speed-up, then either the size of the faster program or the number of points at which speed-up does not apply is enormous (i.e., non-recursive) in the faster program compared to the original.

Thus complexity theory has been only partially successful in its application to real computing, in part because the class of measures admitted by the Blum axioms is too wide. The development of complexity theory for schemata can be expected to provide at least a partial solution for this problem in that it will emphasize structural, rather than arithmetic features of computation. Our study of multiple recursive schemata is a step in this direction.

In § 2 we introduce the multiple recursive schemata **MR** and show that they compute total recursive operators. In § 3 we discuss function classes defined by the multiple recursive schemata and relate them to the Péter hierarchy and transfinite ordinal recursion. In § 4 we present a concrete approach to schemata complexity measures based on a discussion of computation rules and the RASP machine model.

In § 5 we define complexity measures on **MR** based on the discussion in § 4, and discuss a particular measure called the minimal measure. In § 6 we show that it is recursively undecidable whether two schemata have the same complexity and then discuss upper bounds on complexity. In § 7 we give lower bound results for the complexity of **MR** schemata based on their syntax and consider further undecidability questions. In § 8 we discuss the techniques underlying the results of § 7 and some of the issues arising from the present approach to schemata complexity.

We conclude this section with an index of the notation we shall use throughout the remainder of the paper.

Index of notation.

c_R	cost of recursion;
$\mathscr{E}^n$	nth class of the Grzegorczyk hierarchy;
$\mathscr{E}^n_m$	class of $\mathscr{E}^n$ functions limited by $\mathscr{E}^m$ functions;
$\gamma_n(\cdot)$	bounding function of $\mathscr{E}^n$;
M_n	$:= \bigcup_{k=0}^{\infty} \mathscr{E}^k_n$;
M^n	$:=[\lambda x[x+1], \lambda xy[x], \lambda xy[y], \lambda xy[\gamma_n(x, y)];$ Os, limited multiple recursion];
MR	class of multiple recursive schemata;
$\mathbf{MR}_k$	class of k-recursive schemata;
$\mathbf{MR}_{k,m}$	class of k-recursive schemata with m parameters;
N	the nonnegative integers;

Os	operations of substitution;
$\pi_i(\cdot)$	fixed function of an **MR** schema;
R^1	operation of primitive recursion;
$R[\Phi]$	class of functions generable from Φ by R, R^1, and Os operations;
$\mathscr{R}$	class of total recursive functions;
$\mathscr{R}^1$	class of primitive recursive functions;
$\mathscr{R}^n$	class of n-recursive functions;
$\mathscr{R}^n[\Phi]$	class of functions n-recursive in Φ;
$\mathscr{R}_n$	class of n-ary total recursive functions;
$T(r)$	complexity of the term r in the measure T;
TR	complexity of the schema R in the measure T;
T_0R	complexity of the schema R in the minimal measure T_0;
$\mathscr{T},\mathscr{T}^{(i)}$	substitution functionals;
Ω	the undefined value;
$:=$	is defined to be;
$'$	the successor function ($x':=x+1$);
$<_k$	lexicographic ordering on $\mathbf{N}^k$;
$\subseteq$	set-theoretic inclusion;
$\subset$	set-theoretic proper inclusion.

2. The multiple recursive schemata. Constable and Muchnick [9] have described a number of varieties of subrecursive schemata languages based on the LOOP (programming) language of Meyer and Ritchie [22]. These LOOP schemata are of interest because of their particularly simple structure and because some of the varieties have decidable equivalence problems while others have undecidable equivalence problems, even though they all compute the primitive recursive functions $\mathscr{R}^1$ when supplied with a rather trivial interpretation.

In a similar vein, we shall here be concerned with a class of subrecursive schemata, namely the multiple recursive schemata. These differ slightly in origin from most classes of schemata in that they are derived from the theory of recursive functions, rather than from programming. But they are a particularly nice model to utilize in initiating the study of the computational complexity of schemata because they are such highly structured objects—so highly structured that, in some cases, we can read lower bounds on their complexity directly from their structural properties.

The multiple recursive schemata were introduced by Péter [26], but the most systematic treatment of their properties is that of Lachlan. Following Lachlan, we build a definition of multiple recursive schemata from definitions of term and substitution functional. We use lower case Greek letters, with or without affixes, as symbols for functions and f, g, h, with or without subscripts, for function variables. We use x, y, with or without affixes, as variables ranging over the natural numbers $\mathbf{N}$ and i, j, k, m, n, p, q, u, v, w and the corresponding upper case letters, with or without subscripts, as symbols for members of $\mathbf{N}$. We use $z_1, z_2, z_3, \cdots$ as formal variables, whose use will be explained below.

We define a *term* as any formula constructed from function symbols, function variables, natural number variables, formal variables, parentheses, and commas according to the usual composition rules. We could provide a formal syntactic

definition of a term via Backus–Naur form, but this is surely unnecessary, as the reader must have a clear idea what is intended by the informal definition. We use r, s, with or without subscripts, as symbols for terms.

We next define substitution functionals and multiple recursive schemata.

DEFINITION 2.1. A *substitution functional* $\mathcal{T}$ is any finite sequence of ordered triples of natural numbers, say

$$\mathcal{T} = \{\langle n_{i1}, n_{i2}, n_{i3}\rangle | 1 \leqq i \leqq m\}.$$

Let $r_1, r_2, \cdots, r_p$ be any finite sequence of terms. Then an *extension* $r_{p+1}, \cdots, r_{p+m}$ *of the sequence* $r_1, \cdots, r_p$ is given by

$$r_{p+j} = \begin{cases} \text{the term obtained by substituting } r_{n_{i1}} \text{ for } z_{n_{i2}} \text{ in} \\ \qquad\qquad r_{n_{i3}}, \text{ provided that } n_{i1}, n_{i3} < p+j, \\ r_1 \quad \text{otherwise.} \end{cases}$$

The last term r_{p+m} of the extended sequence depends only on $\mathcal{T}$ and $r_1, \cdots, r_p$ and will be denoted by $\mathcal{T}[r_1, \cdots, r_p]$.

DEFINITION 2.2. A *k-recursive schema with m parameters* R or (k, m)-*schema* is a set of $k+1$ equations of the form

$$\begin{aligned} &f(x'_1, \cdots, x'_i, 0, x_{i+2}, \cdots, x_k, y_1, \cdots, y_m) \\ &\quad := \mathcal{T}^{(i)}[x_1, \cdots, x_i, x_{i+2}, \cdots, x_k, y_1, \cdots, y_m, \\ &\qquad f(x_1, z_1, \cdots, z_{k+m-1}), \\ &\qquad f(x'_1, x_2, z_1, \cdots, z_{k+m-2}), \cdots, \\ &\qquad f(x'_1, \cdots, x'_{i-1}, x_i, z_1, \cdots, z_{k+m-i}), \\ &\qquad g_1(z_1, \cdots, z_{u_1}), \cdots, g_p(z_1, \cdots, z_{u_p}), \\ &\qquad \pi_1(z_1, \cdots, z_{v_1}), \cdots, \pi_q(z_1, \cdots, z_{v_q})] \end{aligned} \tag{2.1}$$

for $i = 0, 1, 2, \cdots, k$, where x' denotes the successor of x.

When $i = 0$, the left-hand side is to be interpreted as

$$f(0, x_2, \cdots, x_k, y_1, \cdots, y_m),$$

and when $i = k$ as

$$f(x'_1, \cdots, x'_k, y_1, \cdots, y_m).$$

For $1 \leqq i \leqq q$, $\pi_i(\cdot)$ must be a primitive recursive function; the schema R is fully specified when $\mathcal{T}^{(0)}, \cdots, \mathcal{T}^{(k)}$ and the functions $\pi_1(\cdot), \cdots, \pi_q(\cdot)$ (called the *fixed functions* of R) are given. The $\pi_i(\cdot)$ will frequently by 0-ary functions, i.e., natural number constants.

As an example of a multiple recursive schema, we offer the following

(3, 2)-schema:

$$f(0, x_2, x_3, y_1, y_2) := g_1(x_2, x_3, y_1, y_2),$$
$$f(x_1', 0, x_3, y_1, y_2) := g_2(x_1, f(x_1, x_1, x_3, \pi_1, \pi_2(y_1))),$$
$$f(x_1', x_2', 0, y_1, y_2) := f(x_1', x_2, f(x_1, x_2, x_2, y_1, y_2), y_1, y_2),$$
$$f(x_1', x_2', x_3', y_1, y_2) := f(x_1', x_2', x_3, y_1, x_3),$$

A (k, m)-schema defines an operator mapping the set of p-tuples of total recursive functions (the ith function being an element of $\mathcal{R}_{N_i}$) into the set of $(k+m)$-ary total recursive functions $\mathcal{R}_{k+m}$. We shall use the letters R, S, with or without affixes, to denote (k, m)-schemata. For $m \geqq 0$, the (k, m)-schemata will be referred to as *k-recursive schemata* or *k-schemata* and the k-schemata, for $k \geqq 1$, will be called *multiple recursive schemata* or just *schemata*. We denote the set of all multiple recursive schemata by **MR**, the set of k-schemata by $\mathbf{MR}_k$, and the set of (k, m)-schemata by $\mathbf{MR}_{k,m}$. Thus

$$\mathbf{MR}_k = \bigcup_{m=0}^{\infty} \mathbf{MR}_{k,m}$$

and

$$\mathbf{MR} = \bigcup_{k=1}^{\infty} \mathbf{MR}_k.$$

THEOREM 2.1. *If $\phi_1(\cdot), \cdots, \phi_p(\cdot)$ are functions satisfying $\phi_i(\cdot) \in \mathcal{R}_{N_i}$, then there is a unique function $\phi(\cdot)$ such that the equations obtained from* (2.1) *by substituting ϕ for f and $\phi_1, \cdots, \phi_p$ for $g_1, \cdots, g_p$, respectively, are valid.*

Proof. Let the symbol $<_k$ denote the well-ordering of $\mathbf{N}^k$ given by

$$\langle m_1, \cdots, m_k\rangle <_k \langle n_1, \cdots, n_k\rangle \quad \text{iff} \quad \exists x(1 \leqq x \leqq k \ \& \ m_x < n_x \ \& \ \forall y(1 \leqq y < x \Rightarrow m_y = n_y)),$$

i.e., the usual lexicographic ordering. The proof is by *$<_k$-induction*, that is induction over all k-tuples of natural numbers ordered by $<_k$.

The least k-tuple in this ordering is $\langle 0, \cdots, 0\rangle$ and the definition of $f(0, \cdots, 0, y_1, \cdots, y_m)$ in (2.1) can easily be seen not to contain any occurrences of f on the right-hand side, so that $\phi(0, \cdots, 0, y_1, \cdots, y_m)$ is uniquely determined by $\phi_1(\cdot), \cdots, \phi_p(\cdot)$.

Given $n_1, \cdots, n_k$ satisfying $\langle 0, \cdots, 0\rangle <_k \langle n_1, \cdots, n_k\rangle$, suppose $\phi(\bar{n}_1, \cdots, \bar{n}_k, \bar{q}_1, \cdots, \bar{q}_m)$ is uniquely defined by (2.1) for all $\bar{n}_1, \cdots, \bar{n}_k$ such that $\langle \bar{n}_1, \cdots, \bar{n}_k\rangle <_k \langle n_1, \cdots, n_k\rangle$ and all $\bar{q}_1, \cdots, \bar{q}_m$. Then $\phi(n_1, \cdots n_k, q_1, \cdots, q_m)$ is defined by (2.1) as a term containing only instances of $\phi(\bar{n}_1, \cdots, \bar{n}_k, \bar{q}_1, \cdots, \bar{q}_m)$ such that $\langle \bar{n}_1, \cdots, \bar{n}_k\rangle <_k \langle n_1, \cdots, n_k\rangle$. Thus $\phi(n_1, \cdots, n_k, q_1, \cdots q_m)$ is also uniquely defined by (2.1). □

3. Function classes based on the multiple recursive schemata. If $\Phi \subseteq \mathcal{R}$ and R is an **MR** schema, then $R[\Phi]$ denotes the set of functions generable from the functions in Φ by R, the primitive recursion operator R^1, and the operations of

substitution Os (see, for example, Constable [5])

$$R[\Phi]:=[\Phi; R, R^1, \mathrm{Os}].$$

The set of functions k-recursive in Φ, denoted $\mathscr{R}^k[\Phi]$, is

$$\mathscr{R}^k[\Phi]:= \bigcup_{R\in \mathbf{MR}_k} R[\Phi].$$

Péter [26] proved that for any finite set of functions Φ, $\mathscr{R}^k[\Phi]\subset \mathscr{R}^{k+1}[\Phi]\subset \mathscr{R}$ and that there exists a k-schema R_k such that $\mathscr{R}^k[\Phi]=R_k[\Phi]$ for any Φ. Lachlan calls a k-schema with this property *k-adequate*, gives a number of k-adequate schemata for all k, and studies the properties of k-adequate schemata in general. He proves that if $k\geqq 2$ and R is k-adequate, then for $1\leqq j\leqq k-2$ there exists on the right-hand side of one of the defining equations of R an occurrence of a term of the form

$$f(x'_1, \cdots, x'_{j-1}, x_j, r, \cdots),$$

where r contains either an occurrence of a parameter y_i or one of the recursion variables $x_{j+2}, \cdots, x_k$. This makes it possible for him to prove that some of the k-adequate schemata he discusses are incapable of further simplification for a rather natural notion of simplification which he describes (see Lachlan [16, p. 106]).

Let $\mathscr{R}^k$ denote $\mathscr{R}^k[\lambda x[0], \lambda x[x+1], \lambda xy[x], \lambda xy[y]]$ (by Lachlan's [16] Lemma 2, $\mathscr{R}^1$ coincides with the primitive recursive functions). The sequence $\{\mathscr{R}^k\}$ is known as the *Péter hierarchy* of multiple recursive functions. Péter [26] proved that

$$\mathscr{R}^1\subset \mathscr{R}^2\subset \mathscr{R}^3\cdots\subset \mathscr{R}$$

and that $\mathscr{R}^n$ is exactly the class $\mathscr{F}_n$ proposed by Hilbert in his celebrated but unsuccessful program to solve the continuum problem in *Über das Unendliche.* Ackermann introduced the idea of transfinite ordinal recursion in 1940 and Péter [27] was able to show that $\mathscr{R}^n$ is precisely the ω^n-ordinal recursive functions for certain standard well-orderings of order type ω^n, such as the $<_n$ defined in the preceding section. Robbin constructs a number of transfinite hierarchies based on ordinal recursion, all of which coincide with the Péter hierarchy.

An excellent and highly readable recent survey of the above and other ordinal hierarchies is to be found in Chapter 1 of Moll's doctoral thesis.

In another direction of research, Marčenkov considers a class of multiple recursive schemata which are a subclass of **MR** and defines from them what he calls *k-fold limited recursion* by annexing to the equations of (2.1) the inequality

$$f(x_1, \cdots, x_k, y_1, \cdots, y_m)\leqq h(x_1, \cdots, x_k, y_1, \cdots, y_m). \tag{3.1}$$

This is, of course, a generalization of the idea of limited primitive recursion to multiple recursions. He then defines M_n to be

$$M_n:= \bigcup_{k=0}^{\infty} \mathscr{E}^k_n,$$

where $\mathscr{E}_n^k$ is the set of all $\mathscr{E}^k$ functions[1] limited by $\mathscr{E}^n$ functions and M^n to be

$$M^n := [\lambda x[x+1], \lambda xy[x], \lambda xy[y], \lambda xy[\gamma_n(x, y)]; \text{Os, limited multiple recursion}]$$

(where $\gamma_n(\cdot)$ is a bounding function of $\mathscr{E}^n$). His major result is that for $n \geqq 2$, $M_n = M^n$. His proof of his Theorem 1 [19, p. 54], which he uses to establish that $M_n \subseteq M^n$, is seriously flawed (the function $\phi(\cdot)$ which he defines in the proof is seen on careful inspection to be a constant function), but this gap seems to be bridgeable by a computational complexity argument. He claims (without proof) that $M_n = M^n$ holds for $n = 0$ and 1 as well.

4. Computation rules. To develop a theory of computational complexity for multiple recursive schemata we can either take the highly abstract approach of generalizing the axiomatic development of Blum [2] of complexity theory for recursive functions or the more concrete one of specifying one or more semantic models for multiple recursive schemata and then measuring the computational resource requirements they induce.

Valuable as the abstract approaches of Symes, Lynch, Constable, and Weihrauch are, we reject them for the present work for three reasons. The first is that one of our main objectives is to demonstrate that multiple recursion is a thoroughly unfeasible computational tool in practice. As the ordinary theory of computational complexity for functions shows quite amply, the axiomatic approach admits measures with severely pathological properties, so that we would, in any case, have to restrict our attention to some class of natural measures to obtain such results if we took the axiomatic approach. Our second objective (and concomitantly our second objection) is to begin to assemble a list of properties which make a measure natural, such as those suggested by Hartmanis [13] for the ordinary theory. We believe this can better be accomplished by considering natural measures derived from specific well-understood computation rules and then abstracting their properties than by searching the abstract theory for patches of pathology which we then eliminate. This approach gives us better insight into the properties which constitute naturalness. Finally, multiple recursive schemata are highly structured objects. Surely it is to our advantage to take their structure into account in developing the theory, for it is reasonable to think that it is exactly their structure which determines their natural complexity. It is thus desirable to insure that our measures reflect the structural properties of the underlying objects, which surely not all measures will do.

We shall adopt a more concrete approach—to select a model of computation and then to consider possible computation rules, which transform multiple recursive schemata into algorithms with both function and number inputs, which indicate how the corresponding operators are to be computed. The computation model and rules will then be seen to induce the properties of the class of measures we shall associate with them.

The model of computation which seems most suitable to this work, because of its close correspondence to real computers, is the random-access stored-program machine or RASP of Elgot and Robinson. Hartmanis [12] defines a particular RASP **M1** whose instructions are essentially those of a simple one-address

[1] $\mathscr{E}^k$ is the kth class of the Grzegorczyk hierarchy.

computer with indirect addressing—specifically he allows conditional and unconditional transfer, load, store, add, subtract, and halt instructions. We shall use essentially this RASP as the basis for our concrete complexity measures, but shall not define it precisely here, referring the reader instead to Hartmanis [12] for the details.

It is of interest to note that because we are using a RASP, rather than a Turing machine model, we shall be unconcerned about the length of the numbers we work with. Only their values will be significant. Thus when we obtain polynomial or exponential lower bounds for complexity in § 7 they will be in terms of the values of the inputs. If the reader prefers to think in terms of the Turing machine model, he should translate these into exponential and double exponential bounds, respectively, in terms of the length of the inputs.

We now turn to the subject of computation rules, which may be defined as semantic mappings from schemata to machine models. One issue in selecting a computation rule is whether to allow it memory or not, that is, whether it can remember a value of a function computed in an earlier step of an evaluation so as to avoid recomputing the same value at a later step of the same evaluation. This property is obviously advantageous in evaluating a multiple recursion such as

$$f(0) := \pi_1,$$
$$f(x_1') := g_1(f(x_1), f(x_1)+1),$$

which requires exponentially many evalations of $f(\cdot)$ without memory, but only linearly many with memory (see Fig. 4.1). On the other hand, for most schemata memory is useless but expensive overhead, since no value or very few values will ever be repeated in any given computation. This fact and the fact that allowing memory would embroil us in a discussion of efficient data structures to implement it cause us to discard it as a possible attribute of the computation rules we shall consider.

Another issue is the possibility of sidestepping part of the computational process by proving that the operator or function being defined is in some sense capable of computation by a simpler or less expensive means. We shall discard this as a component in defining computation rules for reasons given below, but keep in mind that this is the essence of the phenomenon of speed-up. This kind of technique can be a highly profitable one for multiple recursions. For example, the 2-recursive schema

$$\begin{aligned} &f(0, x_2) := 2, \\ &f(x_1', 0) := f(x_1, 1), \\ &f(x_1', x_2) := f(x_1, f(x_1', x_2)) \end{aligned} \tag{4.1}$$

defines the same function as the much simpler 2-recursive schema

$$\begin{aligned} &f(0, x_2) := 2, \\ &f(x_1', 0) := 2, \\ &f(x_1', x_2') := 2 \end{aligned} \tag{4.2}$$

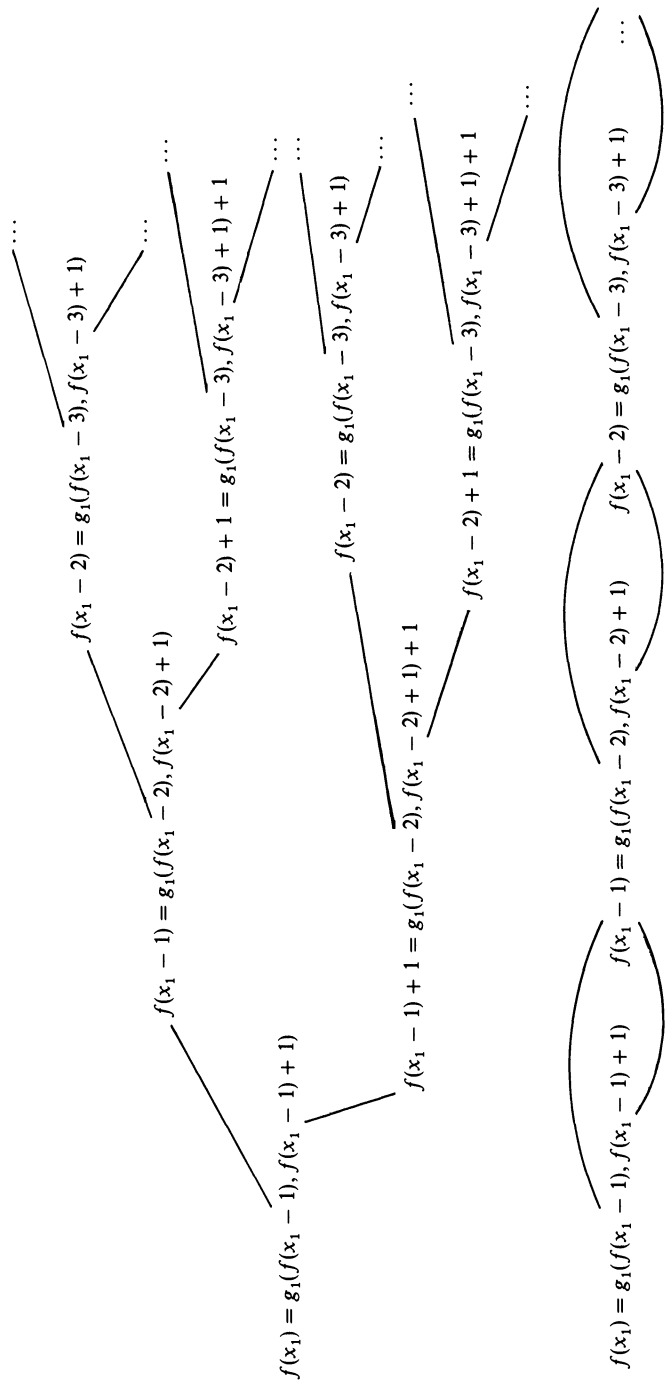


FIG. 4.1. *Computation of $f(x_1)$ with and without memory for the multiple recursion $f(0) := \pi_1$, $f(x_1') := g_1(f(x_1), f(x_1) + 1)$*

and the non-recursive schema

$$f(x_1, x_2) := 2. \tag{4.3}$$

In any natural measure, (4.2) and (4.3) must have lower complexity than (4.1). For the class of measures on which we shall concentrate our attention, the complexity of (4.1) is given by an exponential function, while those of (4.2) and (4.3) are constants. On the other hand, some k-recursions must define functions in $\mathscr{R}^k[\Phi]-\mathscr{R}^{k-1}[\Phi]$ and hence require a computing time which is in $\mathscr{R}^k[\Phi]-\mathscr{R}^{k-1}[\Phi]$ on a RASP, since otherwise they would be in $\mathscr{R}^{k-1}[\Phi]$. For such schemata this technique is a waste of time since ultimately the order of magnitude of their computation time is fixed by their nature. Such a function can be exhibited by changing the initial fixed function in (4.1) to x_2+1, to produce

$$\begin{aligned} &f(0, x_2) := x_2+1, \\ &f(x_1', 0) := f(x_1, 1), \\ &f(x_1', x_2') := f(x_1, f(x_1', x_2)), \end{aligned} \tag{4.4}$$

which defines Ackermann's function, the first known example of a function in $\mathscr{R}^2-\mathscr{R}^1$. The properties of computation rules which include these and other types of simplifications are discussed at length by Cadiou [3].

We now describe four computation rules which use neither of the special techniques just described and which are simple in the sense that at each stage of the computation they only tell us which of the occurrences of f (the function being defined) to replace with its (their) definition next. The four rules are:

1. *Kleene's rule.* Replace all occurrences of f with their definitions simultaneously.

2. *Leftmost-innermost rule.* Replace only the leftmost-innermost occurrence of f (i.e., the occurrence of f furthest to the left whose arguments contain no occurrences of f) by its definition.

3. *Leftmost-outermost rule.* Replace only the leftmost-outermost occurrence of f with its definition.

4. *Normal (or delay) rule* (see Manna, Ness and Vuillemin, [18]. Replace one occurrence of f chosen as follows: try to replace the leftmost-outermost occurrence of f if its arguments are sufficiently evaluated that it can be replaced; if this fails, next try to replace the first occurrence of f which prevented replacement of the one which just failed; repeat the preceding step until one f has been replaced with its definition.

Since these computation rules are discussed extensively elsewhere, we shall simply note that they are all appropriate for the task at hand, and that we have chosen to begin our study of this area with consideration of the leftmost-innermost rule.

5. Concrete complexity measures. We next define a class of computational complexity measures on **MR** based on the leftmost-innermost computation rule. As in our definition of a multiple recursive schema, we first define the complexity of a term and then proceed to schemata.

DEFINITION 5.1. Let $t\pi_1(\cdot), \cdots, t\pi_q(\cdot) \in R^1$, $tg_1(\cdot), \cdots, tg_p(\cdot)$ be variables defined on $\mathscr{R}$ and such that $\pi_i(\cdot)$ and $t\pi_i(\cdot)$, and $tg_j(\cdot)$ and $g_j(\cdot)$ have the same arity and let $c_R \in \mathbf{N} - \{0\}$. Then the *complexity* $T(r)$ *of a term* r is defined to be

1. if r is x_i, x_i', or y_i, then $T(r) := 0$,
2. if $r = \pi_i(z_1, \cdots, z_{v_i})$, then

$$T(r) := t\pi_i(z_1, \cdots, z_{v_i}) + \sum_{j=1}^{v_i} T(z_j),$$

3. if $r = g_i(z_1, \cdots, z_{u_i})$, then

$$T(r) := tg_i(z_1, \cdots, z_{u_i}) + \sum_{j=1}^{u_i} T(z_j),$$

4. if $r = f(z_1, \cdots, z_{k+m})$, then

$$T(r) := tf(z_1, \cdots, z_{k+m}) + c_R + \sum_{j=1}^{k+m} T(z_j).$$

We view $T(\cdot)$ as a syntactic transformation which, given a string of characters representing a term, repeatedly applies rules 1–4 until it obtains a string containing no occurrences of $T(\cdot)$. For example, given

$$r = f(x_1, g_2(x_2, \pi_2), f(x_1', x_2, g_1(y_1))),$$

we compute $T(r)$ by the following:

$$\begin{aligned}
T(r) &= tf(x_1, g_2(x_2, \pi_2), f(x_1', x_2, g_1(y_1))) + c_R + T(x_1) \\
&\quad + T(g_2(x_2, \pi_2)) + T(f(x_1', x_2, g_1(y_1))) \\
&= tf(x_1, g_2(x_2, \pi_2), f(x_1', x_2, g_1(y_1))) + c_R + tg_2(x_2, \pi_2) \\
&\quad + T(x_2) + T(\pi_2) + tf(x_1', x_2, g_1(y_1)) + T(x_1') + T(x_2) \\
&\quad + T(g_1(y_1)) \\
&= tf(x_1, g_2(x_2, \pi_2), f(x_1', x_2, g_1(y_1))) + c_R + tg_2(x_2, \pi_2) \\
&\quad + t\pi_2 + tf(x_1', x_2, g_1(y_1)) + tg_1(y_1) + T(y_1) \\
&= tf(x_1, g_2(x_2, \pi_2), f(x_1', x_2, g_1(y_1))) + c_R + tg_2(x_2, \pi_2) \\
&\quad + t\pi_2 + tf(x_1', x_2, g_1(y_1)) + tg_1(y_1).
\end{aligned}$$

The intuitive grounding of this definition is as follows. Since x_i, x_i', and y_i are given, we let their cost of computation be zero (in a Turing machine model we might want to fix their costs as their base 2 logarithms (i.e. their lengths), but in a RASP model we consider this irrelevant). Since the $\pi_i(\cdot)$ are fixed $\mathscr{R}^1$ functions, we fix their costs of computation as $\mathscr{R}^1$ functions plus the cost of computing their arguments. Since the $g_i(\cdot)$ may be the results of multiple recursions themselves, we allow their costs of computation to be given by functions $tg_i(\cdot)$ corresponding to them plus the cost of computing their arguments. Finally, $f(\cdot)$, the function being defined, is being defined by a process of repeated substitutions. Hence its cost of computation is the cost of the substitution process c_R (for cost of recursion)

plus the cost of what is being substituted, which is given by the function $tf(\cdot)$, plus the cost of computing its arguments.

We then define the complexity of an **MR** schema as follows:

DEFINITION 5.2. Let R be the schema defined in Definition 2.2 and $t\pi_1(\cdot), \cdots, t\pi_q(\cdot)$, c_R, $tg_1(\cdot), \cdots, tg_p(\cdot)$ be as in Definition 5.1. Then the *computational complexity* of R is the schema TR given by

$$
\begin{aligned}
&tf(x_1', \cdots, x_i', 0, x_{i+2}, \cdots, x_k, y_1, \cdots, y_m) \\
&\quad := T(\mathscr{T}^{(i)}[x_1, \cdots, x_i, x_{i+2}, \cdots, x_k, y_1, \cdots, y_m, \\
&\qquad f(x_1, z_1, \cdots, z_{k+m-1}), \\
&\qquad f(x_1', x_2, z_1, \cdots z_{k+m-2}), \cdots, \\
&\qquad f(x_1', \cdots x_{i-1}', x_i, z_1, \cdots, z_{k+m-i}), \\
&\qquad g_1(z_1, \cdots, z_{u_i}), \cdots, g_p(z_1, \cdots, z_{u_p}), \\
&\qquad \pi_1(z_1, \cdots, z_{v_1}), \cdots, \pi_q(z_1, \cdots, z_{v_q})]) \\
&\qquad\qquad \text{for } i = 0, 1, \cdots, k,
\end{aligned} \tag{5.1}
$$

where T is given by Definition 5.1. Thus a complexity measure on **MR** is a mapping $T: \mathbf{MR} \to \mathbf{MR}$ which carries $R[\phi_1(\cdot), \cdots, \phi_p(\cdot)](\cdot)$ to $TR[\phi_1(\cdot), \cdots, \phi_p(\cdot), t\phi_1(\cdot), \cdots, t\phi_p(\cdot)](\cdot)$. Different complexity measures are given by different choices of the constant c_R and the functions $t\pi_1(\cdot), \cdots, t\pi_q(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot)$ in Definition 5.1.

As an example of the complexity of a multiple recursive schema, let us consider the schema

$$
\begin{aligned}
&f(0, x_2) := g_1(x_2), \\
&f(x_1', 0) := f(x_1, \pi_1), \\
&f(x_1', x_2') := f(x_1, f(x_1', x_2)),
\end{aligned} \tag{5.2}
$$

which is an abstraction of (4.1) and (4.4). The complexity of (5.2) is given by

$$
\begin{aligned}
&tf(0, x_2) := tg_1(x_2), \\
&tf(x_1', 0) := tf(x_1, \pi_1) + c_R + t\pi_1, \\
&tf(x_1', x_2') := tf(x_1, f(x_1', x_2)) + tf(x_1', x_2) + 2 \cdot c_R.
\end{aligned}
$$

A particular complexity measure which will be of interest to us in the sequel is one which is, in a sense, minimal:

DEFINITION 5.3. Let $t\pi_1(\cdot), \cdots, t\pi_q(\cdot) \equiv 0$, $tg_1(\cdot), \cdots, tg_p(\cdot) \equiv 0$, and $c_R = 1$ in Definition 5.2. The resulting complexity measure is called the *minimal complexity measure* and is denoted T_0. Notice that we need not list $tg_1(\cdot), \cdots, tg_p(\cdot)$ as arguments to $T_0R[\cdot](\cdot)$ since they are assumed identically zero.

For our example (5.2), the T_0 measure is

$$
\begin{aligned}
&t_0f(0, x_2) := 0, \\
&t_0f(x_1, 0) := t_0f(x_1, \pi_1) + 1, \\
&t_0f(x_1', x_2') := t_0f(x_1, f(x_1', x_2)) + t_0f(x_1', x_2) + 2.
\end{aligned}
$$

We now show that T_0 is indeed minimal in the following sense:

THEOREM 5.1. *Let* $R[g_1(\cdot), \cdots, g_p(\cdot)](\cdot)$ *be in* **MR**. *Let* $c_R \in \mathbf{N}$ *and* $t\pi_1(\cdot), \cdots, t\pi_q(\cdot) \in R$ *be given for some complexity measure* T. *Then*

$$TR[g_1(\cdot), \cdots, g_p(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m)$$
$$\geqq c_R \cdot T_0R[g_1(\cdot), \cdots, g_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m).$$

Proof. The proof is in two parts, both by induction. First we verify that the measure T' obtained from T by setting $t\pi_1(\cdot), \cdots, t\pi_q(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot) \equiv 0$ satisfies

$$T'R[g_1(\cdot), \cdots, g_p(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m) \tag{5.3}$$
$$= c_R \cdot T_0R[g_1(\cdot), \cdots, g_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m)$$

and then that

$$TR[g_1(\cdot), \cdots, g_p(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m)$$
$$\geqq T'R[g_1(\cdot), \cdots, g_p(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m).$$

Notice that in the measures T_0 and T' the terms corresponding to $\langle 0, x_2, \cdots, x_k, y_1, \cdots, y_m\rangle$ must be

$$t_0f(0, x_2, \cdots, x_k, y_1, \cdots, y_m) = 0$$

and

$$t'f(0, x_2, \cdots, x_k, y_1, \cdots, y_m) = 0$$

since the right-hand side of (2.1) with $i = 0$ cannot contain any occurrence of f because of the lexicographic ordering on $\mathbf{N}^k$. Hence (5.3) is valid if $x_1 = 0$. Suppose (5.3) is valid for all $\langle \bar{x}_1, \cdots, \bar{x}_k, \bar{y}_1, \cdots, \bar{y}_m\rangle$ such that

$$\langle \bar{x}_1, \cdots, \bar{x}_k\rangle <_k \langle x_1, \cdots, x_k\rangle.$$

Then $T'R[g_1(\cdot), \cdots, g_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m)$ is given by $t'f(x_1, \cdots, x_k, y_1, \cdots, y_m)$ which is a sum of n terms of the form $t'f(\bar{x}_1, \cdots, \bar{x}_k, \bar{y}_1, \cdots, \bar{y}_m)$ satisfying $\langle \bar{x}_1, \cdots, \bar{x}_k\rangle <_k \langle x_1, \cdots, x_k\rangle$ and a term of the form $n \cdot c_R$ for some $n \in \mathbf{N}$. Similarly $T_0R[(g_1(\cdot), \cdots, g_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m)$ is given by $t_0f(x_1, \cdots, x_k, y_1, \cdots, y_m)$ which is a sum of n corresponding terms of $t_0f(\bar{x}_1, \cdots, \bar{x}_k, \bar{y}_1, \cdots, \bar{y}_m)$ and the constant n. Thus

$$T'R[g_1(\cdot), \cdots, g_p(\cdot), tg_1(\cdot), \cdots, tg_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m)$$
$$= t'f(x_1, \cdots, x_k, y_1, \cdots, y_m)$$
$$= \sum_{i=1}^{n} (t'f(\bar{x}_{1i}, \cdots, \bar{x}_{ki}, \bar{y}_{1i}, \cdots, \bar{y}_{mi}) + n \cdot c_R)^s$$
$$= c_R \cdot \Big(\sum_{i=1}^{n} t_0f(\bar{x}_{1i}, \cdots, \bar{x}_{ki}, \bar{y}_{1i}, \cdots, \bar{y}_{mi}) + n\Big)$$
$$= c_R \cdot t_0f(x_1, \cdots, x_k, y_1, \cdots, y_m)$$
$$= c_R \cdot T_0R[g_1(\cdot), \cdots, g_p(\cdot)](x_1, \cdots, x_k, y_1, \cdots, y_m).$$

It should be obvious that the second part of the proof can be carried out by a similar induction, and hence we leave it to the interested reader. □

T_0 will be useful because any lower bounds we can obtain for it will automatically be extendable to all measures by Theorem 5.1. We view it as isolating the cost of the substitution process in recursion.

6. Properties of the complexity measures. In this section we show that it is recursively undecidable whether two schemata have the same complexity and then discuss upper bounds for their complexity.

Constable and Muchnick describe a technique for showing that the equivalence problem for LOOP programs is recursively unsolvable [9] and adapt a technique of Luckham, Park, and Paterson to show equivalence undecidable for several formulations of LOOP schemata [17]. We use essentially the former technique here to show that it is undecidable whether two **MR** schemata have identical complexity.

THEOREM 6.1. *It is recursively undecidable whether two* **MR** *schemata have identical complexity.*

Proof. Let $\phi_1, \phi_2, \phi_3, \cdots$ enumerate all one-tape, one-input Turing machines with alphabet $\{0, 1\}$. It is well known (see, for example, Constable and Muchnick [9]) that there is a primitive recursive function $\pi_1 : \mathbf{N}^2 \to \mathbf{N}$ such that

$$\pi_1(i, y) = \begin{cases} 1 & \text{if } i \geqq 1 \text{ and } \phi_i(i) \text{ halts in} \leqq y \text{ steps,} \\ 0 & \text{otherwise.} \end{cases}$$

It is recursively undecidable whether $\lambda y[\pi_1(i, y)]$ and $\lambda y[\pi_1(0, y)]$ are the same function, since otherwise the halting problem on index would be decidable.

Let R_i denote the **MR** schema

$$\begin{aligned} &f(0, x_2, y_1) := y_1, \\ &f(x_1', 0, y_1) := y_1, \\ &f(x_1', x_2', y_1) := f(x_1, \pi_1(i, y_1), y_1). \end{aligned}$$

The complexity of R_i is given by

$$\begin{aligned} &tf(0, x_2, y_1) := 0, \\ &tf(x_1', 0, y_1) := 0, \\ &tf(x_1', x_2', y_1) := tf(x_1, \pi_1(i, y_1), y_1) + c_R + t\pi_1(i, y_1), \end{aligned}$$

where we select the function $t\pi_1(\cdot)$ so as to satisfy $t\pi_1(i, y_1) = t\pi_1(j, y_1)$ for all i, j, y_1. Inspecting this formula we discover that

$$tf(x_1, x_2, y_1) = \begin{cases} 0 & \text{if } x_1 = 0 \text{ and } x_2 = 0, \\ c_R + t\pi_1(i, y_1) & \text{if } x_1, x_2 > 0 \text{ and } \pi_1(i, y_1) = 0, \\ x_1 \cdot (c_R + t\pi_1(i, y_1)) & \text{if } x_1, x_2 > 0 \text{ and } \pi_1(i, y_1) = 1. \end{cases}$$

Thus R_0 and R_i have identical complexity iff $\pi_1(i, y_1) = 0$ for all y_1, i.e., iff $\phi_i(i)$ diverges, and so the question is undecidable. □

In Muchnick and Constable [24] we also consider another model for multiple recursive schemata, the vector schemata VR°, and show that essentially comparable results can be obtained for them. They are of interest because they provide the capability to code Turing machine computations (or other bases for undecidability) in their structural, rather than arithmetical, properties.

Though we cannot in general determine whether two multiple recursive schemata have the same complexity, even in the trivial case of those with no function inputs as shown in Theorem 6.1, we shall see that it is possible to bound the complexity of a schema both above and below. At the function level, upper bounds depend heavily on the choice of the $tg_i(\cdot)$'s. For example, the trivial schema

$$\begin{aligned} f(0) &:= g_1(0), \\ f(x_1') &:= g_1(x_1') \end{aligned} \tag{6.1}$$

has complexity

$$tf(x_1) := tg_1(x_1)$$

and thus $t\phi(\cdot)$ for a function $\phi(\cdot)$ defined by (6.1) can be chosen as an arbitrary 1-ary function in $\mathscr{R}$. On the other hand it should be clear that the following lemma holds:

LEMMA 6.2. *If* $\phi(\cdot) \in \mathscr{R}^k[\phi_1(\cdot), \cdots, \phi_p(\cdot)]$, *then*

$$t\phi(\cdot) \in \mathscr{R}^k[\phi_1(\cdot), \cdots, \phi_p(\cdot), t\phi_1(\cdot), \cdots, t\phi_p(\cdot)].$$

Proof. If $\phi(\cdot) \in \mathscr{R}^k[\phi_1(\cdot), \cdots, \phi_p(\cdot)]$, then there is a k-recursive schema R with p function inputs such that if we substitute $\phi_i(\cdot)$ for $g_i(\cdot)$ in R then $f(\cdot) = \phi(\cdot)$. But Definitions 5.2 and 5.3 construct a k-recursive schema TR from R which defines $tf(\cdot)$ from $g_1(\cdot), \cdots, g_p(\cdot)$, $tg_1(\cdot), \cdots, tg_p(\cdot)$, $\pi_1(\cdot), \cdots, \pi_q(\cdot)$, $t\pi_1(\cdot), \cdots, t\pi_q(\cdot)$. Now $\pi_1(\cdot), \cdots, \pi_q(\cdot)$, $t\pi_1(\cdot), \cdots, t\pi_q(\cdot)$ are all in $\mathscr{R}^1$ and consequently, making the appropriate substitutions, we have

$$t\phi(\cdot) \in \mathscr{R}^k[\phi_1(\cdot), \cdots, \phi_p(\cdot), t\phi_1(\cdot), \cdots, t\phi_p(\cdot)]. \qquad \square$$

The converse of Lemma 6.2 is false in general because of our freedom to select the $t\phi(\cdot)$ arbitrarily in $\mathscr{R}$. Thus, for example, with $k = p = 1$, $\phi_1(x_1) = 0$ identically and $t\phi_1(x_1) = \alpha(x_1, x_1)$, where $\alpha(\cdot)$ is Ackermann's function (4.1), we have $t\phi(\cdot) \in \mathscr{R}^1[\lambda x_1[0], \alpha(\cdot)] = \mathscr{R}^2$. Now we can clearly compute $\alpha(\cdot)$ in a time $t\alpha(\cdot)$ in $\mathscr{R}^1[\lambda x_1[0], \alpha(\cdot)] = \mathscr{R}^2$, but $\alpha(\cdot)$ is certainly not in $\mathscr{R}^1[\lambda x_1[0]] = \mathscr{R}^1$ as the direct converse of Lemma 6.2 would require.

7. Lower bounds on schemata complexity. We come now to the results concerning classification of **MR** schemata on predominantly syntactic grounds, where the classes turn out to correspond to nontrivial lower bounds on the complexity of the schemata. In particular we show that a large class of **MR** schemata require exponentially many invocations of recursion for most function inputs. We shall first present an example of this phenomenon, then an exposition

of the motivation for this work, and then finally the relevant results.

Consider the following 1-recursive schema R:

$$\begin{aligned} f(0, y_1) &:= g_1(y_1), \\ f(x_1', y_1) &:= g_2(f(x_1, f(x_1, y_1))). \end{aligned} \tag{7.1}$$

By Definition 5.2, TR is given by

$$\begin{aligned} tf(0, y_1) &:= tg_1(y_1), \\ tf(x_1', y_1) &:= tg_2(f(x_1, f(x_1, y_1))) + tf(x_1, f(x_1, y_1)) + tf(x_1, y_1) + 2 \cdot c_R \end{aligned}$$

and, in particular, T_0R is given by

$$\begin{aligned} t_0f(0, y_1) &:= 0, \\ t_0f(x_1', y_1) &:= t_0f(x_1, f(x_1, y_1)) + t_0f(x_1, y_1) + 2. \end{aligned}$$

We show by induction that $t_0f(\cdot)$ is an exponential function of x_1 (and independent of y_1). We have $t_0f(0, y_1) = 0$ for all y_1. Suppose that $t_0f(x_1, y_1) = 2^{x_1+1} - 2$ for all y_1. Then

$$\begin{aligned} t_0f(x_1', y_1) &= t_0f(x_1, f(x_1, y_1)) + t_0f(x_1, y_1) + 2 \\ &= 2^{x_1+1} - 2 + 2^{x_1+1} - 2 + 2 \\ &= 2^{x_1+1} - 2. \end{aligned}$$

Thus $t_0f(x_1, y_1) = 2^{x_1+1} - 2$ for all x_1, y_1 and for any measure TR, by Theorem 5.1,

$$tf(x_1, y_1) \geqq (2^{x_1+1} - 2) \cdot c_R,$$

so that (7.1) is seen to require at least exponentially many invocations of recursion regardless of the functions substituted for $g_1(\cdot)$ and $g_2(\cdot)$.

We believe results of this sort are interesting for two kinds of reasons. One is ultimately practical in that we envision the construction of programming systems at some point in the future which will be considerably more intelligent than current compilers and task-oriented systems, such as symbolic algebraic manipulators. Such a system might well include in its capabilities an estimate of the feasability of a given computation or computational method. Indeed, a facility akin to this already exists in symbolic algebraic manipulation systems such as SAC-1 at the University of Wisconsin and MACSYMA at the Massachusetts Institute of Technology, which, in many cases, vary the methods applied to a problem of some specific type according to some measure of the size of the current instance of the problem type. If, for example, such a system has available two methods to solve some problem, one of which requires 2^n time units and the other $1000n^2$ units, where n is some measure of the size of the problem instance, it would choose the first method for $n \leqq 17$ ($2^n < 1000n^2$ for $n \leqq 17$) and the other for $n \geqq 18$ ($2^n > 1000n^2$ for $n \geqq 18$). We view our results as extending this type of discrimination upward so that if a programmer constructs a method to solve a problem which involves multiple recursion, a programming system using our results can tell him whether his method is unfeasible.

Secondly, we view our results as extending upward to schemata and operators the kinds of results obtained in recent years in the theory of algorithms and the inherent complexity of classes of problems. Thus, for example, instead of asserting

that algorithms to solve problems such as equivalence of regular expressions with intersection cannot operate in polynomial space (or time) on a Turing machine, we can state that any method to solve any problem which utilizes multiple recursion within the appropriate restrictions must require exponential time on a RASP.

The class of schemata we wish to consider are the pure schemata defined as follows:

DEFINITION 7.1. A k-recursive schema is *j-pure* for some $j \leqq k$ iff the right-hand side of the jth equation in Definition 2.2 includes no occurrences of $g_i(\cdot)$'s or $\pi_i(\cdot)$'s among the first j arguments of any occurrence of $f(\cdot)$ and if $j > k$, then the $(j+1)$st argument is 0 in each. A k-recursive schema is called *pure* iff it is k-pure.

DEFINITION 7.2. Two or more occurrences of $f(\cdot)$ in an expression are *disjoint* iff none of them appears in an argument to another of them; otherwise they are *nested.*

In other words, a k-recursive schema is j-pure iff the first j arguments of any occurrence of $f(\cdot)$ on the right-hand side of the equation defining

$$f(x_1', \cdots, x_j', 0, x_{i+2}, \cdots, x_k, y_1, \cdots, y_m)$$

are all of the form x_i, x_i', or another occurrence of $f(\cdot)$ and the $(j+1)$st is 0 if $j < k$ and is disjointly j-pure if they are all x_i of x_i'. For example, the 3-recursive schema

$$f(0, x_2, x_3, y_1) := \pi_1(x_2, x_3, y_1),$$

$$f(x_1', 0, x_3, y_1) := g_1(f(x_1, \pi_2, x_3, y_1)),$$

$$f(x_1', x_2', 0, y_1) := f(x_1', x_2, x_2, \pi_3(y_1)),$$

$$f(x_1', x_2', x_3', y_1) := g_2(f(x_1', x_2', x_3, g_3(y_1)), f(x_1', x_2, x_3', \pi_1), g_4(x_1, x_2))$$

is disjointly 3-pure and thus disjointly pure.

We can now obtain the following theorems relating lower bounds on complexity and schematic purity:

THEOREM 7.1. *If the k-schema R is pure and the kth defining equation of R includes only one occurrence of $f(\cdot)$ on the right-hand side, then*

$$t_0 f(x_1', \cdots, x_k', y_1, \cdots, y_m) \geqq \min(x_1, \cdots, x_k) + 1.$$

Proof. Let the one occurrence of $f(\cdot)$ on the right-hand side of the kth defining equation be

$$f(e_1, \cdots, e_{k+m}),$$

where each e_i, $1 \leqq i \leqq k$, must be either x_j or x_j' since there is only one occurrence of $f(\cdot)$. We shall define a finite sequence of vectors $E^{(j)} = \langle e_1^{(j)}, \cdots, e_{k+m}^{(j)} \rangle$ by defining

$$e_i^{(0)} = \begin{cases} x & \text{if } 1 \leqq i \leqq k, \\ y_i & \text{if } k+1 \leqq i \leqq k+m, \end{cases}$$

and then extending $E^{(0)}, \cdots, E^{(j)}$ to $E^{(j+1)}$ for as many steps as we can by requiring that $f(e_1^{(j+1)}, \cdots, e_{k+m}^{(j+1)})$ be the single occurrence of $f(\cdot)$ on the right-hand side of the kth defining equation of R (if it is applicable) when the left-hand

side is $f(e_1^{(j)}, \cdots, e_{k+m}^{(j)})$. If the kth defining equation is inapplicable, the sequence terminates with the jth term $E^{(j)}$. Since R is pure and the kth defining equation includes only one occurrence of $f(\cdot)$ on its right-hand side, if $E^{(j)}$ is extendible to $E^{(j+1)}$, then for $1 \leqq i \leqq k$ we have

$$\begin{aligned} e_i^{(j+1)} &\geqq \min(e_1^{(j)}-1, \cdots, e_k^{(j)}-1) \\ &= \min(e_1^{(j)}, \cdots, e_k^{(j)})-1 \\ &\geqq \cdots \geqq \min(x_1', \cdots, x_k')-(j+1) \end{aligned}$$

and thus $e_i^{(j)} > 0$ at least for $0 \leqq j \leqq \min(x_1, \cdots, x_k)$ so that we can extend the sequence at least to $E^{(\min(x_1, \cdots, x_k))}$. Thus if we set $M = \min(x_1, \cdots, x_k)$, then

$$\begin{aligned} & t_0 f(x_1', \cdots, x_k', y_1, \cdots, y_m) \\ &\quad = t_0 f(e_1^{(1)}, \cdots, e_{k+m}^{(1)})+1 \\ &\quad = t_0 f(e_1^{(2)}, \cdots, e_{k+m}^{(2)})+2 \\ &\quad \vdots \\ &\quad = t_0 f(e_1^{(M)}, \cdots, e_{k+m}^{(M)})+M \\ &\quad \geqq M+1 = \min(x_1, \cdots x_k)+1, \end{aligned}$$

as required. □

Thus if the kth defining equation of a pure k-recursive schema has only one occurrence of $f(\cdot)$ on the right-hand side, then it has at least linear complexity on a RASP (or exponential complexity in terms of the lengths of the inputs). The next four theorems give conditions for exponential complexity on a RASP.

THEOREM 7.2. *If the k-recursive schema R is pure and the k-th defining equation of R includes two disjoint occurrences of $f(\cdot)$ on the right-hand side, both of which have x_j as their j-th argument for some $j \leqq k$, then*

$$t_0 f(x_1', \cdots, x_k', y_1, \cdots, y_m) \geqq 2^{\min(x_1' \cdots, x_k')+1}-2.$$

Proof. Let the two disjoint occurrences of $f(\cdot)$ mentioned in the theorem be

$$f(e_1, \cdots, e_{j-1}, x_j, e_{j+1}, \cdots, e_{k+m})$$

and

$$f(\bar{e}_1, \cdots, \bar{e}_{j-1}, x_j, \bar{e}_{j+1}, \cdots, \bar{e}_{k+m}).$$

Then

$$\begin{aligned} & t_0 f(x_1', \cdots, x_k', y_1, \cdots, y_m) \\ & \quad \geqq t_0 f(e_1, \cdots, e_{j-1}, x_j, e_{j+1}, \cdots, e_{k+m}) \\ & \qquad + t_0 f(\bar{e}_1, \cdots, \bar{e}_{j-1}, x_j, \bar{e}_{j+1}, \cdots, \bar{e}_{k+m})+2 \end{aligned} \tag{7.2}$$

and

$$\min(e_1, \cdots, e_k, \bar{e}_1, \cdots, \bar{e}_k) \geqq \min(x_1, \cdots, x_k). \tag{7.3}$$

Thus setting $x_j = 0$, we have

$$t_0 f(x'_1, \cdots, x'_{j-1}, 1, x_{j+1}, \cdots, x'_k, y_1, \cdots, y_m)$$
$$\geqq t_0 f(e_1, \cdots, e_{j-1}, 0, e_{j+1}, \cdots, e_{k+m})$$
$$+ t_0 f(\bar{e}_1, \cdots, \bar{e}_{j-1}, 0, \bar{e}_{j+1}, \cdots, \bar{e}_{k+m}) + 2 \geqq 2.$$

Suppose, as an inductive assumption, that

$$t_0 f(e_1, \cdots, e_{j-1}, x_j, e_{j+1}, \cdots, e_{k+m})$$
$$\geqq 2^{\min(e_1, \cdots, e_{j-1}, x_j, e_{j+1}, \cdots, e_k)+1} - 2$$

for all $e_1, \cdots, e_{j-1}, e_{j+1}, \cdots, e_k \geqq 1$. Then it follows that

$$t_0 f(x'_1, \cdots, x'_k, y_1, \cdots, y_m)$$
$$\geqq t_0 f(e_1, \cdots, e_{j-1}, x_j, e_{j+1}, \cdots, e_{k+m})$$
$$+ t_0 f(\bar{e}_1, \cdots, \bar{e}_{j-1}, x_j, \bar{e}_{j+1}, \cdots, \bar{e}_{k+m})$$
$$\geqq 2^{\min(e_1, \cdots, e_{j-1}, x_j, e_{j+1}, \cdots, e_k)+1} - 2$$
$$+ 2^{\min(\bar{e}_1, \cdots, \bar{e}_{j-1}, x_j, \bar{e}_{j+1}, \cdots, \bar{e}_k)+1} - 2 + 2$$
$$\geqq 2 \cdot 2^{\min(e_1, \cdots, e_k, \bar{e}_1, \cdots, \bar{e}_k, x_j)+1} - 2$$
$$\geqq 2 \cdot 2^{\min(x_1, \cdots, x_k)+1} - 2$$
$$= 2^{\min(x_1, \cdots, x_k)+1} - 2,$$

by (7.2) and (7.3), as required. □

THEOREM 7.3. *If the k-recursive schema R is pure and the k-th defining equation of R includes two nested occurrences of $f(\cdot)$ which both have x_j as their j-th argument for some $j \leqq k$ and if, in addition,*

$$f(x_1, \cdots, x_k, y_1, \cdots, y_m) \geqq \min(x_1, \cdots, x_k)$$

for all $x_1, \cdots, x_k, y_1, \cdots, y_m$, then

$$t_0 f(x'_1, \cdots, x'_k, y_1, \cdots, y_m) \geqq 2^{\min(x'_1, \cdots, x'_k)+1} - 2.$$

Proof. The proof is similar to those of Theorems 7.1 and 7.2 and is left to the reader. □

It is, of course, not in general possible to recursively decide whether $f(x_1, \cdots, x_k, y_1, \cdots, y_m) \geqq \min(x_1, \cdots, x_k)$ for all $x_1, \cdots, x_k, y_1, \cdots, y_m$ as the conditions of the last theorem require. On the other hand, we can clearly come up with a variety of simple conditions on the $\pi_i(\cdot)$'s and $g_i(\cdot)$'s occurring in R which will guarantee that the conditions hold. Since we intend these results to have practical application, we choose this more optimistic interpretation of the theorem.

THEOREM 7.4. *If the k-recursive schema R is pure and the k-th defining equation of R includes two disjoint occurrences of $f(\cdot)$ which do not have any x_j occurring as the j-th argument of both of them for $1 \leqq j \leqq k$, then*

$$t_0 f(x'_1, \cdots, x'_k, y_1, \cdots, y_m) > 2^{\min(x'_1, \cdots, x'_k)}.$$

Proof. The proof is left to the reader. □

A combination of the techniques used in proving Theorems 7.3 and 7.4 will establish

THEOREM 7.5. *If the k-recursive schema R is pure and the k-th defining equation of R includes on the right-hand side two nested occurrences of $f(\cdot)$ which do not have an x_j occurring as the j-th argument of both for $1 \leqq j \leqq k$ and if, in addition,*

$$f(x_1, \cdots, x_k, y_1, \cdots, y_m) \geqq \min(x_1, \cdots, x_k)$$

for all $x_1, \cdots, x_k, y_1, \cdots, y_m$, then

$$t_0 f(x'_1, \cdots, x'_k, y_1, \cdots, y_m) > 2^{\min(x'_1, \cdots, x'_1)}.$$

Proof. The proof is left to the reader. □

It will be noted that the above theorems all have conclusions which provide lower bounds for the complexity of $f(x'_1, \cdots, x'_k, y_1, \cdots, y_m)$, i.e., function values whose recursion arguments are all nonzero, and say nothing about the case in which one or more of the recursion arguments are zero. This is because the conditions of the theorems do not in any way constrain the complexity of $f(x'_1, \cdots, x'_{j-1}, 0, x_{j+1}, \cdots, x_k, y_1, \cdots, y_m)$ for $j < k$, as will be clear from the following example. Let R be the recursion

$$\begin{aligned} &f(0, x_2, y_1) := \pi_1(x_2, y_1), \\ &f(x'_1, 0, y_1) := g_1(x_1, y_1), \\ &f(x'_1, x'_2, y_1) := g_2(f(x'_1, x_2, y_1), f(x_1, x_1, y_1)). \end{aligned}$$

Then $T_0 R$ is given by

$$\begin{aligned} &t_0 f(0, x_2, y_1) := 0, \\ &t_0 f(x'_1, 0, y_1) := 0, \\ &t_0 f(x'_1, x'_2, y_1) := t_0 f(x'_1, x_2, y_1) + t_0 f(x_1, x_1, y_1) + 2, \end{aligned}$$

so that

$$t_0 f(x_1, x_2, y_1) \begin{cases} = 0 & \text{if } x_1 = 0 \text{ or } x_2 = 0, \\ \geqq 2^{\min(x_1, x_2)} & \text{if } x_1 > 0 \text{ and } x_2 > 0, \end{cases}$$

by Theorem 7.4. On the other hand, inspection of the definition of j-pure for $j < k$ and of the details of the proofs of Theorems 7.1–7.5 shows that we can obtain similar results for j-pure schemata, thus establishing analogous lower bounds for the complexity of

$$f(x'_1, \cdots, x'_{j-1}, x_j, 0, x_{j+2}, \cdots, x_k, y_1, \cdots, y_m). \tag{7.4}$$

Because of our concern for practical results from this work, we choose to omit such results since generally in defining a k-recursive schema one will be interested

only in the values of the function computed when all the recursion arguments are nonzero.

It is tempting to think that a lower bound for the complexity of (7.4) for some $j<k$ will automatically produce a lower bound for the complexity of $f(x'_1, \cdots, x'_k, y_1, \cdots, y_m)$. This is, however, also not the case, as can be seen from this example:

$$f(0, x_2, y_1) := \pi_1(x_2, y_1),$$
$$f(x'_1, 0, y_1) := f(x_1, 0, f(x_1, 0, y_1)),$$
$$f(x'_1, x'_2, y_1) := \pi_2,$$

for which the T_0 measure if given by

$$t_0 f(0, x_2, y_1) := 0,$$
$$t_0 f(x'_1, 0, y_1) := t_0 f(x_1, 0, f(x_1, 0, y_1)) + t_0 f(x_1, 0, y_1) + 2,$$
$$t_0 f(x'_1, x'_2, y_1) := 0,$$

so that

$$t_0 f(x_1, x_2, y_1) = \begin{cases} 0 & \text{if } x_1 = 0 \text{ or } (x_1 > 0 \text{ and } x_2 > 0), \\ 2^{x_1+1} - 2 & \text{if } x_1 > 0 \text{ and } x_2 = 0. \end{cases}$$

The results of Theorems 7.1–7.5 can, of course, be extended in a variety of other ways which are less satisfying than the original theorems since they depend on nonsyntactic features of the schemata more heavily. For example, we can show that if a k-recursive schema R (not necessarily pure) has a kth defining equation which includes two disjoint occurrences of $f(\cdot)$ on the right-hand side, namely

$$f(e_1, \cdots, e_{k+m}) \quad \text{and} \quad f(\bar{e}_1, \cdots, \bar{e}_{k+m}),$$

with the property that there exists a constant $c>0$ such that

$$\min(e_1, \cdots, e_k, \bar{e}_1, \cdots, \bar{e}_k) \geqq \min(x_1, \cdots, x_k) - c$$

for the $x_1, \cdots, x_k$ occurring on the left-hand side of the kth defining equation, then

$$t_0 f(x'_1, \cdots, x'_k, y_1, \cdots, y_m) \geqq 2^{\min(x'_1, \cdots x'_k)/c}.$$

Another issue of interest here is whether or not it is decidable whether an **MR** schema satisfying the conditions of one of these theorems has the minimal complexity or greater complexity. This is unclear for the minimal measure T_0, though it seems to be decidable for the cases in which the conditions of the theorem are decidable, but it is easy to show that there exist measures for which it is undecidable.

THEOREM 7.6 *There exists a measure T such that it is recursively undecidable whether a k-recursive schema satisfying the conditions of Theorem 7.2 has the minimal complexity, i.e., whether*

$$tf(x'_1, \cdots, x'_k, y_1, \cdots, y_m) = (2^{\min(x'_1, \cdots, x'_k)+1} - 2) \cdot c_R.$$

Proof. Let $\pi_1(\cdot)$ be as in the proof of Theorem 6.1, let $\pi_2 : \mathbf{N}^2 \to \mathbf{N}$ be such that $t\pi_2(i, x) = \pi_1(i, x)$, and let $\pi_3 : \mathbf{N}^3 \to \mathbf{N}$ be such that $t\pi_3(\cdot) \equiv 0$ in the measure T. Let R_i denote the **MR** schema

$$f(0, x_2, y_1) := y_1,$$
$$f(x_1', 0, y_1) := y_1,$$
$$f(x_1', x_2', y_1) := \pi_3(f(x_1', x_2, y_1), f(x_1', x_2, y_1), \pi_2(i, y_1)).$$

Then TR_i is given by

$$tf(0, x_2, y_1) := 0,$$
$$tf(x_1', 0, y_1) := 0,$$
$$tf(x_1', x_2', y_1) := 2 \cdot tf(x_1', x_2, y_1) + 2 \cdot c_R + \pi_1(i, y_1),$$

so that

$$tf(x_1', x_2', y_1) = (2^{\min(x_1, x_2)} - 2) \cdot c_R$$

iff $\pi_1(i, y_1) = 0$ for all y_1, that is, iff $\phi_i(i)$ diverges. □

A similar result is available for the minimal measure T_0 if we weaken the purity requirement in the most minimal way or if we increase the lower bound ever so slightly. For example, the schema R_i given by

$$f(0, x_2, x_3, y_1) := y_1,$$
$$f(x_1', 0, x_3, y_1) := f(0, x_1, x_3, y_1),$$
$$f(x_1', x_2', 0, y_1) := f(\pi_1(i, y_1), 0, 0, y_1),$$
$$f(x_1', x_2', x_3', y_1) := \pi_2(f(x_1, x_2, x_3, y_1), f(x_1, x_2, x_3, y_1))$$

has the T_0 complexity

$$t_0 f(0, x_2, x_3, y_1) := 0,$$
$$t_0 f(x_1', 0, x_3, y_1) := t_0 f(x_1, x_3, y_1) + 1,$$
$$t_0 f(x_1', x_2', 0, y_1) := t_0 f(\pi_1(i, y_1), 0, 0, y_1) + 1,$$
$$t_0 f(x_1', x_2', x_3', y_1) := 2 \cdot t_0 f(x_1, x_2, x_3, y_1) + 2$$

which satisfies

$$t_0 f(0, x_2, x_3, y_1) = 0,$$
$$t_0 f(x_1', 0, x_3, y_1) = 1,$$
$$t_0 f(x_1', x_2', 0, y_1) = \pi_1(i, y_1) + 1,$$
$$t_0 f(x_1', x_2', x_3', y_1) = 2 \cdot t_0 f(x_1, x_2, x_3, y_1) + 2$$

and thus R_i has the minimal T_0 complexity among *this particular class of schemata* iff $\phi_i(i)$ diverges.

8. Conclusions. In this paper we have defined and studied a set of practical computational complexity measures for multiple recursive schemata. In this section we shall discuss the import of these results and consider a number of questions raised by them.

We first recall the three objectives enumerated in § 4 and consider to what degree we have accomplished them. We regard the first objective, that of demonstrating "that multiple recursion is a thoroughly unfeasible computational tool in practice", as having been accomplished by Theorems 7.2–7.5. Lachlan's [16] Lemma 2 which implies that to define a nonprimitive recursive function a multiple recursion *must* have nested occurrences of $f(\cdot)$ in its defining equations, combined with a survey of the literature, such as Péter [25], which shows that virtually all the **MR** schemata occurring in real function definitions are pure, demonstrates the applicability of these results to real computation. Since they all produce minimally exponential complexity bounds, which translate directly into the time necessary to perform the substitutions in recursion, they demonstrate the futility of using multiple recursion in practical computation.

The second stated objective was to begin to develop a list of criteria which make a complexity measure natural. We have followed Hartmanis' suggestions in *On the problem of finding natural computational complexity measures* [13, pp. 17–18] with regard to incorporating step counting and structural considerations in our definitions of the measures. Our work points up the value of these suggestions, since our results stem in large part from the way in which the measures reflect the structural properties of multiple recursion, but considerable further work is necessary in this area before we can say that a theory of natural complexity measures has been developed.

Thirdly, as just noted, we have certainly taken into account the structure of multiple recursion in defining the complexity measures, and this has clearly been advantageous in the results obtained.

In § 4 we discussed general criteria for computation rules and three specific computation rules other than leftmost-innermost before settling on it as the basis for the measures we have concentrated our attention on. We noted there that the normal rule is particularly interesting because, if properly implemented,[2] it minimizes the number of substitutions necessary in some language models. Clearly we can define complexity measures based on these other computation rules, though the definitions will be more involved. For example, a measure based on the normal rule must take into account that in evaluating, say, $f(2, f(6, 3, 1), 5)$, we must substitute for the inner $f(\cdot)$ rather than the outer one even though it is the case that $f(6, 3, 1) = 0$ because we have in the expression the character string "$f(6, 3, 1)$" rather than "0". That is, we must be able to distinguish between a character string which denotes an instance of $f(\cdot)$ and one which denotes a member of **N**. This points in the direction of defining formal substitutions of character strings and mappings between character strings and numbers and then deriving measures from that relationship. We have done this informally with regard to the leftmost-innermost rule by specifying that T could be viewed as a syntactic transformation on character strings.

[2] See Vuillemin [31] for a discussion of the details of implementing the normal rule.

Another extension of our measures which is worthy of consideration is to define measures which explicitly take simplification of expressions into account. An excellent basis for such work is contained in Cadiou's doctoral thesis. He discusses what he calls "standard simplification schemas" [sic.] which intuitively are simplifications which exploit the property that for some given function $g(\cdot)$, when some of the arguments to $g(\cdot)$ are constants, the value of $g(\cdot)$ is a constant independent of the other arguments. It should be relatively easy to account for such simplifications in complexity measures. There are, of course, much more powerful types of simplifications, such as that involved in passing from (4.1) to (4.3). The theory behind such simplifications has not been extensively explored.

Finally, the ideas underlying our measures can be extended to other types of schemata, such as flowchart schemata and recursion schemata. To some extent this has been done by Weihrauch, who develops a rather similar model of complexity for flowchart and Yanov schemata with notable results. In a future paper we shall present a generalization of our complexity measures which captures the advantages of both Weihrauch's methods and our own.

REFERENCES

[1] W. ACKERMANN, *Zum Widerspruchsfreiheit der Zahlentheorie*, Math. Ann., 117 (1940), pp. 162–194.

[2] M. BLUM, *A machine-independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach., 14 (1967), pp. 322–336.

[3] J. M. CADIOU, *Recursive definitions of partial functions and their computations*, STAN-CS-266-72, Dept. Comput. Sci., Stanford Univ., Stanford, Calif., 1972.

[4] J. P. CLEAVE, *A hierarchy of primitive recursive functions*, Z. Math. Logik Grundlagen Math., 9 (1963), pp. 331–345.

[5] R. L. CONSTABLE, *Extending and refining hierarchies of computable functions*, CS Tech. Rep. 25, Univ. of Wis., Madison, 1968.

[6] ———, *Subrecursive programming languages III: The multiple recursive functions R^n*, Symp. on Computers and Automata, Polytechnic Inst. of Brooklyn, April 1971, pp. 393–410.

[7] ———, *Type two computational complexity*, Proc. 5th Ann. ACM Symp. on Theory of Computing, May 1973, pp. 108–121.

[8] R. L. CONSTABLE AND D. GRIES, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[9] R. L. CONSTABLE AND S. S. MUCHNICK, *Subrecursive program schemata I: Undecidable equivalence problems; II: Decidable equivalence problems*, J. Comput System Sci., 6 (1972), pp. 460–537.

[10] C. C. ELGOT AND A. ROBINSON, *Random-access stored program machines, an approach to programming languages*, J. Assoc. Comput. Mach., 11 (1964), pp. 365–399.

[11] A. GRZEGORCZYK, *Some classes of recursive functions*, Rozprawy Matematyczne, 4 (1953), pp. 4–45.

[12] J. HARTMANIS, *Computational complexity of random access stored program machines*, Math. Systems Theory, 5 (1971), pp. 232–245.

[13] ———, *On the problem of finding natural computational complexity measures*, TR 73–175, Dept. Comput. Sci., Cornell Univ., Ithaca, N.Y., 1973.

[14] J. HARTMANIS AND J. E. HOPCROFT, *An overview of the theory of computational complexity*, J. Assoc. Comput. Mach., 18 (1971), pp. 444–475.

[15] D. HILBERT, *Über das Unendliche*, Math. Ann., 95 (1926), pp. 161–190.

[16] A. H. LACHLAN, *Multiple recursion*, Z. Math. Logik Grundlagen Math, 8 (1962), pp. 81–107.

[17] D. C. LUCKHAM, D. M. R. PARK AND M. S. PATERSON, *On formalised computer programs*, J. Comput. System Sci., 4 (1970), pp. 220–249.

[18] Z. MANNA, S. NESS AND J. VUILLEMIN, *Inductive methods for proving properties of programs*, Proc. ACM Conf. on Proving Assertions about Programs; SIGACT News, No. 14, and SIGPLAN Notices, vol. 7, no. 1, January 1972, pp. 27–50.

[19] S. S. MARČENKOV, *Limited multiple recursion in classes of primitive recursive functions*, Cybernetics, 6 (1970), pp. 35–39.

[20] S. MCCLEARY, *Primitive recursive computations*, Notre Dame J. Formal Logic, 8 (1968), pp. 311–317.

[21] A. R. MEYER AND D. M. RITCHIE, *A classification of the recursive functions*, Z. Math. Logik Grundlagen Math., 18 (1972), pp. 71–82.

[22] ———, *Computational complexity and program structure*, IBM Res. Paper RC-1817, 1967.

[23] R. MOLL, *Complexity classes of recursive functions*, Project MAC Rep. TR-110, Mass. Inst. of Tech., Cambridge, Mass., 1973.

[24] S. S. MUCHNICK AND R. L. CONSTABLE, *Multiple recursive operators have exponential complexity*, 8th Princeton Conf. on Information Sci. and Systems, March 1974.

[25] R. PÉTER, *Rekursive Funktionen*, Akadémiai Kiadó, Budapest, 1951.

[26] ———, *Über die mehrfache Rekursion*, Math. Ann., 113 (1936), pp. 489–527.

[27] ———, *Zusammenhang der mehrfachen und transfiniten Rekursion*, J. Symbolic Logic, 15 (1950), pp. 248–272.

[28] J. W. ROBBIN, *Subrecursive hierarchies*, Ph.D. dissertation, Dept. of Math., Princeton Univ., Princeton, N.J., 1965.

[29] J. RUTLEDGE, *On Ianov's program schemata*, J. Assoc. Comput. Mach., 11 (1964), pp. 1–9.

[30] C.-P. SCHNORR, *Does the computational speed-up concern programming?*, Automata, Languages and Programming, M. Nivat, ed., North-Holland, Amsterdam, 1973, pp. 585–591.

[31] J. E. VUILLEMIN, *Proof techniques for recursive programs*, Ph.D. dissertation, STAN-CS-73-393, Dept. of Comput. Sci., Stanford Univ., Stanford, Calif., 1973.

[32] K. WEIHRAUCH, *On the computational complexity of program schemata*, Tech. Rep. TR 74–196, Dept. of Comput. Sci., Cornell Univ., Ithaca, N.Y., 1974.

[33] Y. I. YANOV, *The logical schemes of algorithms*, Problems of Cybernetics (USSR), (1960), pp. 82–140.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

A POWERDOMAIN CONSTRUCTION*

G. D. PLOTKIN†

Abstract. We develop a powerdomain construction, $\mathcal{P}[\cdot]$, which is analogous to the powerset construction and also fits in with the usual sum, product and exponentiation constructions on domains. The desire for such a construction arises when considering programming languages with nondeterministic features or parallel features treated in a nondeterministic way. We hope to achieve a natural, fully abstract semantics in which such equivalences as $(p\ par\ q) = (q\ par\ p)$ hold. The domain $(D \to \text{Truthvalues})$ is not the right one, and instead we take the (finitely) generable subsets of D. When D is discrete they are ordered in an elementwise fashion. In the general case they are given the coarsest ordering consistent, in an appropriate sense, with the ordering given in the discrete case. We then find a restricted class of algebraic inductive partial orders which is closed under $\mathcal{P}[\cdot]$ as well as the sum, product and exponentiation constructions. This class permits the solution of recursive domain equations, and we give some illustrative semantics using $\mathcal{P}[\cdot]$.

It remains to be seen if our powerdomain construction does give rise to fully abstract semantics, although such natural equivalences as the above do hold. The major deficiency is the lack of a convincing treatment of the fair parallel construct.

1. Introduction. When one follows the Scott–Strachey approach to the semantics of programming languages, various constructions on domains arise naturally. These include sum, product and exponentiation constructions. Their use is illustrated in [12], [18]. Encountering languages with nondeterministic and parallel programming features induces the desire for a powerdomain construction analogous to the powerset construction on sets. Unfortunately, domains of the form $(D \to \text{Truthvalues})$ will not do—as will be seen—and we present here an alternative, rather more complicated proposal.

Milner [10], [11] handled nondeterminism by using oracles. Unfortunately, the resulting semantics does not give some intuitive equivalences since; for example, the programs $(p\ or\ q)$ and $(q\ or\ p)$ have different meanings in general. Let us say that two well-formed program fragments are behaviorally equivalent iff whenever embedded in a context to form a program, they give rise to the same behavior. Behavior itself is to be defined in some operational way. Relative to some such notion we say that a semantics is fully abstract iff behavioral and denotational equivalence coincide [11], [13], [19]. We would expect that $(p\ or\ q)$ and $(q\ or\ p)$ would be behaviorally equivalent. So Milner's semantics is not fully abstract.

Milner asked if there was a generalization to relations of the notion of a continuous function. Rather than consider relations $R \subseteq D \times E$ directly we define $\mathcal{P}[E]$, the powerdomain of E, and use continuous functions $R : D \to \mathcal{P}[E]$. As a result, we obtain a semantics in which the programs $(p\ or\ q)$ and $(q\ or\ p)$ always do have the same meaning. But it remains an open question whether we thus achieve a fully abstract semantics.

We begin by considering a simple language with a "nondeterministic branch" feature. In this setting it is quite natural to consider sets when looking for a

* Received by the editors April 30, 1975, and in revised form December 10, 1975.

† Department of Artificial Intelligence, University of Edinburgh, EH1 9NW Scotland. This work was supported by an SRC Grant.

semantics. Indeed, a straightforward construction is available which we believe has intuitive appeal. This construction forms the basis of the subsequent more general powerdomain construction.

The need for that is demonstrated by considering a more elaborate language with a simple parallel processing facility. In particular, one wants to find domains satisfying recursion equations involving the hypothetical powerdomain construction $\mathcal{P}[\cdot]$, just as equations involving $+$, $\times$ and $\rightarrow$ were considered in the deterministic case. Our approach is to use consistency criteria to determine the ordering of $\mathcal{P}[D]$, and to consider only certain subsets of D as candidate members of $\mathcal{P}[D]$. This gives a rather indirect definition of $\mathcal{P}[D]$. The main body of the paper considers a wide class of domains D for which it is possible to establish a direct definition of $\mathcal{P}[D]$, and examines some of its properties. This class allows us to define continuous functions and predicates analogous to some standard ones on sets and also to solve recursive domain equations involving $\mathcal{P}[\cdot]$. The final part of the paper applies these results by giving an illustrative semantics for the simple language considered. We also give one for a language of Milner's which has more extended multiprocessing features.

2. Establishing a definition. The first programming language considered has simple nondeterministic choice points. The illustrative programs in Fig. 1 are written in it. A formal definition will be given later.

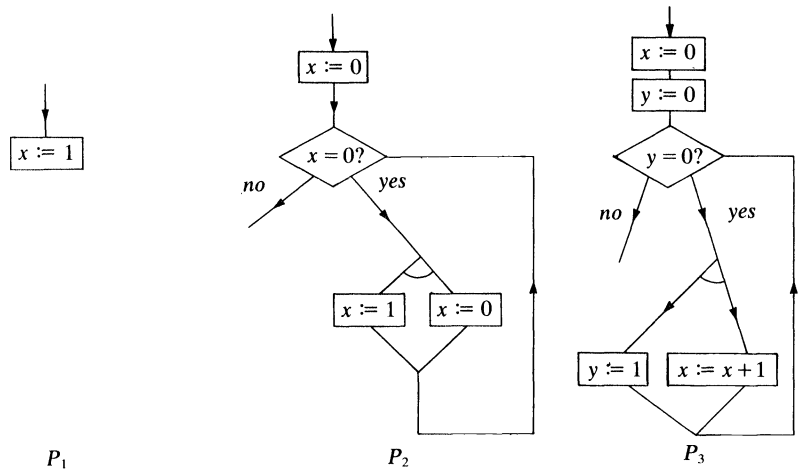


FIG. 1

In this language, states are integer vectors of the appropriate length, and it is clear how execution sequences are defined. From a given starting vector an execution can, in general, result in one of several possible final vectors and may even fail to terminate. For example, the first program P_1 always terminates with $x = 1$; P_2 has many possible execution sequences. One does not terminate, but the others all terminate after varying amounts of time with $x = 1$. An execution of P_3 either does not terminate or else terminates with $y = 0$ and $x =$ an arbitrary number. It is therefore natural to let the meaning of a program, P, be a function.

DEFINITION. An inductive partial order (ipo) is a partial order with a bottom element in which every directed subset has a least upper bound.

Now let S be the set of possible states of the program P, and let $S_\perp = S \cup \{\perp\}$ be the ipo ordered by: $x \sqsubseteq y$ iff $x = \perp$ or $x = y$. The meaning of P will be a function p from S to $\mathcal{P}(S_\perp)$, the collection of subsets of $S_\perp$. We want $s' \in p(s)$ iff either $s' = \perp$ and, starting from s, P has a nonterminating execution sequence or else $s' \neq \perp$ and, starting from s, P has an execution sequence ending in s'.

Notice the use of ipo's here. Sometimes they are called cpo's—complete partial orders—instead. At the moment, we are using them for convenience instead of complete lattices: including a top element would, we believe, not permit so natural a development. Later, it seems to be essential to use ipo's instead of complete lattices. The construction of $S_\perp$ from S is an example of a general construction, which will be used again.

If the meaning of P_i is p_i $(i = 1, 2, 3)$ then we have $p_1(\langle m, n\rangle) = \{\langle 1, n\rangle\}$, $p_2(\langle m, n\rangle) = \{\perp, \langle 1, n\rangle\}$ and $p_3(\langle m, n\rangle) = \{\perp\} \cup \{\langle j, 0\rangle | j \geqq 0\}$. The use of functions rather than relations as meanings reflects the input-output asymmetry. Whichever is used, we feel it would be a mistake to take meanings in either $S \to \mathcal{P}(S)$ or $\mathcal{P}(S \times S)$ as we want to distinguish P_1 from P_2. That is, we do not want nondeterminism to mask nontermination. This difficulty with using the relational approach [1], [2], [7] to handle nondeterminism is noted by Milner.

How are we to define the ordering relation, $\sqsubseteq$, on meanings? Since each p is a function, $\sqsubseteq$ could be defined pointwise if only we had a definition of $\sqsubseteq$ on sets. Now in the deterministic case, the orderings arose because one introduced $\perp$ in order to be able to deal with partial functions as total functions. Then the order reflected the fact of partialness and was induced by: $\perp \sqsubseteq$ anything.

In our case the analogous course is to define the ordering on sets elementwise, just as it was defined coordinatewise on previous occasions. So, for sets X and Y, in $\mathcal{P}(S_\perp)$, we put $X \sqsubseteq Y$ iff Y is obtained from X by replacing $\perp$ in X by some nonempty set. That is, $X \sqsubseteq Y$ if $(\perp \notin X$ and $Y = X)$ or else $(\perp \in X$ and $Y \supseteq (X - \{\perp\})$ and Y is nonempty). There is a neater definition. Let D be an arbitrary ipo. The *Milner ordering* $\sqsubseteq_M$ on $\mathcal{P}(D)$ is defined by:

$$X \sqsubseteq_M Y \quad \text{iff} \quad (\forall x \in X.\exists y \in Y.x \sqsubseteq y) \quad \text{and} \quad (\forall y \in Y.\exists x \in X.x \sqsubseteq y).$$

It is the ordering on subsets of D induced elementwise by the ordering of D; in particular it is the same as the one defined above for $\mathcal{P}(S_\perp)$ when $D = S_\perp$. For arbitrary ipo's $\sqsubseteq_M$ is only a quasi-order, for if $x \sqsubseteq y \sqsubseteq z$ are three distinct elements in D, then $\{x, z\} =_M \{x, y, z\}$, where $=_M$ is the equivalence induced by $\sqsubseteq_M$.

We pause to consider a useful fact about $\sqsubseteq_M$. Suppose $f : D \to E$ where D and E are ipo's.

Notation. If $X \subseteq D$, then $f(X) =_{\text{def}} \{f(x) | x \in X\}$.

Now it is not hard to see that if $X, Y \subseteq D$ and $X \sqsubseteq_M Y$, then $f(X) \sqsubseteq_M f(Y)$, as f is monotonic.

In the present case $(S_\perp)$, it is tempting to allow $X \sqsubseteq Y$ if $X \subseteq Y$ although one can hardly then claim $\sqsubseteq$ as a "less defined than" ordering. But then $\{\perp, s\} \sqsubseteq \{s\}$, by the elementwise criterion and $\{s\} \sqsubseteq \{\perp, s\}$ by the subset one for any $s \in S$, and it follows that P_1 and P_2 have equivalent meanings—against our wishes. So the

temptation will be resisted. However, both $\cup$ and $\{\ \}$ will be seen to be continuous functions. So a proof system based on $\sqsubseteq$ with symbols for $\cup$ and $\{\ \}$ can be used to prove both subset and membership relations if required, as $X \subseteq Y$ iff $X \cup Y = Y$ and $s \in X$ iff $\{s\} \subseteq X$.

Having considered the ordering on meanings, which meanings should be considered? Again we want to know which members of $\mathcal{P}(S_\perp)$ to allow. Since $p(s) \neq \emptyset$, even if P does not terminate on s, we exclude the empty set.

Now the set of all initial segments of execution sequences of a given nondeterministic program P, starting from a given state, will form a tree. The branching points will correspond to the choice points in the program. Since there are always only finitely many alternatives at each such choice point, the branching factor of the tree is always finite. That is, the tree is finitary. Now Konig's lemma says that if every branch of a finitary tree is finite, then so is the tree itself. In the present case this means that if every execution sequence of P terminates, then there are only finitely many execution sequences. So if an output set of P is infinite it must contain $\perp$. Therefore we require infinite sets in $\mathcal{P}[S_\perp]$ to contain $\perp$.

DEFINITION. $\mathcal{P}[S_\perp]$, the powerdomain of $S_\perp$, is the set $\{X \subseteq S_\perp | X \neq \emptyset$, and X is finite or contains $\perp\}$ ordered by $\sqsubseteq_M$.

Notice that we can define $\mathcal{P}[D]$ for any denumerable discrete ipo D analogously:

DEFINITION. An ipo D is *ω-discrete* iff it is denumerable and for x, y in D, $x \sqsubseteq y$ iff $x = \perp$ or $x = y$. The ω-discrete ipo's are just those of the form $X_\perp$, where X is a denumerable set.

DEFINITION. $\mathcal{P}[D]$ is the set $\{X \subseteq D | X \neq \emptyset$, and X is finite or contains $\perp\}$ ordered by $\sqsubseteq_M$, when D is ω-discrete.

Clearly $S_\perp$ is ω-discrete and so, for example, is $N_\perp$ defined similarly from the set of nonnegative integers. Other examples are $\mathbb{1} = \{\perp\}$, the one-point lattice and $\mathbb{O} = \{\perp, \top\}$ the two-point complete lattice and $\mathbb{T} = \text{Truthvalues} = \{\perp, \textit{true}, \textit{false}\}$. The elements of ω-discrete ipo's can often be taken to correspond to discrete items of output. As such, the same justification given for the definition of $\mathcal{P}[S_\perp]$ applies also to that of $\mathcal{P}[D]$ when D is ω-discrete.

THEOREM 1. *$\mathcal{P}[D]$ is an* ipo *in which every element is the limit of an increasing sequence of finite elements. The functions $\cup$ and $\{\cdot\}$ are continuous.*

Proof. First $\sqsubseteq_M$ is a partial order, for if $X =_M Y$ and $x \in X$, then either $x \neq \perp$ and $x \in Y$ as $X \sqsubseteq_M Y$ or $x = \perp$ and $x \in Y$ as $X \sqsupseteq_M Y$. Therefore $X \subseteq Y$ and similarly $Y \supseteq X$, which proves antisymmetry. Transitivity and reflexivity are clear. The set $\{\perp\}$ is the $\perp$ of $\mathcal{P}[D]$.

We now prove that if X, Y are sets in $\mathcal{P}[D]$ with an upper bound Z and X does not contain $\perp$, then $X \sqsupseteq_M Y$. For if $y \in Y$, there is a $z \in Z$ and an $x \in X$ such that $y \sqsubseteq z \sqsupseteq x$. As $x \neq \perp$, $y \sqsubseteq x$. Similarly if $x \in X$, there is a $z \in Z$ and a $y \in Y$ such that $x \sqsubseteq z \sqsupseteq y$. As $x \neq \perp$, $x \sqsupseteq y$. By the definition of $\sqsubseteq_M$, $X \sqsupseteq_M Y$.

Now suppose $\mathcal{X} \subseteq \mathcal{P}[D]$ is directed. If there is a set $X \in \mathcal{X}$ which does not contain $\perp$, then, by the above remarks, $X = \bigsqcup \mathcal{X}$.

Otherwise every set in $\mathcal{X}$ contains $\perp$. Let $X^* = \cup \mathcal{X}$. If $X \in \mathcal{X}$, then as $X \subseteq X^*$ and $\perp \in X$, $X \sqsubseteq_M X^*$. If Y is any upper bound of $\mathcal{X}$, then $X^* \sqsubseteq_M Y$. For if $x \in X^*$, then there is an X in $\mathcal{X}$ which contains x and so, as $X \sqsubseteq_M Y$, there is a $y \in Y$ such

that $x \sqsubseteq y$. Conversely, if $y \in Y$, then $y \supseteq \perp \in X^*$. Thus we have proved that $X^* = \bigsqcup \mathscr{X}$. So $\mathscr{P}[D]$ is an ipo.

If $X \in \mathscr{P}[D]$, and is not finite, let $X_n = \{\perp, s_1, \cdots, s_n\}$ where $s_1, \cdots$ is a listing of X. Then $X = \bigcup_{n \geqq 0} X_n = \bigsqcup_{n \geqq 0} X_n$. It is easy to verify that the functions $\bigcup$ and $\{\ \}$ considered as members of $\mathscr{P}^2[D] \to \mathscr{P}[D]$ and $D \to \mathscr{P}[D]$, respectively, are continuous, which concludes the proof. □

It should be emphasized that if $\mathscr{X}$ is a finite directed subset of $\mathscr{P}[D]$, $\bigsqcup \mathscr{X}$ and $\bigcup \mathscr{X}$ need not be equal.

So we have our powerdomain construction for ω-discrete domains, such as $S_\perp$. Some subsidiary questions arise. First, if we had excluded $\varnothing$ but included everything else, an ipo would still have been obtained although not every element would have been a limit of finite sets. At the present level the choice of $\mathscr{P}[D]$ for ω-discrete D is governed by considerations of economy. We shall see later that a similar choice in the general case gives rise to a difficulty in our theory. Notice by the way that $\mathscr{P}[D]$ could not have been taken as $(D \to \mathbb{T})$, where a function f represented the set $\{d \in D | f(d) = \mathit{true}\}$ as no function would have represented $\{\perp\}$.

Let us put these ideas together to give the semantics for our simple language. Its grammar is specified by:

$$\nu ::= x_1 | \cdots | x_n$$

$$\tau ::= \nu | 0 | 1 | (\tau_1 + \tau_2) | (\tau_1 \dot{-} \tau_2)$$

$$\pi ::= (\nu := \tau) | (\pi_1; \pi_2) | (\pi_1 \ \mathit{or}\ \pi_2) | (\mathit{if}\ \nu\ \mathit{then}\ \pi_1\ \mathit{else}\ \pi_2) | (\mathit{while}\ \nu\ \mathit{do}\ \pi).$$

following the style of Scott and Strachey [18]. Here ν ranges over the set of variables, τ over terms and π over statements.

We take $S = \{\langle m_1, \cdots, m_n \rangle | m_i \geqq 0\}$.

For $p : S \to \mathscr{P}[S_\perp]$ let $p_\perp : (S_\perp \to \mathscr{P}[S_\perp])$ be defined by

$$p_\perp(s) = \begin{cases} \{\perp\}, & (s = \perp), \\ p(s), & (s \neq \perp). \end{cases}$$

Notice here that although S is just a set, $S \to \mathscr{P}[S_\perp]$ can be considered to be an ipo with the induced pointwise ordering. Here and elsewhere we always intend the interpretation as an ipo. When D and E are taken to be ipo's, $D \to E$ is always to be the ipo of all continuous functions from D to E.

It is not hard to show that the function which sends p to $p_\perp$ is continuous.

The combinator *: $(S \to \mathscr{P}[S_\perp])^2 \to (S \to \mathscr{P}[S_\perp])$ is defined by: $p^* q = \lambda s \{s'' | \exists s' \in p(s).s'' \in q_\perp(s')\}$. It is tedious to show directly that * is continuous, although it is clearly well-defined. An indirect proof of continuity will be given in § 6.

The combinator COND: $N_\perp \to \mathscr{P}[S_\perp] \to \mathscr{P}[S_\perp] \to \mathscr{P}[S_\perp]$ is defined by:

$$\mathrm{COND}\ x\ X\ Y = \begin{cases} \{\perp\}, & (x = \perp), \\ X, & (x = 0), \\ Y, & (x > 0). \end{cases}$$

The semantics of the language is then given by two functions $\mathcal{V}$: Terms $\to$ $(S \to N)$ and $\mathcal{M}$: Statements $\to (S \to \mathcal{P}[S_\perp])$ where:

$\mathcal{V}[\![x_i]\!](s) = (s)_i$,
$\mathcal{V}[\![0]\!](s) = 0$,
$\mathcal{V}[\![1]\!](s) = 1$,
$\mathcal{V}[\![(\tau_1 + \tau_2)]\!](s) = \mathcal{V}[\![\tau_1]\!](s) + \mathcal{V}[\![\tau_2]\!](s)$,
$\mathcal{V}[\![(\tau_1 \dot{-} \tau_2)]\!](s) = \mathcal{V}[\![\tau_1]\!](s) \dot{-} \mathcal{V}[\![\tau_2]\!](s)$,
$\mathcal{M}[\![(x_i := \tau)]\!] = \lambda s \in S.\{\langle (s)_1, \cdots, (s)_{i-1}, \mathcal{V}[\![\tau]\!](s), (s)_{i+1}, \cdots, (s)_n \rangle\}$,
$\mathcal{M}[\![(\pi_1; \pi_2)]\!] = \mathcal{M}[\![\pi_1]\!]^* \mathcal{M}[\![\pi_2]\!]$,
$\mathcal{M}[\![(\pi_1 \text{ or } \pi_2)]\!] = \lambda s \in S.\mathcal{M}[\![\pi_1]\!](s) \cup \mathcal{M}[\![\pi_2]\!](s)$,
$\mathcal{M}[\![\text{if } x_i \text{ then } \pi_1 \text{ else } \pi_2]\!] = \lambda s \in S.\mathrm{COND}(s)_i(\mathcal{M}[\![\pi_1]\!](s))(\mathcal{M}[\![\pi_2]\!](s))$,
$\mathcal{M}[\![\text{while } x_i \text{ do } \pi]\!] = Y(\lambda p \in (S \to \mathcal{P}[S_\perp]).\lambda s \in S.\mathrm{COND}((s)_i)((\mathcal{M}[\![\pi]\!]^* p)(s))(\{s\}))$.

A thorough treatment would now investigate the connection between this semantics and an operational one, as envisaged above, with extensions to more elaborate languages. One would also like to see proof systems based on $\sqsubseteq$ and consider their relationship, both practical and theoretical, to those based on the relational approach [1], [2], [7]. However, we press on to situations requiring a more involved $\mathcal{P}$.

Now suppose we introduce a parallel operation into the language by adding the clauses, $\pi ::= (\pi_1 \textit{ par } \pi_2)$.

The execution of a program is now conceived of as the performance of a sequence of elementary operations on the state. Statements of the form $(\pi_1 \textit{ par } \pi_2)$ perform an arbitrary interleaving of the elementary operations of π_1 and π_2.

The meanings of statements can no longer be functions. For example if $\pi_1 = ((x := 0); (x := x+1))$, and $\pi_2 = (x := 1)$, then although π_1 and π_2 have the same meaning as functions, $(\pi_1 \textit{ par } \pi_2)$ and $(\pi_2 \textit{ par } \pi_2)$ have quite different meanings. (This example is taken from [10], [11].)

Suppose meanings are entities $r \in R$, say, and π has meaning r. The execution of π on $s \in S$ either results in a final state, or results in some state and reaches a point in π from which computation can be resumed. From a given s, several of these possibilities can obtain. This is modeled by assuming that $r(s) \in S$ or $r(s) \in S \times R$ and we want $R \cong S \to \mathcal{P}[S_\perp + (S_\perp \times R)]$.

This domain R of resumptions is inspired by Milner's domain of processes. It is simpler, and is useful when side-effects have a fixed limited form arising from, say, a predetermined number of common registers or buffers (when a slightly different S is needed).

So we do indeed need to solve recursive domain equations involving $\mathcal{P}$. As another example, we will be able to eliminate the use of oracles in giving the semantics of Milner's multiprocessing language. This will use a domain, P, of nondeterministic processes, satisfying the equation:

$$P = V \to \mathcal{P}[L \times V \times P],$$

where V is a domain of values and L of L-values.

The method of investigation is to find a wide class of ipo's D for which $\mathcal{P}[D]$ can be defined and in which such equations can be solved. This is done by defining

first the members of $\mathcal{P}[D]$, then the ordering of $\mathcal{P}[D]$ for an arbitrary ipo D and then finding a class of ipo's in which this definition leads to pleasant and useful properties.

In the case of the $\to$ construction, continuity successfully cut down the cardinality of $D \to E$ and gave a smooth mathematical theory into which computability fitted nicely [17], [18]. For $\mathcal{P}[\,\cdot\,]$ we want an abstract notion of generable set. When $D = S_\perp$ we expect this will coincide with the finite nonempty subsets of $S_\perp$ and the infinite ones containing $\perp$, at least if the finitary tree nature of the execution sequences is a general feature. Not only ω-discrete domains figure as output domains. Suppose one added a print instruction to the language. Output would then be a sequence of integers, and the possible sequences of a given program form the branches of a finitary tree. Here outputs belong to N^ω the ipo of finite and infinite sequences of nonnegative integers with the subsequence ordering.

Such considerations suggest a definition. Let Ω be the ipo of finite and infinite sequences of 0's and 1's, with the subsequence ordering. This is the oracle ipo [8]; it is the infinite binary tree with limit points added.

DEFINITION. A subset X of D is *finitely generable* iff there is a continuous function $f : \Omega \to D$ such that $X = Bd(f) =_{\text{def}} \{f(\omega) | \omega \text{ infinite}\}$.

There is, actually, a connection with the oracle idea. Suppose we have a semantics in which the denotation of a program takes as argument an oracle ω, in Ω, which is used to determine the direction to take at choice points. Then the program will deliver a result, $f(\omega)$, in its output domain D. In the kind of semantics advocated here, the denotation of that program would not take an oracle as argument but would be, essentially, $Bd(f)$. (Certain complications, considered below, prevent it being exactly $Bd(f)$.)

When D is ω-discrete, its finitely generable subsets form the domain of $\mathcal{P}[D]$. For one can see that all the sets in $\mathcal{P}[D]$ are finitely generable. Conversely suppose $X = Bd(f)$ is a finitely generable subset of D. Let T be the finitary tree consisting of these finite elements, ω, of Ω such that $f(\omega) = \perp$. If T is finite, then X is finite. If T is infinite, it has an infinite branch by König's lemma, and so $\perp \in X$. In either case X is in $\mathcal{P}[D]$. When D is N^ω, the output sets considered above are finitely generable as can be seen from the connection with oracles.

The definition does not lose any sets by considering only Ω. It can be shown without much difficulty that if T is any finitary tree with limit points added, and $f : T \to D$, then $Bd(f) =_{\text{def}} \{f(\omega) | \omega \text{ a maximal member of } T\}$ is finitely generable. It follows that every finite nonempty subset of any ipo D is finitely generable, as is every denumerable nonempty subset containing a lower bound of itself.

Notice that if $X \subseteq D$ is finitely generable and $g : D \to E$, then $g(X)$ is also finitely generable. That is, the definition is consistent with that of $\mathcal{P}[E]$ for ω-discrete E in that, starting with finitely generable sets and applying suitable continuous functions, only members of $\mathcal{P}[E]$ are reached. Unfortunately, that does not determine the finitely generable sets. For example, if $N_\top = \{\perp, \top, 0, 1, \cdots\}$ is the lattice formed from N by adding $\perp$ and $\top$, then $X = \{\top, 0, 1, \cdots\}$ is not finitely generable, but if E is ω-discrete and $f : N_\top \to E$, then $f(X)$ is finite.

It is not unreasonable therefore to include only the finitely generable sets in the domain $\mathscr{P}[D]$. However there are reasons for considering other sets. Consider the program:

$$P = ((y := 0); (x := 0)); ((y := 1) \textit{ par } (\textit{while } y \textit{ do } x := x + 1)).$$

According to the parallel construct sketched above, P can either stop with x set to an arbitrary integer or fail to terminate. Now postulate a *fair* parallel construct which at any point will never restrict all its computation to one branch, unless the other has terminated. Clearly, every fair computation sequence of P terminates, but the set of results of the fair computation sequences is not finitely generable, being $\{\langle 0, 1\rangle, \langle 1, 1\rangle, \langle 2, 1\rangle, \cdots\}$. It may be possible to handle this by considering the *denumerably generable* sets instead of the finitely generable ones. These are obtained by replacing Ω in the definition by N^{ω}. In the ω-discrete case this leads to the alternative mentioned above. Another approach is not to consider *all* fair computation sequences, but rather parallel constructs which give rise to finitely generable sets of fair computation sequences. The importance of the problem is that the assumption of fairness is needed to prove the absence of deadlock of certain co-operating processes [5]. This poses one of the more interesting problems left unresolved in this paper.

At any rate we will form $\mathscr{P}[D]$ from $\mathscr{F}[D]$ the collection of finitely generable subsets of D, and an as yet undetermined ordering, $\sqsubseteq$. Since $\sqsubseteq$ will be only a quasi-ordering in general, $\mathscr{P}[D]$ will actually be the collection of equivalence classes with the induced partial ordering.

It remains to define this ordering. In the cases of $+$, $\times$ and $\to$ not only is the induced ordering natural, it is also, in a sense, necessary. For example in the case of $D_1 \times D_2$ one expects that the projection functions $\pi_i : D_1 \times D_2 \to D_i$ $(i = 1, 2)$ are monotonic. So if $\langle d_1, d_2\rangle \sqsubseteq \langle e_1, e_2\rangle$, then $d_i = \pi_i(\langle d_1, d_2\rangle) \sqsubseteq \pi_i(\langle e_1, e_2\rangle) = e_i$ $(i = 1, 2)$.

Notation. Let D and E be ipo's and suppose E is ω-discrete. If $f : D \to \mathscr{P}[E]$, then $\tilde{f} : \mathscr{F}[D] \to \mathscr{P}[E]$ is defined by:

$$\tilde{f}(X) = \bigcup \{f(x) | x \in X\} \qquad (X \in \mathscr{F}[D]).$$

We omit the justification of the definition of $\tilde{f}$ for the moment.

DEFINITION. Let D be an ipo. The quasi-order $\sqsubseteq$ on $\mathscr{F}[D]$ is defined by:

$$X \sqsubseteq Y \text{ iff } (\forall \omega\text{-discrete } E. \forall f : D \to \mathscr{P}[E]. \tilde{f}(X) \sqsubseteq_M \tilde{f}(Y)), \qquad (X, Y \in \mathscr{F}[D]).$$

In the case of $\mathscr{P}[D]$ we would expect that if $f : D \to \mathscr{P}[E]$, where E is ω-discrete, then $\tilde{f} : \mathscr{F}[D] \to \mathscr{P}[E]$ would be monotonic.

So if $\sqsubseteq$ is our hypothetical ordering we expect that, for X, Y in $\mathscr{F}[D]$,

(1) $\qquad (X \sqsubseteq_M Y \text{ implies } X \sqsubseteq Y) \quad \text{and} \quad (X \sqsubseteq Y \text{ implies } X \sqsubseteq Y).$

The first part arises from the naturalness of the elementwise criterion. The second part arises from the above discussion and our expectation that in the case of ω-discrete domains, $\sqsubseteq$ will be $\sqsubseteq_M$.

We pause here to verify that $\tilde{f}$ is indeed well-defined. That is if X is finitely generable, then $\tilde{f}(X)$ is in $\mathscr{P}[D]$.

Suppose $X = Bd(g)$ for some $g : \Omega \to D$. Let T be the finitary tree consisting of those finite ω in Ω such that $\perp \in f(g(\omega))$. If T is finite, then $\tilde{f}(X)$ is finite. If T is infinite then, by König's lemma T has an infinite branch $\omega_0, \omega_1, \omega_2, \cdots$. Since $\perp \in f(g(\omega_i))$ for all $i \geqq 0$, it follows that $\perp \in \bigsqcup_{i \geqq 0} f(\omega_i) = f(\bigsqcup_{i \geqq 0} \omega_i) \subseteq \tilde{f}(X)$. As $\tilde{f}(X) \neq \emptyset$, it follows that in either case $\tilde{f}(X) \in \mathscr{P}[D]$.

In some cases the second part of (1) can be computationally justified. If E is the output domain, in some programming language, and D is part of a value domain, then a set $X \subseteq D$ might arise as the set of possible values of a variable at a stage in the computation. With everything else fixed, we get a continuous function $f : D \to \mathscr{P}[E]$ where $f(d)$ is the set of possible outputs arising from $d \in X$. Then $\tilde{f}(X)$ is interpreted as the set of all possible outputs, which should vary monotonically with X.

Here are some elementary facts about $\sqsubseteq$:

THEOREM 2. (i) *For X, Y in $\mathscr{F}[D]$, $X \sqsubseteq Y$ iff $\forall f : D \to \mathbb{O}. f(X) \sqsubseteq_M f(Y)$.*

(ii) *If $X \sqsubseteq Y$ and $f : D \to E$, then $f(X) \sqsubseteq f(Y)$ (X, Y in $\mathscr{F}[D]$).*

(iii) *If $X \sqsubseteq_M Y$, then $X \sqsubseteq Y$ (X, Y in $\mathscr{F}[D]$).*

(iv) *If D is ω-discrete and $X \sqsubseteq Y$, then $X \sqsubseteq_M Y$ (X, Y in $\mathscr{F}[D]$).*

Proof. (i) Let X, Y be in $\mathscr{F}[D]$. Suppose $f : D \to \mathbb{O}$. Define $g : D \to \mathscr{P}[\mathbb{O}]$ by $g(d) = \{f(d)\}$. Then if $X \sqsubseteq Y$, $\tilde{g}(X) \sqsubseteq_M \tilde{g}(Y)$, so $f(X) \sqsubseteq_M f(Y)$.

Conversely suppose $f(X) \sqsubseteq_M f(Y)$ for all $f : D \to \mathbb{O}$ and $g : D \to \mathscr{P}[E]$ where E is ω-discrete. Suppose e is in $\tilde{g}(X)$ and $e \neq \perp$. Define $h : \mathscr{P}[E] \to \mathbb{O}$ by:

$$h(Z) = \begin{cases} \perp & (e \notin Z), \\ \top & (e \in Z), \end{cases} \quad (Z \text{ in } \mathscr{P}[E]).$$

This does define a continuous function. Taking $f = h \circ g$, we find that $\top \in f(X) \sqsubseteq_M f(Y) = h(g(Y))$. Therefore e is in $\tilde{g}(Y)$ which is half the proof that $\tilde{g}(X) \sqsubseteq_M \tilde{g}(Y)$. If $\perp \in \tilde{g}(X)$, we are finished. Otherwise $\tilde{g}(X)$ is finite since it is in $\mathscr{P}[E]$ and so for each x in X, $g(x)$ is finite and $\perp \notin g(x)$.

Define $h : \mathscr{P}[E] \to \mathbb{O}$ by:

$$h(Z) = \begin{cases} \top & (Z = g(x) \text{ for some } x \in X), \\ \perp & (\text{otherwise}), \end{cases} \quad (Z \text{ in } \mathscr{P}[E]).$$

Then h is continuous and with $f = h \circ g$, $\{\top\} = f(X) \sqsubseteq_M f(Y) = h(g(Y))$. Thus $\tilde{g}(Y) \subseteq \tilde{g}(X)$ which concludes the proof of part (i).

(ii) Suppose $X \sqsubseteq Y$ for X, Y in $\mathscr{F}[D]$ and $f : D \to E$. If $g : E \to \mathbb{O}$, then $g(f(X)) \sqsubseteq_M g(f(Y))$ by part (i). So $f(X) \sqsubseteq f(Y)$ by part (i).

(iii) Suppose $X \sqsubseteq_M Y$ for X, Y in $\mathscr{F}[D]$. If $f : D \to E$, then $f(X) \sqsubseteq_M f(Y)$. Therefore $X \sqsubseteq Y$ by part (i).

(iv) Suppose D is ω-discrete and $X \sqsubseteq Y$ for X, Y in $\mathscr{F}[D]$. Define $f : D \to \mathscr{P}[D]$ by: $f(d) = \{d\}$. Then

$$\begin{aligned} X &= \tilde{f}(X) \\ &\sqsubseteq_M \tilde{f}(Y) \quad (\text{by the definition of } \sqsubseteq) \\ &= Y. \end{aligned}$$

□

Unfortunately (1) does not determine $\sqsubseteq$. For example let $X_0 = \{\perp\} \cup \{0^n 1^\omega | n \geqq 0\}$, $X_1 = X_0 \cup \{0^\omega\}$, using an obvious notation, define finitely generable subsets of Ω. We claim that $X_0 \simeq X_1$ where $\simeq$ is the equivalence induced by $\sqsubseteq$. But clearly, $X_0 \neq_M X_1$. So $X \sqsubseteq Y$ does not imply $X \sqsubseteq_M Y$, in general.

To see that $X_0 \simeq X_1$, let $f: D \to \mathbb{O}$. Clearly $f(X_0) \subseteq f(X_1)$. If $\perp \in f(X_1)$, then $\perp = f(\perp) \in f(X_0)$; if $\top \in f(X_1)$, then $\top = f(d)$ for some d in X_1. If $d \in X_0$, $\top \in f(X_0)$. Otherwise, $d = 0^\omega$. Then for some n, $\top = f(0^n)$. Therefore $\top \in f(X_0)$ in this case too. Therefore $f(X_0) = f(X_1)$ and so, by the theorem, $X_0 \simeq X_1$.

So our analysis has left us in rather a quandary as to the definition of $\sqsubseteq$ on $\mathscr{F}[D]$. One expedient would be to search for other necessary conditions on $\sqsubseteq$ by looking for functions other than those of the form $\tilde{f}$. These might be provided by considering programming features which allow some inspection of all possible execution sequences of subprograms. We do not consider such "AND" programming here (but see Manna [9]).

Since we would like our semantics to be fully abstract, we choose the coarser relation $\sqsubseteq$. By Theorem 2(iv) this is consistent with our earlier definition of $\mathscr{P}[D]$ for D ω-discrete. We conjecture that choosing $\sqsubseteq_M$ instead would not even give an ipo, but lack a counter-example. We were not able to develop a satisfactory independently justified definition of $\sqsubseteq$ for sequence domains like Ω or N^ω. However $\sqsubseteq$ does have some pleasant properties for these domains. Suppose X, Y are two members of $\mathscr{F}[D]$, where D is such a sequence domain. If every member of Y is infinite then $X \sqsubseteq Y$ iff $X \sqsubseteq_M Y$; if every member of X is infinite, $X \sqsubseteq Y$ iff X and Y are identical. The second claim follows from the first which will be proved later.

We now have to within isomorphism a definition of $\mathscr{P}$:

DEFINITION. $\mathscr{P}[D] \cong \langle \mathscr{F}[D]/\simeq, \sqsubseteq/\simeq \rangle$ where $\mathscr{F}[D]/\simeq$ is the set of finitely generable subsets of D modulo $\simeq$, the equivalence induced by the quasi-order $\sqsubseteq$, and $\sqsubseteq/\simeq$ is the induced partial order.

When D is ω-discrete we retain the earlier definition as the standard one; later we will fix on a standard one for other cases.

As yet we have no guarantee that $\mathscr{P}[D]$ is an ipo when D is not discrete, let alone that we may solve recursive domain equations using $\mathscr{P}$. We need a suitable class of ipo's in which such guarantees can be obtained. We begin by showing that $\mathscr{P}[D]$ is a well-defined ipo if D is finite. That is carried out in § 3. By considering limits of infinite sequences of finite ipo's we arrive at a category SFP. This category is described in § 4. It turns out that if D is an SFP object so is $\mathscr{P}[D]$. Indeed $\mathscr{P}$ is a functor on SFP. We also obtain good internal descriptions of $\mathscr{P}[D]$. This work occupies § 5 and part of § 6. In § 6 we show that various associated functions are continuous. These functions are useful for defining various denotational semantics. Finally in § 7 we can show that a wide variety of recursive domain equations are solvable in SFP. Our main method is categorical, and we also have a universal domain method. It is possible to use Scott's $\mathscr{P}(\omega)$, [17], but not in a particularly satisfactory way. The last section, § 8, applies this work by giving some illustrative semantics as mentioned above.

3. Finite powerdomains. If D is finite, then $\mathscr{P}[D]$ as described in the last section is certainly a partial order. It has a least element which is the equivalence

class of $\{\perp\}$ under $\simeq$. Since D is finite it follows that $\mathscr{P}[D]$ is indeed an ipo. However we really want a more pleasant description of $\mathscr{P}[D]$. This is given by the next theorem.

DEFINITION. Let E be an ipo. A subset X of E is *convex* iff $(\forall x, y, z \in E.$ $(x \sqsubseteq y \sqsubseteq z$ and $x \in X$ and $z \in X)$ implies $y \in X)$.

The convex closure operator, Con, is defined on subsets of E by, $\mathrm{Con}(X) =_{\mathrm{def}} \{y \in E | \exists x, z \in X. x \sqsubseteq y \sqsubseteq z\}$ $(X \subseteq E)$.

THEOREM 3. (i) Con *is a closure operator on any* ipo E. *For any* $X, Y \subseteq E$, $\mathrm{Con}(X)$ *is the least convex set containing* X, $X =_M \mathrm{Con}(X)$, *and* $X =_M Y$ *iff* $\mathrm{Con}(X) = \mathrm{Con}(Y)$.

(ii) *If* D *is finite,* $X \sqsubseteq Y$ *iff* $X \sqsubseteq_M Y$ $(X, Y \in \mathscr{F}[D])$.

(iii) *If* D *is finite,* $\mathscr{P}[D] \cong \langle \{\mathrm{Con}(X) | X \subseteq D, X \neq \varnothing\}, \sqsubseteq_M \rangle$.

Proof. (i) The proof is straightforward.

(ii) By Theorem 2(iii) we know that $X \sqsubseteq_M Y$ implies $X \sqsubseteq Y$. Suppose $X \sqsubseteq Y$ and $x \in X$. Define $f: D \to \Phi$ by:

$$f(d) = \begin{cases} \top, & (d \sqsupseteq x), \\ \perp, & (d \not\sqsupseteq x). \end{cases}$$

Then f is continuous. As $\top \in f(X) \sqsubseteq_M f(Y)$ by Theorem 2(i), $y \sqsupseteq x$ for some y in Y.

Suppose $y \in Y$. Now define $f: D \to \mathbb{O}$ by

$$f(d) = \begin{cases} \perp, & (d \sqsubseteq y), \\ \top, & (d \not\sqsubseteq y). \end{cases}$$

Again f is continuous and now $\perp \in f(Y) \sqsupseteq_M f(X)$. So $x \sqsubseteq y$ for some x in X. Therefore $X \sqsubseteq_M Y$ as required.

(iii) By definition, $\mathscr{P}[D] \cong \langle \mathscr{F}[D]/\simeq, \sqsubseteq/\simeq \rangle$. Clearly the finitely generable subsets of D are the nonempty ones. By (i) and (ii), Con assigns to each X in $\mathscr{F}[D]$ the maximal member of its equivalence class under $\simeq$. It therefore induces a 1–1 correspondence between $\mathscr{F}[D]/\simeq$ and $\{\mathrm{Con}(X) | X \subseteq D, X \neq \varnothing\}$ which is an isomorphism of the partial orders. □

From now on we will take $\langle \{\mathrm{Con}(X) | X \subseteq D, X \neq \varnothing\}, \sqsubseteq_M \rangle$ as our standard definition of $\mathscr{P}[D]$ when D is finite. This is consistent with our previous standard definition for the discrete case.

Let us consider the extension of functions to finite powerdomains. Suppose $f: D \to E$ for finite ipo's D and E. Define $\hat{f}: \mathscr{P}[D] \to \mathscr{P}[E]$ by:

$$\hat{f}(X) = \mathrm{Con}(f(X)), \qquad (X \in \mathscr{P}[D]).$$

The function $\hat{f}$ is monotonic and therefore continuous; for, if $X \sqsubseteq_M Y$, $\hat{f}(X) =_M f(X) \sqsubseteq_M f(Y) =_M \hat{f}(Y)$ by Theorem 2.2. Note that $\hat{I}_D$ is $I_{\mathscr{P}[D]}$ where I_D and $I_{\mathscr{P}[D]}$ are the identities on D and $\mathscr{P}[D]$, respectively. Finally, extension commutes with composition; for if D, E, F are finite ipo's, $f: D \to E$ and $g: E \to F$, then for $X \in \mathscr{P}[D]$, $\hat{g}(X) =_M g(X)$, so $\hat{f}(\hat{g}(X)) =_M \hat{f}(g(X)) = \mathrm{Con}(f(g(X))) = \widehat{f \circ g}(X)$.

The finite ipo's allow us to show that even if D is a lattice, $\mathcal{P}[D]$ need not be. Let $\mathbb{T}_\top = \{\bot, \text{true}, \text{false}, \top\}$ be the lattice of truth values and take $D = \mathbb{T}_\top \times \mathbb{T}_\top$. Let $a = \langle \textit{true}, \bot\rangle$, $b = \langle \textit{false}, \bot\rangle$, $c = \langle \bot, \textit{true}\rangle$ and $d = \langle \bot, \textit{false}\rangle$. Let $A = \{a, b\}$, $B = \{c, d\}$, $C = \{a \sqcup c, c \sqcup d\}$ and $D = \{a \sqcup d, b \sqcup c\}$. Then one can check that C and D are incomparable minimal upper bounds of A and B in $\mathcal{P}[D]$. Therefore $\mathcal{P}[D]$ is not a lattice. Indeed it is not even a semilattice, which we take to be a closed subset of a lattice, as in [17]. So converting $\mathcal{P}[D]$ into a lattice would require one to add many points—not just a top element. It is not clear to the author how to keep these separate from the bona fide elements.

4. The category SFP. In this section we present the class of domains over which our powerdomain construction works. They are certain limits in the category IPO. They will form a category SFP and $\mathcal{P}$ will be a functor from SFP to SFP. An alternate characterization of the SFP objects in terms of their order structure will provide a priori reasons for their usefulness.

Perhaps we could comment on our use of category theory. No deep theorems of category theory are used. Rather, it allows a systematic development of the material. We cannot give such a development entirely within Scott's $\mathcal{P}(\omega)$, [17].

We begin by considering the relevant limits in IPO.

DEFINITION. IPO-P is the category whose objects are the ipo's and whose morphisms $p : D \to E$ are pairs $\langle \varphi, \psi\rangle$ where $\varphi : D \to E$ and $\psi : E \to D$. Composition is defined by:

$$q \circ p = \langle (q)_1 \circ (p)_1, (p)_2 \circ (q)_2\rangle.$$

The identity on D is $\langle I_D, I_D\rangle$ where $I_D : D \to D$ is the usual identity. It will also be called I_D and we rely on context to settle the ambiguity. Composition is continuous with respect to the induced componentwise ordering on morphisms.

The category IPO-P is useful because it has the interesting subcategory, IPO-PR. If $p : D \to E$ let $p^\dagger = \langle p_2, p_1\rangle$. We have the law, $(q \circ p)^\dagger = p^\dagger \circ q^\dagger$, and also $I_D^\dagger = I_D$. A pair $p : D \to E$ is a *projection* (of E onto D) iff $p^\dagger \circ p = I_D$ and $p \circ p^\dagger \sqsubseteq I_E$, under the induced componentwise ordering. This agrees with the usual definition of projection. The composition of two projections is a projection as are the identities. Note that the set of projections between two ipo's forms an ipo under the induced ordering.

DEFINITION IPO-PR is the subcategory of IPO-P with the same objects and with projections as morphisms.

If D and E are isomorphic in IPO-PR, then they are isomorphic in IPO, that is, as ipo's.

The use of these derived categories is suggested by the work of Reynolds [14], [15] and Wand [20], [21].

The category IPO-PR has direct limits of sequences. We give a definition which is, essentially, taken from [6].

DEFINITION. A *directed sequence* (in IPO-PR) is a family $\mathcal{D} = \langle D_m, p_{mn}\rangle$ of ipo's D_m $(m \geqq 0)$ together with projections $p_{mn} : D_m \to D_n$ $(0 \leqq m \leqq n)$ such that $p_{mm} = I_{D_m}$ and $p_{mn} \circ p_{lm} = p_{ln}$ when $0 \leqq l \leqq m \leqq n$.

DEFINITION. A *cone* from a directed sequence $\mathscr{D} = \langle D_m, p_{mn}\rangle$ to an ipo D is a family $\langle r_m\rangle$ of projections $r_m : D_m \to D$ such that:

$$r_m \circ p_{lm} = r_l \qquad (0 \leqq l \leqq m).$$

Such a cone is *universal* iff whenever $\langle r'_m\rangle$ is a cone from $\mathscr{D}$ to an ipo D' then there is a unique mediating projection $r' : D \to D'$ such that

$$r'_m = r' \circ r_m \qquad (m \geqq 0).$$

In this case, D is a *direct limit* of D and we write: $D = \lim_{\to} \mathscr{D}$.

It follows from the uniqueness of the mediating projections that a direct limit is unique, up to isomorphism, and the mediating projection from one direct limit to another is an isomorphism.

LEMMA 1. *Let $\mathscr{D} = \langle D_m, p_{mn}\rangle$ be a directed sequence and let $\langle r_m\rangle$ and $\langle r'_m\rangle$ be cones from $\mathscr{D}$ to D and D', respectively. Then $\langle r'_m \circ r_m^{\dagger}\rangle$ is an increasing sequence. The cone $\langle r_m\rangle$ is universal iff $\bigsqcup_{n\geqq 0} r_n \circ r_n^{\dagger} = I_D$. If $\langle r_m\rangle$ is universal, then the unique mediating morphism $r' : D \to D'$ is $r' = \bigsqcup_{n\geqq o} r'_n \circ r_n^{\dagger}$.*

Proof. Suppose $n \geqq m$. Then $r'_m \circ r_m^{\dagger} = (r'_n \circ p_{mn}) \circ (r_n \circ p_{mn})^{\dagger} = r'_n \circ p_{mn} \circ p_{mn}^{\dagger} \circ r_n \sqsubseteq r'_n \circ I_{D_m} \circ r_n = r'_n \circ r_n$. Therefore $\langle r'_m \circ r_m^{\dagger}\rangle$ is an increasing sequence.

Suppose $\langle r_m\rangle$ is universal. Since $\langle r_m\rangle$ is a cone from $\mathscr{D}$ to D there is a unique mediating projection $r : D \to D$ such that $r_m = r \circ r_m$ $(m \geqq 0)$. Since I_D is a mediating projection, $r = I_D$. We show that $\bigsqcup_{n\geqq 0} r_n \circ r_n^{\dagger}$ is also a mediating projection. It then follows that it too is I_D.

Now, $(\bigsqcup_{n\geqq 0} r_n \circ r_n^{\dagger}) \circ r_m = \bigsqcup_{n\geqq m} r_n \circ r_n^{\dagger} \circ (r_n \circ p_{mn}) = \bigsqcup_{n\geqq m} r_n \circ p_{mn} = r_m$. Therefore $\bigsqcup_{n\geqq 0} r_n \circ r_n^{\dagger}$ is indeed the mediating projection I_D.

Conversely, suppose $\bigsqcup_{n\geqq 0} r_n \circ r_n^{\dagger} = I_D$ and let $r' : D \to D'$ be $\bigsqcup_{n\geqq 0} r'_n \circ r_n^{\dagger}$. We show that r' is a mediating projection from D to D' and that if r'' is any mediating projection, then $r'' = r'$.

First,

$$\begin{aligned} r' \circ r_m = \left(\bigsqcup_{n\geqq 0} r'_n \circ r_n^{\dagger}\right) \circ r_m &= \bigsqcup_{n\geqq m} r'_n \circ (r_n^{\dagger} \circ r_n \circ p_{mn}) \\ &= \bigsqcup_{m\geqq n} r'_n \circ p_{mn} \\ &= r'_m \qquad (m \geqq 0). \end{aligned}$$

Therefore r' is a mediating projection from D to D'.

Next, suppose r'' is such a mediating projection. Then,

$$\begin{aligned} r'' = r'' \circ \left(\bigsqcup_{n\geqq 0} r_n \circ r_n^{\dagger}\right) &= \bigsqcup_{n\geqq 0} (r'' \circ r_n) \circ r_n^{\dagger} \\ &= \bigsqcup_{n\geqq 0} r'_n \circ r_n^{\dagger} \\ &= r'. \end{aligned}$$

□

We can now show that IPO-PR has direct limits of sequences.

THEOREM 4. *Let* $\mathcal{D} = \langle D_m, p_{mn}\rangle$ *be a directed sequence where* $p_{mn} = \langle \varphi_{mn}, \psi_{nm}\rangle$. *Let* D_∞ *be the set of vectors,* $\{\langle d_m\rangle | m \geqq 0$ *and* $d_m = \psi_{nm}(d_m)$ *if* $0 \leqq m \leqq n\}$ *with the pointwise ordering:* $d \sqsubseteq e$ *iff* $\forall m \geqq 0.(d)_m \sqsubseteq (e)_m$ $(d, e \in D)$. *Then* D *is an ipo. Define* $i_m : D_m \to D$, $j_m : D \to D_m$ $(m \geqq 0)$ *by:*

$$(i_m(d_m))_n = \begin{cases} \varphi_{mn}(d_m), & (n \geqq m), \\ \psi_{mn}(d_m), & (n < m), \end{cases}$$
$$i_m(d) = (d)_m.$$

Then i_m *and* j_m *are indeed continuous and* $r_m = \langle i_m, j_m\rangle$ *is a projection. Further,* $\langle r_m\rangle$ *is a universal cone and so* $D = \lim_{\to} \mathcal{D}$.

Proof. The proof is a straightforward point-by-point verification. Universality is proved using the criterion given by Lemma 1. We omit the details which can be taken over from the usual proofs for complete lattices, as given in [16], [14], [21]. □

DEFINITION. The category SFP (Sequences of Finite inductive Partial orders) has as objects those ipo's $D = \lim_{\to} \mathcal{D}$, where $\mathcal{D} = \langle D_m, p_{mn}\rangle$ is a directed sequence of finite ipo's D_m. Its morphisms $f : D \to E$ are the continuous functions with the usual composition.

Every finite ipo is an SFP object as it is a trivial limit of finite ipo's.

The categories SFP-P and SFP-PR are defined analogously to IPO-P and IPO-PR and are actually full subcategories of them. In other words if D and E are SFP-P (SFP-PR) objects, then the set of morphisms from D to E is the same in SFP-P (SFP-PR) as it is in IPO-P (IPO-PR). The notions of directed sequence, cone, universality and limit in SFP-PR are defined analogously to the corresponding notions in IPO-PR.

We now turn to an alternate characterization of SFP objects.

DEFINITION. An element d in an ipo D is *finite* iff whenever $X \subseteq D$ is directed and $d \sqsubseteq \bigsqcup X$, then $d \sqsubseteq x$ for some x in X.

DEFINITION. An ipo D is *algebraic* iff for any x in D the set $\{d | d \sqsubseteq x$ and d is finite$\}$ is directed and has least upper bound x. D is *ω-algebraic* iff it is algebraic and has denumerably many finite elements.

DEFINITION. Let X be a subset of an ipo D. An element u of D is a *minimal upper bound* of X iff it is an upper bound of X and it is not strictly greater than any other upper bound of X; $\mathcal{U}(X)$ is defined to be the set of all minimal upper bounds of X. $\mathcal{U}(X)$ is a *complete* set of upper bounds of X iff whenever u is an upper bound of X, then $u \sqsupseteq v$ for some v in $\mathcal{U}(X)$. $\mathcal{U}^*(X)$ is defined to be the least set containing X and closed under $\mathcal{U}$.

If an ipo D is algebraic and X is a finite set of finite elements, then every element of $\mathcal{U}(X)$ is finite. For suppose $u \in \mathcal{U}(X)$. Since X is a subset of $\{d | d \sqsubseteq u$ and d finite$\}$ and that set is directed there is an upper bound, u', of X in the set. Since u is a minimal upper bound of X it must be u' and so u is finite as u' is.

THEOREM 5. (i) *An* ipo D *is an* SFP*-object iff it is ω-algebraic, and, whenever X is a finite set of finite elements of X, then $\mathcal{U}(X)$ is a complete set of upper bounds of X and $\mathcal{U}^*(X)$ is finite.*

(ii) *The category* SFP-PR *has direct limits of directed sequences.*

Proof. (i) Every finite ipo clearly satisfies the conditions. We show that they are preserved under direct limits of directed sequences in IPO-PR. Suppose $\mathscr{D} = \langle D_m, p_{mn}\rangle$ is a directed sequence in IPO-PR, and the D_m satisfy the conditions. Let D_∞ and $r_m = \langle i_m, j_m\rangle$ be as described in Theorem 4.

First we show that D_∞ is ω-algebraic. Each element of the form $i_m(d_m)$, where d_m is a finite element of D_m, is finite. For if $X \subseteq D$ is directed and $i_m(d_m) \sqsubseteq \bigsqcup X$, then $d_m = j_m \circ i_m(d_m) \sqsubseteq \bigsqcup_{x\in X} j_m(x)$. So as d_m is finite there is an $x \in X$ such that $d_m \sqsubseteq j_m(x)$. Then $i_m(d_m) \sqsubseteq i_m \circ j_m(x) \sqsubseteq x$. So $i_m(d_m)$ is indeed finite.

Further every element in D_∞ is a least upper bound of a directed set of such elements. For if $d \in D_\infty$, then

$$\begin{aligned} d &= \bigsqcup_{m\geqq 0} i_m(j_m(d)) \\ &= \bigsqcup_{m\geqq 0}{}^{\cdot}\, i_m(\bigsqcup\{d_m \in D_m \mid d_m \text{ finite and } d_m \sqsubseteq j_m(d)\}) \\ &\qquad\qquad (\text{as } D_m \text{ is algebraic}) \\ &= \bigsqcup\{i_m(d_m) \mid m \geqq 0,\ d_m \in D_m,\ d_m \text{ finite and } d_m \sqsubseteq j_m(d)\}, \end{aligned}$$

and the set on the right is a directed set of finite elements of D_∞.

So D_∞ must be algebraic and its finite elements are those of the form $i_m(d_m)$ where $m \geqq 0$ and d_m is a finite member of D_m. Since each D_m is ω-algebraic, there are denumerably many such elements and so D_∞ is also ω-algebraic.

We now consider the operation $\mathscr{U}$. Let it be $\mathscr{U}_m$ in D_m ($m \geqq 0$) and $\mathscr{U}_\infty$ in D_∞. Then if $X_m \subseteq D_m$, $\mathscr{U}_\infty(i_m(X_m)) = i_m(\mathscr{U}_m(X_m))$. (We are using a notation for function application which was defined in § 2.) For it is straightforward to check that if u is a minimal upper bound of $i_m(X_m)$, then so is $j_m(u)$ of X_m and $u = i_m \circ j_m(u)$. Conversely if u is a minimal upper bound of X_m, then so is $i_m(u)$ of $i_m(X_m)$.

Now suppose $X \subseteq D$ is a finite set of finite elements of D. Then $X = i_m(X_m)$ for some finite set of finite elements of some D_m. Therefore, $\mathscr{U}_\infty(X) = i_m(\mathscr{U}_m(X_m))$. If u in D is an upper bound of X, then $j_m(u)$ is an upper bound of X_m. Therefore there is a v in $\mathscr{U}_m(X_m)$ such that $v \sqsubseteq j_m(u)$. Then $i_m(v) \in \mathscr{U}_\infty(X)$ and $i_m(v) \sqsubseteq u$. So $\mathscr{U}_\infty(X)$ is complete.

Next we show that $\mathscr{U}^*_\infty(X)$ is finite. First we define $\mathscr{U}^r$ for any ipo E and $r \geqq 0$:

$$\begin{aligned} \mathscr{U}^0(Y) &= \varnothing, \\ \mathscr{U}^{r+1}(Y) &= \cup\{\mathscr{U}(Y') \mid Y' \subseteq \mathscr{U}^r(Y)\} \cup Y \qquad (Y \subseteq E). \end{aligned}$$

Then for $Y \subseteq E$ we have $\mathscr{U}^r(Y) \subseteq \mathscr{U}^{r+1}(Y) \subseteq \mathscr{U}^*(Y)$ ($r \geqq 0$). If $\mathscr{U}^r(Y) = \mathscr{U}^{r+1}(Y)$ for some $r \geqq 0$, then $\mathscr{U}^*(Y) = \mathscr{U}^r(Y)$.

We now show by induction on r that $\mathscr{U}^r_\infty(i_m(Y)) = i_m(\mathscr{U}^r_m(Y))$ for all $Y \subseteq D_m$. It is clear for $r = 0$. For $r+1$ we have,

$$\begin{aligned} \mathscr{U}^{r+1}_\infty(i_m(Y)) &= \cup\{\mathscr{U}_\infty(Y') \mid Y' \subseteq \mathscr{U}^r_\infty(i_m(Y))\} \cup i_m(Y) \\ &= \cup\{\mathscr{U}_\infty(Y') \mid Y' \subseteq i_m(\mathscr{U}^r_m(Y))\} \cup i_m(Y) \\ &\qquad\qquad (\text{by induction hypothesis}) \end{aligned}$$

$$= \cup\{\mathcal{U}_\infty(i_m(Y'))\,|\,Y' \subseteq \mathcal{U}_m^r(Y)\} \cup i_m(Y)$$
$$= \cup\{i_m(\mathcal{U}_m(Y'))\,|\,Y' \subseteq \mathcal{U}_m^r(Y)\} \cup i_m(Y)$$

(by a previous remark)

$$= i_m(\mathcal{U}_m^{r+1}(Y)) \qquad (Y \subseteq D_m).$$

Now, since $\mathcal{U}_m^*(X_m)$ is finite there is an r such that $\mathcal{U}_m^r(X_m) = \mathcal{U}_m^{r+1}(X_m) = \mathcal{U}_m^*(X_m)$. Therefore $\mathcal{U}_\infty^r(X) = i_m(\mathcal{U}_m^r(X_m)) = i_m(\mathcal{U}_m^{r+1}(X_m)) = \mathcal{U}_\infty^{r+1}(X)$. Therefore $\mathcal{U}_\infty^*(X) = \mathcal{U}_\infty^r(X) = i_m(\mathcal{U}_m^*(X_m))$, by the above remarks, and so $\mathcal{U}_\infty^*(X)$ is finite. We have therefore shown that D satisfies all the conditions.

We now show that if D satisfies all the conditions, then it is the limit of a directed sequence of finite ipo's.

Let D have finite elements, $\perp = e_0, e_1, \cdots$.

Let $D_m = \langle \mathcal{U}_D^*(\{e_0, e_1, \cdots, e_m\}), \sqsubseteq\rangle$ where $\sqsubseteq$ is inherited from D $(m \geqq 0)$. Then D_m is a finite ipo. Define $p_{mn} : D_m \to D_n$ $(0 \leqq m \leqq n)$ by:

$$\varphi_{mn}(d_m) = d_m (d_m \in D_m),$$
$$\psi_{nm}(d_n) = \sqcup\{x \in D_m | x \sqsubseteq d_n\} \qquad (d_n \in D_n),$$
$$p_{nm} = \langle \varphi_{mn}, \psi_{nm}\rangle \qquad (0 \leqq m \leqq n).$$

Then ψ_{nm} is well-defined as the set on the RHS (right-hand side) is directed as it contains $\perp$ and by the properties of $\mathcal{U}$. Both φ_{mn} and ψ_{nm} are monotonic and therefore continuous. One can check that each p_{mn} is a projection and $\langle D_m, p_{mn}\rangle$ is a directed sequence. Define $r_m : D_m \to D$ by:

$$i_m(d_m) = d_m \qquad (d_m \in D_m),$$
$$j_m(d) = \sqcup\{x \in D_m | x \sqsubseteq d\} \qquad (d \in D, m \geqq 0),$$
$$r_m = \langle i_m, j_m\rangle.$$

As before j_m is well-defined and is monotonic and therefore continuous. Clearly i_m is well-defined and continuous. One can check that $\langle r_m\rangle$ is a cone from $\mathcal{D} = \langle D_m, p_{mn}\rangle$ to D.

Further,

$$\bigsqcup_{m \geqq 0} i_m \circ j_m(d) = \bigsqcup_{m \geqq 0} \sqcup\{x \in D_m | x \sqsubseteq d\} = d \qquad (d \in D).$$

Therefore by Lemma 1, $D = \lim_{\to} \mathcal{D}$, as required.

(ii) Suppose $\mathcal{D}$ is a directed sequence in SFP-PR. Then it is also a directed sequence in IPO-PR. It therefore has a direct limit, D, in IPO-PR. The argument in part (i) of the proof shows that D is actually an SFP object. As SFP-PR is a full subcategory of IPO-PR, D is also the direct limit in SFP-PR of $\mathcal{D}$. □

The internal characterization of SFP objects given by Theorem 5 allows us to argue for the reasonableness of the category SFP. First, all denotational semantics for programming languages, published till now, use ω-algebraic ipo's. The example given in § 3 shows that we should probably not expect consistent completeness. Our axioms for $\mathcal{U}$ give a kind of substitute.

It may be that at some time we shall want a concept of continuous ipo and a powerdomain construction for it. Universal domains provide a candidate construction, as will be seen. However the author does not know a good definition of the notion of a continuous ipo. Finally, considering ω-algebraic ipo's rather than just algebraic ones provides a framework for computability considerations.

5. Powerdomains in the category SFP. In this section we obtain sufficient conditions for $\mathcal{P}[D]$ to be an ipo. Indeed we show that if D is an SFP object, so is $\mathcal{P}[D]$. As a byproduct it turns out that instead of taking equivalence classes in $\mathcal{P}[D]$ we can choose maximal members of the equivalence classes. With the aid of a topological closure operator and the convex closure operator, Con, defined in § 3, we get an internal characterization of the maximal equivalence classes. We then go on to obtain similar characterizations of the finite elements of $\mathcal{P}[D]$ and l.u.b.'s of sequences in $\mathcal{P}[D]$. Finally we can provide evidence that the $\sqsubseteq$ ordering on sets of sequences (of integers, say) is not too unnatural. It turns out that, in many cases, $\sqsubseteq$ is just $\sqsubseteq_M$.

Let D be any SFP object. Then by definition we have a directed sequence $\mathcal{D} = \langle D_m, p_{mn}\rangle$ of finite ipo's such that $D = \lim_{\rightarrow} \mathcal{D}$. Let $\langle r_m\rangle = \langle\langle i_m, j_m\rangle\rangle$ be the universal cone from $\mathcal{D}$ to D. Let $\hat{\mathcal{D}} = \langle \mathcal{P}[D_m], \hat{p}_{mn}\rangle$ where $\hat{p}_{mn} = \langle \hat{\varphi}_{mn}, \hat{\psi}_{nm}\rangle$ $(m \leqq n)$. Let $E = \lim_{\rightarrow} \hat{\mathcal{D}}$ with universal cone, $\langle \tilde{r}_m\rangle = \langle \tilde{i}_m, \tilde{j}_m\rangle$, say. By establishing an isomorphism between E and $E' = \langle \mathcal{F}[D]/\simeq, \sqsubseteq/\simeq\rangle$ we shall see, by Theorem 5, that $\mathcal{P}[D]$ is an SFP object. The technique used is to define functions $\xi : E \to \mathcal{F}[D]$ and $\eta : \mathcal{F}[D] \to E$ which will induce the isomorphism. They are defined by:

$$\xi(e) = \{d \in D \mid \forall m \geqq 0. j_m(d) \in e_m\} \qquad (e \in E),$$

$$\eta(X) = \langle \mathrm{Con}\,(j_m(X))\rangle_{m=0}^{\infty} \qquad (X \in \mathcal{F}[D]).$$

We should show that they are well-defined. Suppose $e \in E$. Consider the finitary tree whose nodes are sequences $\langle d^{(1)}, \cdots, d^{(n)}\rangle$ $(n \geqq 0)$ with $d^{(m)} \in e_m$ $(0 < m \leqq n)$ and $\psi_{(m+1)m}(d^{(m+1)}) = d^{(m)}$ $(0 < m < n)$. If T is its completion, considered as an ipo, define $f : T \to E$ by:

$$f(\omega) = \begin{cases} \bot & (\omega \text{ has length } 0), \\ i_n((\omega)_n & (\omega \text{ has length } n > 0), \\ \bigsqcup_{n>0} i_n((\omega)_n) & (\omega \text{ is infinite}). \end{cases}$$

Then $\xi(e)$ is $Bd(f)$, the boundary of f, and so is finitely generable.
As for η, we have:

$$\begin{aligned} \hat{\psi}_{mn}(\eta(X)_m) &= \mathrm{Con}\,(\psi_{mn}(j_m(X))) \\ &= \mathrm{Con}\,(j_n(X)) \\ &= \eta(X)_n \quad \text{for } m \geqq n. \end{aligned}$$

In the arguments that follow, we often use König's lemma in a particular way. Suppose $\langle X_n\rangle_{n=0}^{\infty} \in E$ for $n \geqq 0$ and $u^{(n)} \in X_n$ for $n \geqq 0$. Then there is an x in D such that for all $n \geqq 0$, $j_n(x) \in X_n$ and there is an $m \geqq n$ such that $\psi_{mn}(u^{(m)}) = j_n(x)$. To see this, consider the finitary tree whose nodes are sequences $\langle d^{(1)}, \cdots, d^{(n)}\rangle$ $(n \geqq 0)$ with $d^{(m)}$ in X_m $(0 < m \leqq n)$, with $\psi_{(m+1)m}(d^{(m+1)}) = d^{(m)}.(0 < m < n)$ and

such that there is a $p \geqq n$ such that $d^{(n)} = \psi_{pn}(u^{(p)})$. As it is infinite, it follows from König's lemma that it has an infinite branch, $\omega_0, \omega_1, \cdots$. Let $x = \bigsqcup_{n>0} i_n((\omega_n)_n)$. Then x is well-defined and has the required properties.

LEMMA 2. (i) *If* $e \sqsubseteq e'$, *then* $\xi(e) \sqsubseteq_M \xi(e')$ $(e, e' \in E)$.

(ii) *If* $X \sqsubseteq Y$, *then* $\eta(X) \sqsubseteq \eta(Y)$ $(X, Y \in \mathcal{F}[D])$.

(iii) $\eta \circ \xi = I_E$.

(iv) *If* $X \in \mathcal{F}[D]$, *then* $\xi \circ \eta(X) \supseteq X$, $\xi \circ \eta(X) \supseteq_M X$ and $\xi \circ \eta(X) \simeq X$.

Proof. (i) Suppose $x \in \xi(e)$. Then for all $n \geqq 0$ there are $u^{(n)} \in (e')_n$ such that $u^{(n)} \supseteq j_n(x)$. Applying König's lemma in the form described we obtain a $y \in D$ such that for all $n \geqq 0$, $y_n \in (e')_n$ and there is an $m \geqq n$ such that $y_n = \psi_{mn}(u^{(m)})$. As $u^{(m)} \supseteq j_m(x)$, it follows that $y_n \supseteq x_n$. Therefore $y \in \xi(e')$ and $y \supseteq x$.

Suppose $y \in \xi(e')$. Then for all $n \geqq 0$ there are $u^{(n)} \in (e)_n$ such that $u^{(n)} \sqsubseteq y_n$. A similar argument using König's lemma provides an $x \in \xi(e)$ such that $x \sqsubseteq y$.

(ii) Here,

$$\begin{aligned}\eta(X) &= \langle \mathrm{Con}\,(j_m(X))\rangle_{m=0}^{\infty} \\ &\sqsubseteq \langle \mathrm{Con}\,(j_m(Y))\rangle_{m=0}^{\infty} \quad (\text{as } X \sqsubseteq Y,\ j_m(X) \sqsubseteq_M j_m(Y) \\ &\qquad\qquad \text{by Theorems 2.2 and 3.2}) \\ &= \eta(Y).\end{aligned}$$

(iii) We must show that $\eta(\xi(e)) = e$ for any $e \in E$. That is, we must show that $e_m = \mathrm{Con}\,(j_m\{d \in D \mid \forall m \geqq 0. j_m(d) \in e_m\})$ for all $m \geqq 0$. Clearly RHS $\subseteq$ LHS (left-hand side). Take $x_m \in e_m$. Since $\hat{\psi}_{nm}(e_n) = e_m$ for $n \geqq m$, there are $u^{(n)}$, $v^{(n)}$ in e_n such that $\psi_{nm}(u^{(n)}) \supseteq x_m \supseteq \psi_{nm}(v^{(n)})$ for $n \geqq m$. Define $u^{(n)} = v^{(n)} = \psi_{mn}(x_m)$ for $n < m$. Applying König's lemma twice in the form prescribed above we obtain u, v in D such that $u_n, v_n \in e_n$ for all n and $u_m \supseteq x_m \supseteq y_m$, which shows that $x_m \in$ RHS.

(iv) We have, $\xi \circ \eta(X) = \{d \in D \mid \forall m \geqq 0. j_m(d) \in \mathrm{Con}\,(j_m(X))\}$. If $d \in X$, then $j_m(d) \in j_m(X)$ and so $\xi \circ \eta(X) \supseteq X$. To show that $X \sqsubseteq_M \xi \circ \eta(X)$ it only remains to prove that if $y \in \xi \circ \eta(X)$, then $x \sqsubseteq y$, for some $x \in X$. So suppose $y \in \xi \circ \eta(X)$. Then for all $m \geqq 0$, $j_m(y)$ is in $\mathrm{Con}\,(j_m(X))$. There is an $x^{(m)}$ in X such that $j_m(x^{(m)}) \sqsubseteq j_m(y)$. Therefore for all $n \geqq m$, $j_m(x^{(n)}) \sqsubseteq j_m(y)$.

Since X is finitely generable there is a $g : \Omega \to D$ such that $X = Bd(g)$. Let T be the finitary subtree of Ω consisting of the finite ω in Ω such that for infinitely many $n \geqq 0$, $g(\omega) \sqsubseteq x^{(n)}$. Clearly the empty sequence is a node in T. If ω is a node in T, then one of its two successors in Ω is in T. Therefore T has infinitely many nodes and we may apply König's lemma to obtain an infinite branch $\omega_0, \omega_1, \cdots$.

Now if ω is a node in T, then if $m \geqq 0$, there is an $n \geqq m$ such that $g(\omega) \sqsubseteq x^{(n)}$. Then $j_m(g(\omega)) \sqsubseteq j_m(x^{(n)}) \sqsubseteq j_m(y)$ by the definition of T and the properties of the $x^{(n)}$. Therefore $g(\omega) \sqsubseteq y$. Now if we put $x = \bigsqcup_{n \geqq 0} g(\omega_i)$, then $x \in X$ and $x \sqsubseteq y$, which concludes the proof that $X \sqsubseteq_M \xi \circ \eta(X)$.

To show that $\xi \circ \eta(X) \simeq X$ it only remains to show that $\xi \circ \eta(X) \sqsubseteq X$. We use Theorem 2(i). Let $f : D \to \mathbb{O}$. We must show that $f(\xi \circ \eta(X)) \sqsubseteq_M f(X)$. If $y \in f(X)$, then $y = f(x)$ for some x in $X \subseteq \xi \circ \eta(X)$. Therefore $y \in f(\xi \circ \eta(X))$. So we must show that if $x \in f(\xi \circ \eta(X))$, then for some $y \in f(X)$, $x \sqsubseteq y$. The case $x = \bot$ is trivial. If $x = \top$, then, by the continuity of f, there is a d in $\xi \circ \eta(X)$ and an $m \geqq 0$ such that $\top = f(i_m \circ j_m(d))$. As $j_m(d) \in \mathrm{Con}\,(j_m(X))$, there is an $x \in X$ such that $j_m(d) \sqsubseteq j_m(x)$. Then $\top = f(x) \in f(X)$, as required. □

THEOREM 6. *$\mathscr{P}[D] \cong E$ and is, therefore, an* SFP *object.*

Proof. Let $[X]$ be the equivalence class of X in $\mathscr{F}[D]$ modulo $\simeq$. Define $\xi/\simeq : E \to E'$ by:

$$(\xi/\simeq)(e) = [\xi(e)]$$

and $(\eta/\simeq) : E' \to E$ by:

$$(\eta/\simeq)([X]) = \eta(X).$$

By Lemma 2(ii), $\eta/\simeq$ is well-defined. By Lemmas 2(i) and 2(ii) and Theorem 2(iii), $\xi/\simeq$ and $\eta/\simeq$ are monotonic. By Lemmas 2(iii) and 2(iv) they are mutual inverses.

It is possible to choose maximal members of the equivalence classes to get a better description of $\mathscr{P}[D]$. Define an operation on $\mathscr{F}[D]$: by: $X^* = \xi \circ \eta(X)$.

COROLLARY 1. (i) *The operation X^* is a closure operation on $\mathscr{F}[D]$.*

(ii) *For X, Y in $\mathscr{F}[D]$, $X \sqsubseteq Y$ iff $X^* \sqsubseteq_M Y^*$, and $X \simeq Y$ iff $X^* = Y^*$.*

(iii) $\mathscr{P}[D] \cong \langle \{X^* | X \in \mathscr{F}[D]\}, \sqsubseteq_M \rangle$.

Proof. (i) We show that if $X \subseteq Y$, then $X^* \subseteq Y^*$; that $X \subseteq X^*$, and that $X^{**} = X^*$. The first is immediate from the definitions of ξ and η, the second from Lemma 2(iv) and the third from Lemma 2(iii).

(ii) That $X \sqsubseteq Y$ implies $X^* \sqsubseteq_M Y^*$ follows from Lemmas 2(i) and 2(iii). The converse follows from Lemma 2(iv) and Theorem 2(iii). The other part is similar.

(iii) We can define $\alpha : E' \to \langle \{X^* | X \in \mathscr{F}[D]\}, \sqsubseteq_M \rangle$ and $\beta : \langle \{X^* | X \in \mathscr{F}[D]\}, \sqsubseteq_M \rangle \to E'$ by:

$$\alpha([X]) = X^* \qquad (X \in \mathscr{F}[D]),$$
$$\beta(X) = [X] \qquad (X = X^* \in \mathscr{F}[D]).$$

Part (ii) shows that α is well-defined and monotonic and that β is monotonic. By part (i) and Lemma 2(iv) they are mutual inverses. Therefore they are isomorphisms.

We will take this definition of $\mathscr{P}[D]$ as our standard one. It can be seen that this agrees with the previous definition when D is finite. For by Lemma 2(iv), X^* is the maximal member of $[X]$, and by Theorem 3, so is Con (X) when D is finite. It also agrees with the earlier standard definition when D is discrete.

From the proof of Theorem 6 and that of Corollary 1 we see that if we define $\eta^* : \mathscr{P}[D] \to E$ and $\xi^* : E \to \mathscr{P}[D]$ by $\eta^* = (\eta/\simeq) \circ \beta$ and $\xi^* = \alpha \circ (\xi/\simeq)$, then η^* and ξ^* are isomorphisms. For they are mutual inverses. One sees too that $\eta^*(X) = \eta(X)$, for all X in $\mathscr{P}[D]$ and, using Lemma 2(iii), $\xi^*(e) = \xi(e)$ for e in E.

With the aid of a lemma we can get a picture of the finite elements of $\mathscr{P}[D]$.

LEMMA 3. (i) *For $X \in \mathscr{P}[D]$, $\tilde{j}_n \circ \eta^*(X) =$ Con $(j_n(X))$ $(n \geqq 0)$.*

(ii) *For $X \in \mathscr{P}[D_n]$, $\xi^* \circ \tilde{i}_n(X) =$ Con $(i_n(X))$ $(n \geqq 0)$.*

Proof. (i) The proof is immediate from the definition of η^*.

(ii) For $X \in \mathscr{P}[D_n]$,

$$\xi^* \circ \tilde{i}_n(X) = \{d \in D | j_m(d) \in (\tilde{i}_n(X))_m, m \geqq 0\}$$
$$= \{d \in D | j_m(d) \in \hat{\varphi}_{nm}(X), m \geqq n\}.$$

This clearly includes $i_n(X)$ and therefore since it is closed it includes Con $(i_n(X))$. Now if $d \in \xi^* \circ \tilde{i}_n(X)$, then for all $m \geqq n$, $j_m(d) \in \hat{\varphi}_{nm}(X)$ and therefore there are

$u^{(m)}, v^{(m)} \in X$ such that $i_n(u^{(m)}) \sqsubseteq i_m \circ j_m(d) \sqsubseteq i_n(v^{(m)})$ for $m \geqq n$. Since X is finite, infinitely many of the $u^{(m)}$ are identical, as are infinitely many of the $v^{(m)}$. Therefore there are u, v in X such that $i_n(u) \sqsubseteq d \sqsubseteq i_n(v)$, showing that d is in Con $(i_n(X))$ as required.

From the proof of Theorem 5, we see that the finite elements of E are those of the form $\tilde{i}_n(X)$ where $X \in \mathcal{P}[D_n]$. Therefore the finite elements of $\mathcal{P}[D]$ have the form, $\xi^* \circ \tilde{i}_n(X)$ or, by the lemma, Con $(i_n(X))$ where $X \in \mathcal{P}[D_n]$. That is, they are the convex closures of finite nonempty sets of finite elements of D.

One can then obtain another interesting characterization of $\mathcal{P}[D]$: it is the completion in IPO of the partial order

$$\langle \{\text{Con}\,(X) | X \text{ is a finite, nonempty set of finite elements of } D\}, \sqsubseteq_M \rangle.$$

The proof is straightforward. Since we do not use the result, we omit the proof.

The description of the closure operation X^* in terms of ξ and η is external to D, involving an arbitrary choice of a sequence $\mathcal{D}$ whose limit is D. Considering the example of subsets of Ω which are $\simeq$ but not $=_M$ suggests looking for a suitable notion of limit. This is provided by a topology of positive and negative information.

DEFINITION. The *Cantor topology* on D has, as sub-basis, the sets $P_e = \{x | x \sqsupseteq e\}$ and $N_e = \{x | x \not\sqsupseteq e\}$ for finite e.

It can be shown that when D is a lattice, the Cantor topology is the order convergence topology defined by Birkhoff in [3].

DEFINITION. An *information* α is a pair with $(\alpha)_1$ a finite member of D and $(\alpha)_2$ a finite set of finite elements of D.

$$\mathcal{N}_\alpha =_{\text{def}} P_{\alpha_1} \cap \bigcap_{e \in \alpha_2} N_e.$$

Each $\mathcal{N}_\alpha$ is open and closed and the $\mathcal{N}_\alpha$ form a basis for the topology, for

$$P_e \cap P_{e'} = \cup\{P_d | d \in \mathcal{U}_D(\{e, e'\})\}.$$

Notice that if D is finite, this is just the discrete topology. In fact in general it is the topology inherited from the discrete topologies on the D_n. For, as a set, D is the limit of the inverse system of sets $\langle D_n, \psi_{mn} \rangle$ and since the topologies in the D_n are discrete, the ψ_{mn} are continuous, in the topological sense. Therefore, by [4], D has an inherited topology with basis $\{j_n^{-1}(X) | X \subseteq D_n\}$, and this is just $\{\mathcal{N}_\alpha\}$. Now as topological spaces, the D_n are all compact. Therefore, by [4], so is D. Consequently D is Hausdorff, and every finite subset of D is closed.

We called the topology the Cantor topology since under the natural injection $\varepsilon : D \to \mathcal{P}(\omega)$, it can be viewed as a subspace of Cantor space. Here $\varepsilon(d) = \{n | e_n \sqsubseteq d\}$ $(d \in D)$, where $e_0, e_1, \cdots$ is an enumeration of the finite elements of D [17]. It is easily seen to be continuous as a map from D under the Cantor topology to Cantor space. It is not hard to show that $\{\varepsilon(d) | d \in D\}$ is a closed set in Cantor space and so we have another proof that the topology is compact.

Recall the usual definition of a limit of a sequence:

$$\lim \langle x_n \rangle_{n=0}^{\infty} = a \text{ iff every neighborhood of } a \text{ contains almost all the } x_n.$$

This can be reformulated using the sub-basis:

$\lim \langle x_n \rangle_{n=0}^{\infty} = a$ iff.($\forall P_e . a \in P_e$ implies almost all the x_n are in P_e)
and ($\forall N_e . a \in N_e$ implies almost all the x_n are in N_e).

Then the elementary properties of lim are:

c0: $\lim \vec{x} = a$ and $\lim \vec{x} = b$ implies $a = b$.

c1: If $\vec{x}$ is eventually constant, its limit is that constant.

c2: If $\lim \vec{x} = a$ and $\vec{y}$ is a subsequence of $\vec{x}$, $\lim \vec{y} = a$.

Example. In Ω, $\lim \langle 0^n 1^\omega \rangle_{n=0}^{\infty} = 0^\omega$.

Since D is compact, every sequence has a convergent subsequence. Since D has a denumerable basis, a set is closed iff it is closed under limits of sequences. This is all the directly topological information required about D.

LEMMA 4. (i) *Suppose* $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$ *is an increasing sequence. Then* $\lim \langle x_n \rangle_{n=0}^{\infty} = \bigsqcup_{n \geqq 0} x_n$.

(ii) *Let* $\langle x_n \rangle_{n=0}^{\infty}$ *be a sequence in* D. *Suppose* $\lim \langle i_n \circ j_n(x_n) \rangle_{n=0}^{\infty} = x$. *Then* $\lim \langle x_n \rangle_{n=0}^{\infty} = x$.

(iii) *Let* $\langle x_n \rangle_{n=0}^{\infty}$ *and* $\langle y_n \rangle_{n=0}^{\infty}$ *be sequences in* D. *Suppose* $\lim \langle x_n \rangle_{n=0}^{\infty} = a$, $\lim \langle y_n \rangle_{n=0}^{\infty} = b$ *and* $x_n \sqsubseteq y_n$ *for all* n. *Then* $a \sqsubseteq b$.

(iv) (Milner) *Every nonempty closed subset of* D *is finitely generable.*

Proof. (i) If $\bigsqcup_{n \geqq 0} x_n \in P_e$, then some and hence almost all the x_n are in P_e. If $\bigsqcup_{n \geqq 0} x_n \in N_e$, then every $x_n \in N_e$.

(ii) If $x \in P_e$, then almost all the $i_n \circ j_n(x_n)$ are in P_e. As $x_n \sqsupseteq i_n \circ j_n(x_n)$ for all n, it follows that almost all the x_n are in P_e. If $x \in N_e$, then all the $i_n \circ j_n(x_n)$ are in N_e for $n \geqq n_0$, say. For $n \geqq n_1$, say $e = i_n \circ j_n(e)$. Then for $n \geqq \max(n_0, n_1)$, $x_n \in N_e$ as otherwise $x_n \sqsupseteq e$ and $i_n \circ j_n(x_n) \sqsupseteq i_n \circ j_n(e) = e$.

(iii) If a is greater than some finite e, then $a \in P_e$ so almost all the x_n are in P_e, so almost all the y_n are in P_e. Therefore, as D is algebraic, $b \sqsupseteq a$.

(iv) Let $X \subseteq D$ be closed and nonempty. Let T be the completion of the finitary tree with nodes of the form $\langle i_1 \circ j_1(x), \cdots, i_n \circ j_n(x) \rangle$ for $x \in D$ and $n \geqq 0$, with the subsequence ordering. Define $g : T \to D$ by

$$g(\omega) = \begin{cases} \bot & (\omega = \langle \cdot \rangle), \\ (\omega)_n & (\omega \text{ finite and not } \langle \cdot \rangle), \\ \bigsqcup_{n \geqq 0} (\omega)_n & (\omega \text{ infinite}). \end{cases}$$

Clearly $X \subseteq Bd(g)$. If $x \in Bd(g)$, then by (i) and (ii) there are $x_n \in X$ such that $x = \lim \langle i_n \circ j_n(x_n) \rangle_{n=0}^{\infty} = \lim \langle x_n \rangle_{n=0}^{\infty}$. But since X is closed, $x \in X$. □

We can now characterize the closure operation which gives the maximal members of $\simeq$-equivalence classes. Let Cl (X) be the closure of X in the Cantor topology.

THEOREM 7. (i) *For any* $X \subseteq D$, Con $\circ$ Cl (X) *is the least convex set containing* X *which is closed in the Cantor topology.*

(ii) *For* $X \in \mathcal{F}[D]$, $X^* =$ Con $\circ$ Cl (X).

(iii) *For* $X \in \mathcal{F}[D]$, $X \simeq$ Cl (X).

Proof. (i) Con ∘ Cl (X) is clearly convex. Suppose $\vec{y} = \langle y_n \rangle_{n=0}^{\infty}$ is a convergent sequence in Con ∘ Cl (X). Then there are sequences $\vec{x} = \langle x_n \rangle_{n=0}^{\infty}$ and $\vec{z} = \langle z_n \rangle_{n=0}^{\infty}$ in Cl (X) such that $x_n \sqsubseteq y_n \sqsubseteq z_n$ for all n. Now $\vec{x}$ has a subsequence $\vec{x}'$ converging to $x' \in \text{Cl}(X)$. As $x_n \sqsubseteq y_n$ for all n, $x' \sqsubseteq \lim \vec{y}$. Similarly there is a z' in Cl (X) such that $\lim \vec{y} \sqsubseteq z'$. Therefore $\lim \vec{y}$ is in Con ∘ Cl (X).

So Con ∘ Cl (X) is also closed in the Cantor topology. If $Y \supseteq X$ and is convex and topologically closed, then $Y = \text{Con} \circ \text{Cl}(Y) \supseteq \text{Con} \circ \text{Cl}(X)$.

(ii) By definition, $X^* = \{d \in D \mid \forall m \geqq 0. j_m(d) \in \text{Con}(j_m(X))\}$. Therefore $d \in X^*$ iff for all $m \geqq 0$ there are $u^{(m)}$, $v^{(m)}$ in X such that $j_m(u^{(m)}) \sqsubseteq j_m(d) \sqsubseteq j_m(v^{(m)})$. Since every sequence has a convergent subsequence, and $j_m(x) \sqsubseteq j_m(y)$ implies $j_n(x) \sqsubseteq j_n(y)$ for $n \leqq m$, we may assume without loss of generality that $\langle u^{(m)} \rangle_{m=0}^{\infty}$ and $\langle v^{(m)} \rangle_{m=0}^{\infty}$ are convergent to, say, u and v, respectively. Then

$$\begin{aligned} u = \lim \langle u^{(m)} \rangle_{m=0}^{\infty} &= \lim \langle i_m \circ j_m(u^{(m)}) \rangle_{m=0}^{\infty} \qquad \text{(by Lemma 4(ii))} \\ &\sqsubseteq \lim \langle i_m \circ j_m(d) \rangle_{m=0}^{\infty} \\ &= d \\ &\sqsubseteq v \qquad \text{(similarly).} \end{aligned}$$

Therefore d is in Con ∘ Cl (X) and so $X^* \subseteq \text{Con} \circ \text{Cl}(X)$.

Conversely, if x is in Cl (X), then $x = \lim \langle x_n \rangle_{n=0}^{\infty}$ where x_n is in X for all n. By the definition of limits, for all m there is an n such that $j_m(x) = j_m(x_n) \in \text{Con}(j_m(X))$. So, by the definition of X^*, $x \in X^*$. Therefore $\text{Cl}(X) \subseteq X^*$. Thus $X^* = \text{Con}(X^*) \supseteq \text{Con} \circ \text{Cl}(X)$.

(iii) We have, $X \simeq X^* = \text{Con} \circ \text{Cl}(X) \simeq \text{Cl}(X)$, using part (ii) and Theorem 2(iii). □

The next theorem gives a picture of the l.u.b. operation on increasing sequences.

THEOREM 8. *Suppose* $X_0 \sqsubseteq_M X_1 \sqsubseteq_M \cdots$ *is an increasing sequence of nonempty sets closed in the Cantor topology. Then* $\bigsqcup_{n \geqq 0} \text{Con}(X_n) = \{\bigsqcup_{n \geqq 0} x_n \mid$ *for all* $n \geqq 0$, $x_n \sqsubseteq x_{n+1}$ *and* $x_n \in X_n\}^*$.

Proof. Let $Y = \{\bigsqcup_{n \geqq 0} x_n \mid \text{for all } n \geqq 0, x_n \sqsubseteq x_{n+1} \text{ and } x_n \in X_n\}$. If $x_m \in X_m$, then there are x_n in X_n. $(n \neq m)$ such that $\langle x_n \rangle_{n=0}^{\infty}$ is an increasing sequence. Therefore there is a y in Y such that $y \sqsupseteq x_n$. So Y is nonempty and Y^* is in $\mathcal{P}[D]$. As clearly every element of Y has a lower bound in every X_m, $Y \sqsupseteq_M X_m$ for all m.

Suppose that $Z \sqsupseteq_M X_m$ for all m, for some Z in $\mathcal{P}[D]$. We show that $Y \sqsubseteq_M Z$. If $y \in Y$, then there is an increasing sequence, $\langle x_m \rangle_{m=0}^{\infty}$, such that $y = \bigsqcup_{m \geqq 0} x_m$ and $x_m \in X_m \sqsubseteq_M Z$. Therefore, for all $m \geqq 0$ there is a z_m in Z such that $z_m \sqsupseteq x_m$. So if z is the limit of a convergent subsequence of $\langle z_m \rangle_{m=0}^{\infty}$, then $z \sqsupseteq x$, and since Z is closed, z is in Z.

Conversely take z in Z. For each $m \geqq 0$ there is an x_m in X_m such that $z \sqsupseteq x_m$. We can then find for all m and $n \leqq m$, $u^{(m,n)}$ in X_n such that $u^{(m,0)} \sqsubseteq u^{(m,1)} \sqsubseteq \cdots \sqsubseteq u^{(m,m)} = x_m \sqsubseteq z$. Let $Y_m = \{u^{(n,m)} \mid n \geqq m\} \subseteq X_m$. If $\langle v_s \rangle_{s=0}^{\infty}$ is a convergent sequence with limit v, whose members are in Y_m, then there is a sequence $\langle v_s' \rangle_{s=0}^{\infty}$ in Y_{m+1} such that $v_s' \sqsupseteq v_s$ for almost all s. Then if v' is the limit of a convergent subsequence of $\langle v_s' \rangle_{s=0}^{\infty}$, we have $v' \sqsupseteq v$. So if we take v_0 in Y_0 to be the limit of a convergent sequence whose members are in Y_0, we can successively

choose $v_1, v_2 \cdots$ such that $v_0 \sqsubseteq v_1 \sqsubseteq \cdots$ and v_m is the limit of a convergent sequence whose members are in $Y_m \subseteq X_m$. So as the X_m are closed in the Cantor topology, $v_m \in X_m$ for all m. But each v_m is the limit of elements less than z. Therefore $v_m \sqsubseteq z$ and $\bigsqcup_{m \geq 0} v_m$ is in Y and is less than z. Thus $Z \sqsupseteq_M Y \simeq Y^*$. □

We now have a picture of $\mathscr{P}[D]$ which is enough for our present purposes: its elements have the form Con ∘ Cl (X) for X in $\mathscr{F}[D]$; its ordering is $\sqsubseteq_M$; its finite elements have the form Con (X) where X is a finite set of finite elements of D and l.u.b.'s of sequences have the form described in the above theorem. It would be interesting to find out a good general form for the l.u.b.'s of directed sets.

Let us consider a special case, incidentally verifying some assertions made in § 2. Suppose S^ω is the domain of finite and infinite sequences of elements from a set S, with the subsequence ordering.

THEOREM 9. (i) *If $X \in \mathscr{F}[S^\omega]$, then X^* is the least convex set containing X closed under* l.u.b.'s *of increasing sequences. In particular, if every element in X is infinite, then $X^* = X$.*

(ii) *Suppose X, $Y \in \mathscr{F}[S^\omega]$ and every element of Y is infinite. Then $X \sqsubseteq Y$ iff $X \sqsubseteq_M Y$. If every element of X is also infinite, then $X \sqsubseteq Y$ iff $X = Y$.*

Proof. (i) Suppose y is a limit point of X. Then $y \in X^*$, by Theorem 7(ii). As $X \sqsubseteq_M X^*$ by Lemma 2(iv), $y \sqsupseteq x$ for some x in X. If x is infinite $y = x$. Otherwise x is finite. Now if $y \in P_e$ where $e \sqsupseteq x$, as y is a limit point, some u in X is also in P_e. Therefore e is in Con (X). Thus y is the l.u.b. of an increasing sequence of elements of Con (X). By Theorems 7(i) and 7(ii), X^* is as described. When every element of X is infinite, the least convex set containing X and closed under l.u.b.'s of increasing sequences is X.

(ii) Suppose every element of Y is infinite. If $X \sqsubseteq_M Y$, then $X \sqsubseteq Y$ by Theorem 2(iii). If $X \sqsubseteq Y$, then

$$\begin{aligned} X &\sqsubseteq_M X^* && \text{(by Lemma 2(iv))} \\ &\sqsubseteq_M Y^* && \text{(by Corollary 1(ii))} \\ &= Y && \text{(by part (i)).} \end{aligned}$$

If every element of X is infinite, then $X \sqsubseteq_M Y$ implies $X = Y$. □

We hope that this theorem increases the plausibility of the $\sqsubseteq$ ordering for sequence domains.

6. Continuity of various functions. In this section we consider various functions which are useful when defining denotational semantics. Although we cannot manipulate sets as freely as usual it is nonetheless possible to obtain reasonable analogues of many standard functions. In § 5 we showed that $\mathscr{P}[\,\cdot\,]$ sends SFP objects to SFP objects. We now extend $\mathscr{P}[\,\cdot\,]$ to maps and thereby obtain a functor from SFP to SFP.

We will feel free to use the results of previous sections without explicit reference, when their application presents no particular difficulties. The results used in this way are Theorems 2, 3, 5 (for the properties of SFP objects), Lemma 2(iv) (in the form $X \simeq X^*$ and $X \sqsubseteq_M X^*$ when $X \in \mathscr{F}[D]$), Corollary 1, Lemma 4 and Theorem 7. We also use the easily proved fact that if D is an SFP-object, U is a finite set of finite elements of D and $X \in \mathscr{F}[D]$, then $U \sqsubseteq X^*$ implies $U \sqsubseteq_M X$.

The next lemma provides a useful criterion for continuity.

LEMMA 5. *Let E and F be algebraic* ipo's. *Let f be a monotonic map from E to F. Then f is continuous iff for every x in E and every finite y in F, such that $y \sqsubseteq f(x)$, there is a finite u in E such that $u \sqsubseteq x$ and $y \sqsubseteq f(u)$.*

Proof. Suppose f is continuous, $x \in E$, $y \sqsubseteq f(x)$ and y is finite. Then, as E is algebraic and f is continuous,

$$y \sqsubseteq f(x) = \bigsqcup\{f(u) | u \sqsubseteq x \text{ and } u \text{ is finite}\}$$

and the set on the RHS is directed. As y is finite $y \sqsubseteq f(u)$ for some finite $u \sqsubseteq x$.

Conversely, suppose f is monotonic and the condition of the hypothesis holds. Suppose $X \subseteq E$ is directed. As f is monotonic, $f(\bigsqcup X) \sqsupseteq \bigsqcup f(X)$. Conversely, suppose $y \sqsubseteq f(\bigsqcup X)$ and y is finite. Then by assumption, there is a finite $u \sqsubseteq \bigsqcup X$ such that $y \sqsubseteq f(u)$. As u is finite and X is directed, $u \sqsubseteq x$ for some x in X. So $y \sqsubseteq f(u) \sqsubseteq f(x) \sqsubseteq \bigsqcup f(X)$, as f is monotonic. As F is algebraic, $f(\bigsqcup X) \sqsubseteq \bigsqcup f(X)$. □

Function extension. Suppose $f: D \to E$ where D and E are SFP objects. Define $\hat{f}: \mathcal{P}[D] \to \mathcal{P}[E]$ by:

$$\hat{f}(X) = (f(X))^* \qquad (X \in \mathcal{P}[D]).$$

We use Lemma 5 to show that $\hat{f}$ is continuous. First, suppose X, Y are in $\mathcal{P}[D]$ and $X \sqsubseteq Y$. Then $\hat{f}(X) \simeq f(X) \sqsubseteq f(Y) \simeq \hat{f}(Y)$. So f is monotonic.

Next suppose $X \in \mathcal{P}[D]$, $V \in \mathcal{P}[E]$, V is finite in $\mathcal{P}[E]$ and $V \sqsubseteq \hat{f}(X)$. Then $V = U^*$ for a finite set U of finite elements of E. So $U \sqsubseteq_M U^* = V \sqsubseteq \hat{f}(X) \simeq f(X)$. Therefore $U \sqsubseteq_M f(X)$. As X is closed in the Cantor topology, the construction in the proof of Lemma 4(iv) provides us with a $g: \Omega \to D$ such that $X = Bd(g)$ and if $\omega \in \Omega$ is finite so is $g(\omega)$.

Now set $U_\omega = \{u \in U | u \sqsubseteq f \circ g(\omega)\}$ $(\omega \in \Omega)$. Clearly if $\omega, \nu \in \Omega$ and $\omega \sqsubseteq \nu$, then $U_\omega \subseteq U_\nu \subseteq U$. Let $T = \{\omega \in \Omega | \omega = \bot \text{ or } \exists \nu \sqsupseteq \omega \cdot U_\nu \neq U_\omega\}$. Clearly every element of T is finite and T is a finitary tree under the subsequence ordering. As T has no infinite branches (U is finite), König's lemma tells us that T is finite. Let Y be the set of those elements of Ω whose predecessor is in T. Since T is a finite subtree of Ω every infinite element of Ω is greater than some element of Y. Therefore $g(Y) \sqsubseteq_M X$. Also $U \sqsubseteq_M f \circ g(Y)$. For if $u \in U$, then $u \sqsubseteq f(g(\omega))$ for some infinite ω. If $\nu \in Y$ and $\nu \sqsubseteq \omega$, then $U_\nu = U_\omega$ and so $u \sqsubseteq f \circ g(\nu)$. Conversely, if $\nu \in Y$, choose an infinite $\omega \in \Omega$ such that $\nu \sqsubseteq \omega$. Then $U_\nu = U_\omega$ and so $u \sqsubseteq f \circ g(\nu)$ for some $u \in U$, as $f \circ g(\omega) \in f(X) \sqsupseteq_M U$.

So $V = U^* \sqsubseteq_M U \sqsubseteq_M f(g(Y)) \simeq f((g(Y))^*) \simeq \hat{f}((g(Y))^*)$, $(g(Y))^*$ is a finite element of $\mathcal{P}[D]$ and $(g(Y))^* \sqsubseteq_M g(Y) \sqsubseteq_M Y$. Therefore f is continuous, by Lemma 5.

Just as in the finite case, function extension preserves all the identities and commutes with composition. Suppose $f: D \to E$ and $g: E \to F$ where D, E and F are SFP objects. Take X in $\mathcal{P}[D]$. As X is finitely generable, so is $f(X)$ and so $f(X) \simeq \hat{f}(X)$. Therefore $g(f(X)) \simeq g(\hat{f}(X)) \simeq \hat{g}(\hat{f}(X))$ (as $f(X)$ is finitely generable). Therefore as $\widehat{g \circ f}(X)$ is finitely generable, $\hat{g} \circ \hat{f}(X) = \widehat{g \circ f}(X)$.

It follows that if we define the action of $\mathscr{P}$ on morphisms $f: D \to E$ by:

$$\mathscr{P}[f] = \hat{f},$$

then $\mathscr{P}$ turns into a functor from SFP to SFP.

The singleton function. Let D be an SFP object. Define $[\![\cdot]\!]: D \to \mathscr{P}[D]$ by:

$$[\![x]\!] = \{x\} \qquad (x \in D).$$

It is clear that $[\![\cdot]\!]$ is monotonic. Suppose V is a finite element of $\mathscr{P}[D]$ and $V \sqsubseteq \{x\}$ (for x in D). Then $V = U^*$ where U is a finite set of finite elements of D. So $U \sqsubseteq_M \{x\}$. Therefore x is an upper bound of U and there is a finite d less than x in $\mathscr{U}(U)$. Then $V \sqsubseteq [\![d]\!]$ and d is finite and $d \sqsubseteq x$. By Lemma 5, $[\![\cdot]\!]$ is continuous.

Union. Let D be an SFP object. Define $\uplus: \mathscr{P}[D] \times \mathscr{P}[D] \to \mathscr{P}[D]$ by:

$$X \uplus Y = \mathrm{Con}(X \cup Y) \qquad (X, Y \in \mathscr{P}[D]).$$

If X and Y are closed in the Cantor topology so is $X \cup Y$. Therefore $\uplus$ is well-defined.

If $X \sqsubseteq X'$ and $Y \sqsubseteq Y'$ for X, X', Y, Y' in $\mathscr{P}[D]$, then $X \cup Y \sqsubseteq_M X' \cup Y'$ as $X \sqsubseteq_M X'$ and $Y \sqsubseteq_M Y'$. Therefore $\uplus$ is monotonic.

For continuity, we use the fact that the Cartesian product of two algebraic ipo's is an algebraic ipo. If E, F are algebraic ipo's, its finite elements are those of the form $\langle d, e \rangle$ where $d \in E$, $e \in F$ and d and e are finite.

Now, suppose V, X, Y are in $\mathscr{P}[D]$, V is a finite element of $\mathscr{P}[D]$, and $V \sqsubseteq X \uplus Y$. Then $V = U^*$ where U is a finite set of finite elements of D. Then $U \sqsubseteq_M V \sqsubseteq_M X \uplus Y \sqsubseteq_M X \cup Y$.

We can then find nonempty sets U_1 and U_2 such that $U = U_1 \cup U_2$, $U_1 \sqsubseteq_M X$ and $U_2 \sqsubseteq_M Y$. But then $\langle U_1^*, U_2^* \rangle$ is a finite element of $\mathscr{P}[D] \times \mathscr{P}[D]$, by the above discussion of Cartesian product, $\langle U_1^*, U_2^* \rangle \sqsubseteq \langle X, Y \rangle$ and $V \sqsubseteq_M U = U_1 \cup U_2 \sqsubseteq_M U_1^* \uplus U_2^*$. By Lemma 5, $\uplus$ is continuous.

We may now see that any reasonable proof system for domains in SFP based on $\sqsubseteq$ may also be used to prove theorems about $\subseteq$ and $\in$ provided it has symbols and suitable axioms for $\cup$ and $[\![\cdot]\!]$. For $X \subseteq Y$ iff $X \cup Y = Y$ iff $X \uplus Y = \mathrm{Con}(Y) = Y$ $(X, Y \in \mathscr{P}[D])$ and $x \in X$ iff $[\![x]\!] \subseteq X$ $(x \in D, X \in \mathscr{P}[D])$. Further $\uplus$ is associative, commutative and idempotent. The notation $[\![x_1, \cdots, x_m]\!]$ will be useful—it abbreviates $[\![x_1]\!] \uplus \cdots \uplus [\![x_n]\!]$.

Big Union. Let D be an SFP object. The "big union" function, $\bigcup\!\!\!\!\bigcup: \mathscr{P}[\mathscr{P}[D]] \to \mathscr{P}[D]$ is defined by:

$$\bigcup\!\!\!\!\bigcup(\mathscr{X}) = \mathrm{Con}(\{x \in D \mid \exists X \in \mathscr{X}. x \in X\}) \qquad (\mathscr{X} \in \mathscr{P}[\mathscr{P}[D]]).$$

To see that $\bigcup\!\!\!\!\bigcup$ is well-defined we must prove that if $\mathscr{X} \in \mathscr{P}[\mathscr{P}[D]]$, then $Y = \{x \in D \mid \exists X \in \mathscr{X}. x \in X\}$ is closed in the Cantor topology. Let $\langle x_n \rangle_{n=0}^{\infty}$ be a convergent sequence in Y, with limit x. Each x_n is in some X_n in $\mathscr{X}$ and, without loss of generality, we may assume that $\langle X_n \rangle_{n=0}^{\infty}$ is convergent with limit X in $\mathscr{X}$. If x is in P_e, then almost all the x_n are in P_e and so almost all the X_n are in $P_{\{\perp, e\}^*}$. Therefore X too is in $P_{\{\perp, e\}^*}$ and so some element in X is in P_e. If $\langle e_n \rangle_{n=0}^{\infty}$ is an increasing sequence of finite elements whose l.u.b. is x we can therefore find a sequence $\langle y_n \rangle_{n=0}^{\infty}$ in X such that $y_n \supseteq e_n$. Taking a convergent subsequence we find an upper bound of x in X. Now suppose $x \in N_e$. Then so are almost all the x_n and so

almost all the X_n are in $N_{\{e\}}$. Therefore X too is in $N_{\{e\}}$ and so some element in X is in N_e. We now obtain a lower bound of x in X. Since X is convex, x itself is in X and so is in Y, showing that, as required, Y is closed.

Next we show that $\bigcup$ is monotonic. Suppose $\mathscr{X} \sqsubseteq \mathscr{X}'$ for $\mathscr{X}$, $\mathscr{X}'$ in $\mathscr{P}[\mathscr{P}[D]]$. Then $\mathscr{X} \sqsubseteq_M \mathscr{X}'$. Now if $x \in \bigcup\mathscr{X}$, then $x \in X$ for some $X \in \mathscr{X}$. Therefore $X \sqsubseteq_M X'$ for some $X' \in \mathscr{X}'$. Therefore $x \sqsubseteq x'$ for some $x' \in X' \subseteq \bigcup\mathscr{X}'$. Similarly, if $x' \in \bigcup\mathscr{X}'$, then $x' \sqsupseteq x$ for some $x \in \bigcup\mathscr{X}$. Therefore $\bigcup\mathscr{X} \sqsubseteq_M \bigcup\mathscr{X}'$ and so $\bigcup\mathscr{X} \sqsubseteq_M \bigcup\mathscr{X}'$.

For continuity, suppose $\mathscr{X} \in \mathscr{P}[\mathscr{P}[D]]$, $V \in \mathscr{P}[D]$, V is finite in $\mathscr{P}[D]$, and $V \sqsubseteq \bigcup\mathscr{X}$. Then $V = U^*$ where U is a finite set of finite elements of D. So $U \sqsubseteq_M \bigcup\mathscr{X}$. Let $\mathscr{X}' = \{Y^* | Y \subseteq U \text{ and } \exists X \in \mathscr{X}.\, Y \sqsubseteq_M X\}$. Then $\mathscr{X}'$ is a finite set of finite elements of $\mathscr{P}[D]$, $\mathscr{X}' \sqsubseteq_M \mathscr{X}$ and $U = \bigcup\mathscr{X}'$. Therefore, $V \sqsubseteq \bigcup(\mathscr{X}')^*$, $(\mathscr{X}')^*$ is finite in $\mathscr{P}[\mathscr{P}[D]]$, and $(\mathscr{X}')^* \sqsubseteq \mathscr{X}$. So by Lemma 5, $\bigcup$ is continuous.

It can be shown that if $\mathscr{X}$ is any nonempty subset of $\mathscr{P}[D]$, then $\bigcup\mathscr{X}^* = (\bigcup\mathscr{X})^*$. This fact should increase the intuitive appeal of certain definitions. It can be used to prove that $p^*q = \bigcup \circ \mathscr{P}[q_\perp] \circ p$, where * is the operation defined in § 2, which was used to give the denotational semantics of a simple nondeterministic language. The operator * is therefore continuous, as we shall see in § 7 that $\mathscr{P}$ acts continuously on functions.

Cartesian Product. Let D and E be SFP objects. We shall see below that $D \times E$ is also an SFP object.

Define $\otimes : \mathscr{P}[D] \times \mathscr{P}[E] \to \mathscr{P}[D \times E]$ by:

$$\otimes(X, Y) = X \times Y \qquad (X \in \mathscr{P}[D],\ Y \in \mathscr{P}[E]).$$

We will leave most of the details to the reader. First $\otimes$ is well-defined for it is not hard to show that if $X \in \mathscr{P}[D]$ and $Y \in \mathscr{P}[E]$, then $X \times Y \in \mathscr{P}[D \times E]$. If $X, X' \in \mathscr{P}[D]$, $Y, Y' \in \mathscr{P}[E]$, $X \sqsubseteq_M X'$ and $Y \sqsubseteq_M Y'$, then $X \times Y \sqsubseteq_M X' \times Y'$. Therefore $\otimes$ is monotonic.

For continuity the essential observation is this. Suppose U is a finite set of finite elements of $D \times E$ and $U \sqsubseteq_M X \times Y$, where $X \in \mathscr{P}[D]$, $Y \in \mathscr{P}[E]$. Let $(U)_1 = \{(u)_1 | u \in U\}$. For each x in X choose an element $d(x)$ in $\mathscr{U}(\{u \in (U)_1 | u \sqsubseteq x\})$ such that $d(x) \sqsubseteq x$. Set $U_1 = \{d(x) | x \in X\}$. Define U_2 similarly. Then $U \sqsubseteq_M U_1 \times U_2$.

The Cartesian product function has a useful application. Suppose $f : D_1 \times \cdots \times D_n \to E$ where the D_i and E are SFP objects and we are using an iterated Cartesian product. Then we can define an extension of f to a continuous function $g : \mathscr{P}[D_1] \times \cdots \times \mathscr{P}[D_n] \to \mathscr{P}[E]$ by:

$$g(X_1, \cdots, X_n) = \tilde{f}(X_1 \otimes X_2 \otimes \cdots \otimes X_n),$$

where $\otimes$ is being used as an infix operator, associating to the left. This kind of extension allows us to define the general comprehension notation used in § 8.

Some other functions. We can define a weak analogue of $\cap$, if the SFP object D is a semi-lattice. In that case every set X has a greatest lower bound. $\sqcap X$ and $x \sqcap y$ is continuous in x and y. Then we could define $\cap : \mathscr{P}[D] \times \mathscr{P}[D] \to \mathscr{P}[D]$ by

$$\cap(X, Y) = \mathscr{P}[\sqcap](X \otimes Y) \qquad (X, Y \in \mathscr{P}[D]).$$

In this case $\sqcap$ itself can be regarded as a continuous function $\overline{\sqcap}: \mathscr{P}[D] \to D$ defined by:

$$\overline{\sqcap}(X) = \sqcap X \qquad (X \in \mathscr{P}[D]).$$

If D is a lattice, $\sqcup$ can be regarded as a continuous function $\overline{\sqcup}: \mathscr{P}[D] \to D$ defined by:

$$\overline{\sqcup}(X) = \sqcup X.$$

7. Solving recursive domain equations. We now consider how to solve domain equations involving $\mathscr{P}$. Enough category theory has been developed to allow a presentation along the lines of [13], [14], [20], [21]. One could also construct a universal domain along the lines of [16], and this will be discussed.

The category-theoretic approach casts $+$, $\times$, $\to$ and $\mathscr{P}$ as locally continuous, symmetric functors and looks for solutions to recursive domain equations as fixed-points of such functors. These can be found by an analogous method to that of the fixed-point theorem [17].

DEFINITION. A functor T; $(\text{IPO-P})^k \to (\text{IPO-P})((\text{SFP-P})^k \to (\text{SFP-P}))$ is *locally continuous* iff wherever $P_i \subseteq \text{Hom}\,(D_i, E_i)$ are directed sets of morphisms for $i = 1, k$ then:

$$T(\sqcup P_1, \cdots, \sqcup P_k) = \sqcup\{T(p_1, \cdots, p_k) | p_i \in P_i, 1 \leqq i \leqq k\},$$

the set on the right being directed.

DEFINITION. A functor $T: (\text{IPO-P})^k \to (\text{IPO-P})((\text{SFP-P})^k \to (\text{SFP-P}))$ is *symmetric* iff when p_i is in $\text{Hom}\,(D_i, E_i)$ for $i = 1, k$, $T(p_1^\dagger, \cdots, p_k^\dagger) = (T(p_1, \cdots, p_k))^\dagger$.

These properties are preserved under composition (of functors) and the projection and constant functors are all symmetric and locally continuous.

If $T: (\text{IPO-P})^k \to (\text{IPO-P})$ is symmetric, its restriction T_{PR} to $(\text{IPO-PR})^k$ can be considered to be in $(\text{IPO-PR})^k \to (\text{IPO-PR})$, for then, if $p_1, \cdots, p_k$ are projections:

$$\begin{aligned} T_{\text{PR}}(p_1, \cdots, p_k)^\dagger \circ T_{\text{PR}}(p_1, \cdots, p_k) &= T_{\text{PR}}(p_1^\dagger, \cdots, p_k^\dagger) \circ T(p_1, \cdots, p_k) \\ &= T_{\text{PR}}(p_1^\dagger \circ p_1, \cdots, p_k^\dagger \circ p_k) \\ &= T_{\text{PR}}(I, \cdots, I) \\ &= I. \end{aligned}$$

Similarly, $T_{\text{PR}}(p_1, \cdots, p_k) \circ T_{\text{PR}}(p_1, \cdots, p_k)^\dagger \sqsubseteq I$.

For simplicity we shall confuse T_{PR} and T when the context leaves the choice indifferent or makes clear which is intended. Similar remarks apply vis-a-vis SFP-P and SFP-PR.

Next we describe $+$, $\times$, $\to$ and $\mathscr{P}$ as functors. The first three will be locally continuous, symmetric functors from $(\text{IPO-P})^2$ to (IPO-P). We will see later that they cut down to functors from $(\text{SFP-P})^2$ to SFP-P.

We choose a separated sum for $+$. Given two ipo's D and E, $D+E$ is $\{\langle 1, d\rangle | d \in D\} \cup \{\perp\} \cup \{\langle 2, e\rangle | e \in E\}$ with the obvious ordering. If $f: D \to D'$ and

$g: E \to E'$ are continuous functions, define $f+g: D+E \to D'+E'$ by

$$f+g(x)=\begin{cases}\perp & (x=\perp),\\ \langle 1, f((x)_2)\rangle & (x\neq\perp, (x)_1=1),\\ \langle 2, g((x)_2)\rangle & (x\neq\perp, (x)_1=2).\end{cases}$$

Now if $p: D \to D'$, $q: E \to E'$ are IPO-P morphisms, we define $p+q = \langle p_1+q_1, p_2+q_2\rangle$. This defines + as a functor (IPO-P)$^2 \to$ (IPO-P).

The cartesian product of two ipo's D and E is $D \times E$ which is the usual product with the induced componentwise ordering.

If $f: D \to D'$, $g: E \to E'$ are functions, $f \times g: D \times E \to D' \times E'$ is given by: $f \times g(\langle d, e\rangle) = \langle f(d), g(e)\rangle$.

If $p: D \to D'$, $q: E \to E'$ are morphisms, $p \times q: D \times E \to D' \times E'$ is given by: $p \times q = \langle p_1 \times q_1, p_2 \times q_2\rangle$. This defines Cartesian product as a functor (IPO-P)$^2 \to$ (IPO-P).

Exponentiation is more interesting. Its action on objects is clear, but it is not defined in the same fashion on morphisms. Suppose $p: D \to D'$, $q: E \to E'$ are morphisms. Define $p \to q: \mathrm{Hom}((D \to E), (D' \to E'))$ by: $(p \to q)_1(f) = q_1 \circ f \circ p_2$ $(f \in (D \to E))(p \to q)_2(g) = q_2 \circ g \circ p_1$ $(g \in (D' \to E'))$. (See Fig. 2.)

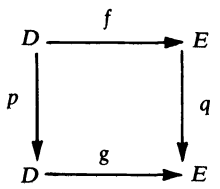


FIG. 2

This defines the exponentiation functor. The verification of symmetry and local continuity is straightforward for all three functors.

We have already seen that $\mathcal{P}$ can be taken as a functor from SFP to SFP. It induces a corresponding functor, which we also call $\mathcal{P}$, on SFP-P. Its action on objects is the same as $\mathcal{P}$. Let p be an SFP-P morphism. We define

$$\mathcal{P}[p] = \langle \mathcal{P}[p_1], \mathcal{P}[p_2]\rangle.$$

$\mathcal{P}$ is clearly symmetric. For continuity, we show that function extension preserves limits, too. This is clearly equivalent to showing $\mathrm{EXT}: (D \to E) \to (\mathcal{P}[D] \to \mathcal{P}[E])$ where $\mathrm{EXT}(f) = \hat{f}$ for $f: D \to E$. Now it is not hard to see that EXT is monotonic.

To prove continuity it is only necessary to take a directed set $F \subseteq (D \to E)$, a set X in $\mathcal{P}[D]$ and a map $g: D \to \mathbb{O}$ and prove that $g(\mathrm{EXT}(\bigsqcup F)(X)) \sqsubseteq_M g(\bigsqcup_{f \in F} \mathrm{EXT}(f)(X))$, since the other half follows from the monotonicity of EXT.

Now,

$$\text{LHS} = \hat{g}(((\widehat{\sqcup F})(X)))$$
$$= \{g \circ (\sqcup F)(x) | x \in X\}, \quad \text{and}$$

and

$$\text{RHS} = \hat{g}(\bigsqcup_{f \in F} (\hat{f}(X)))$$
$$= \bigsqcup_{f \in F} \hat{g}(\hat{f}(X))$$
$$= \bigsqcup_{f \in F} \{g \circ f(x) | x \in X\}.$$

If $\top \in \text{LHS}$, then $\top = g \circ (\sqcup F)(x)$ for some x in X. Therefore $\top = g \circ f(x)$ for some $f \in F$ and so $\top \in \text{RHS}$.

If $\bot \in \text{RHS}$, then $\forall f \in F. \exists x \in X. g \circ f(x) = \bot$. So $\forall n \geqq 0$, $f \in F. \exists x \in X. g \circ f \circ i_n \circ j_n(x) = \bot$. But $\{g \circ f \circ i_n \circ j_n | f \in F\}$ is both finite and directed. Therefore $\forall n \geqq 0. \exists x^{(n)} \in X. \forall f \in F. g \circ f \circ i_n \circ j_n(x^{(n)}) = \bot$. Without loss of generality we may assume that $\langle x^{(n)} \rangle_{n=0}^{\infty}$ converges to, say, x. Then $g \circ f(x) = \bot$ for all f in F and so $\bot \in \text{LHS}$. This finishes the proof of continuity of EXT.

We have now succeeded in exposing $+$, $\times$, $\to$ and $\mathcal{P}$ as examples of symmetric, locally continuous functors. Solving domain equations can be viewed as finding fixed points of such functors. For example suppose we want to find a domain R of resumptions which satisfies

$$R \cong S_\bot \to \mathcal{P}[S_\bot + (S_\bot \times R)].$$

Define a functor, T: (SFP-P) $\to$ (SFP-P) by

$$T(R) = S_\bot \to \mathcal{P}[S_\bot + (S_\bot \times R)]$$

on objects and similarly on morphisms.

Then we want to find a fixed point of T. We need a more global form of continuity.

Notation. Suppose $\mathcal{D}^i = \langle D^i_m, p^i_{mn} \rangle$ is a directed sequence when $1 \leqq i \leqq k$, and T: (IPO-P)$^k \to$ (IPO-P)((SFP-P)$^k \to$ (SFP-P)). Then we define $T(\mathcal{D}^1, \cdots, \mathcal{D}^k)$ to be the directed sequence $\langle T(D^1_m, \cdots, D^k_m), T(p^1_{mn}, \cdots, p^k_{mn}) \rangle$.

THEOREM 10 (Global Continuity). *Suppose* T: (IPO-P)$^k \to$ (IPO-P) *is symmetric and locally continuous. Let* $\mathcal{D}^1, \cdots, \mathcal{D}^k$ *be directed sequences in* IPO-PR. *Then* $T(\lim_{\to} \mathcal{D}^1, \cdots, \lim_{\to} \mathcal{D}^k) \cong \lim_{\to} T(\mathcal{D}^1, \cdots, \mathcal{D}^k)$ $(k > 0)$. *The analogous result holds for* SFP.

Proof. Let $\langle r^i_m \rangle$ be a universal cone from $\mathcal{D}^i$ to $\lim_{\to} \mathcal{D}^i$ $(1 \leqq i \leqq k)$. Then $\langle r_m \rangle = \langle T(r^1_m, \cdots, r^k_m) \rangle$ is a cone from $T(\mathcal{D}^1, \cdots, \mathcal{D}^k)$ to $T(\lim_{\to} \mathcal{D}^1, \cdots, \lim_{\to} \mathcal{D}^k)$. This is easily checked from the functor laws. To show that it is universal,

we use the criterion of Lemma 1 and show that $\bigsqcup_{n\geqq 0} r_n \circ r_n^{\intercal} = I$:

$$\begin{aligned}
&\bigsqcup_{n\geqq 0} T(r_n^1, \cdots, r_n^k) \circ T(r_n^1, \cdots, r_n^k)^{\dagger} \\
&\quad = \bigsqcup_{n\geqq 0} T(r_n^1 \circ r_n^{1\dagger}, \cdots, r_n^k \circ r_n^{k\dagger}) \\
&\quad = T\Big(\bigsqcup_{n\geqq 0} r_n^1 \circ r_n^{1\dagger}, \cdots, \bigsqcup_{n\geqq 0} r_n^k \circ r_n^{k\dagger}\Big) \qquad \text{(by local continuity)} \\
&\quad = T(I, \cdots, I) \qquad \text{(by Lemma 1 as the } \langle r_m^i \rangle \text{ are universal)} \\
&\quad = I.
\end{aligned}$$

The proof for SFP is similar. □

COROLLARY 2. *Suppose* $T: (\text{IPO-P})^k \to (\text{IPO-P})$ *is symmetric and locally continuous. If* $T(D_1, \cdots, D_k)$ *is an* SFP *object whenever* $D_1, \cdots, D_k$ *are finite, then* T *cuts down to a symmetric and locally continuous functor* $T_{\text{SP}}: (\text{SFP-P})^k \to (\text{SFP-P})$.

Proof. We need only show that if $D_1, \cdots, D_n$ are SFP objects, so is $T(D_1, \cdots, D_n)$. There are directed sequences, $\mathscr{D}^i$, of finite ipo's with limit D_i. Then $T(D_1, \cdots, D_n) = \lim_{\to} T(\mathscr{D}^1, \cdots, \mathscr{D}^k)$, by Theorem 10. Hence by Theorem 5(ii) and the hypothesis, $T(D_1, \cdots, D_n)$ is an SFP object. □

It follows that $+$, $\times$ and $\to$ cut down to symmetric, locally continuous functors from $(\text{SFP-P})^2$ to (SFP-P). This could have been proved directly, but in the case of $\to$ the details are rather tedious.

The next corollary enables us to solve recursive domain equations. It is partially analogous to the fixed-point theorem of [17], but does not give any "leastness" information. This, and much else, can be found in [21].

COROLLARY 3. *Let* $T: (\text{SFP-P}) \to (\text{SFP-P})$ *be symmetric and locally continuous. Then there is an* SFP *object* D *such that* $T(D) \cong D$.

Proof. Let $\mathscr{D} = \langle T^m(\mathbb{1}), p_{mn} \rangle$ where $\mathbb{1}$ is the one-point ipo. $p_{mn} = T^{n-1}(p_{\mathbb{1}}) \circ \cdots \circ T^m(p_{\mathbb{1}})$ $(m \leqq n)$, where $p_{\mathbb{1}}$ is the unique projection $p: \mathbb{1} \to T(\mathbb{1})$. Let $D = \lim_{\to} \mathscr{D}$. Then

$$\begin{aligned}
T(D) &\cong T(\lim_{\to} \mathscr{D}) \quad \text{(by Theorem 10)} \\
&\cong \lim_{\to} \mathscr{D} \quad \text{(since } T(\mathscr{D}) \text{ is obtained from } \mathscr{D} \text{ by dropping the first domain in } \mathscr{D}).
\end{aligned}$$

It is well-known that dropping a term of a sequence in this way does not affect the limit. □

When we want to solve simultaneous equations such as

$$D \cong N_{\perp} + \mathscr{P}[E] + D,$$

$$E \cong (N_{\perp} \to D) + E,$$

a slight extension of Theorem 8(i) to functors $T:(\text{SFP-P})^n \to (\text{SFP-P})^n$ is necessary. For example, in the case at hand one considers $T:(\text{SFP-P})^2 \to (\text{SFP-P})^2$ defined by $T(\langle D, E\rangle) = \langle N_\perp + \mathcal{P}[E] + D, (N_\perp \to D) + E\rangle$, etc.

An alternative approach to solving recursive equations is to follow the universal domain idea [16] and solve just one equation, $\mathcal{U} \cong N_\perp + (\mathcal{U} + \mathcal{U}) + (\mathcal{U} \times \mathcal{U}) + (\mathcal{U} \to \mathcal{U}) + \mathcal{P}[\mathcal{U}]$. Then we can represent subdomains by retractions. These are continuous functions $f:\mathcal{U} \to \mathcal{U}$ such that $f^2 = f$; f *represents* the ipo $\text{dom}(f) = \{d \in \mathcal{U} \mid f(d) = d\}$. As usual the retractions can be considered as elements of $\mathcal{U}$ and there are continuous functions $\oplus$, $\otimes$ and $\ominus$ such that if f and g are retractions, then so are $f \oplus g$, $f \otimes g$ and $f \ominus g$ and we have $\text{dom}(f \oplus g) \cong \text{dom}(f) + \text{dom}(g)$, $\text{dom}(f \otimes g) \cong \text{dom}(f) \times \text{dom}(g)$ and $\text{dom}(f \ominus g) \cong \text{dom}(f) \to \text{dom}(g)$.

There is also a suitable function $\mathcal{P}$ for $\mathcal{P}$ defined by:

$$\mathcal{P}(f) = \varphi_{\mathcal{P}} \circ \mathcal{P}[f] \circ \psi_{\mathcal{P}}.$$

Here $\langle \varphi_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ is the evident projection of $\mathcal{U}$ onto $\mathcal{P}[\mathcal{U}]$, EXT is the function extension defined above and we have followed the usual practice and identified $N_\perp$, $\mathcal{U} + \mathcal{U}$, $\mathcal{U} \times \mathcal{U}$, $\mathcal{U} \to \mathcal{U}$ and $\mathcal{P}[\mathcal{U}]$ as subdomains of $\mathcal{U}$. One then finds solutions to the recursion equations by solving the corresponding equations using the fixed point theorem. For example, to solve $P \cong V \to \mathcal{P}[L \times V \times P]$, where V and L are represented by r and l, respectively, one defines a continuous function f on $\mathcal{U}$ by $f = \lambda p \in \mathcal{U}.(r \ominus (\mathcal{P}(l \otimes v \otimes p)))$. Its least fixed point, p, is a retraction and represents a domain satisfying the equation. Simultaneous equations can also be solved in this way.

We will not pause to spell out the details of this approach. It should be noted that in general $\text{dom}(f)$ is *not* algebraic and hence not an SFP object. Rather it is, presumably, a continuous ipo in an appropriate sense. Thus this approach should lead to stronger results as it allows the possibility of a powerdomain construction on certain continuous objects. Extension of our direct approach, linking algebraic ipo's to continuous ones should also give these results.

Questions relating to $\mathcal{P}(\omega)$ also arise. By Theorem 1.6 of [17], we can embed any SFP object in $\mathcal{P}(\omega)$, as described in § 5, and this gives rise to a lattice with intermediate points as remarked at the beginning of this section. But these intermediate points seem to clutter up the domain and we do not know any simple continuous function analogous to $\mathcal{P}$ defined above. Of course we do know some such continuous function since $\mathcal{U}$ will be embeddable in $\mathcal{P}(\omega)$ and $\mathcal{P}(\omega)$ is embeddable in $\mathcal{U}$—indeed it can be embedded in $\mathcal{P}[N_\perp]$. But, rather than tagging $\mathcal{P}(\omega)$ on at the end, as it were, what is wanted is a simple development of $\mathcal{P}[\,\cdot\,]$ in the context of $\mathcal{P}(\omega)$ or a similar "simple" structure. In Scott's words, we want an analytic, not a synthetic development. However, we have at least developed the powerdomain construction enough to apply it to give some semantics as promised in the introduction.

8. Applications. We conclude by first giving the semantics for our illustrative language with simple parallelism and then giving an oracle-free semantics for Milner's multiprocessing language.

The programs of the first language are described by the grammar:

$$\pi ::= (\nu := \tau) | (\pi_1; \pi_2) | (\pi_1 \textit{ or } \pi_2) | (\textit{if } \nu \textit{ then } \pi_1 \textit{ else } \pi_2)$$
$$| (\textit{while } \nu \textit{ do } \pi) | (\pi_1 \textit{ par } \pi_2),$$

where ν and τ are as described in § 2.

The semantic domain of resumptions R is constructed, as described in the previous section, to satisfy the equation

$$R \cong S_\perp \to \mathcal{P}[S_\perp + (S_\perp \times R)].$$

To avoid being pedantic, we will identify R with the RHS and regard $S_\perp$, $(S_\perp \times R)$ as subsets of $S_\perp + (S_\perp \times R)$. The discriminator function $(:S_\perp): S_\perp + (S_\perp \times R) \to \mathbb{T}$ is defined by:

$$(x : S_\perp) = \begin{cases} \perp & (x = \perp), \\ \textit{true} & (x \in S_\perp), \\ \textit{false} & (x \in (S_\perp \times R)). \end{cases} \qquad (x \in S_\perp + (S_\perp \times R))$$

Suppose $---x---$ and $\sim\sim y \sim\sim$ are expressions possibly having occurrences of variables x and y ranging over D and E, respectively, such that $\sim\sim y \sim\sim$ is continuous in y and so defines a continuous function $f : E \to \mathcal{P}[D]$ and $---x---$ is continuous in x and so defines a continuous function $g : D \to F$, where D, E, F are SFP objects. Then the expression,

$$[\![---x--- | x \in \sim\sim y \sim\sim]\!]$$

is taken as defining $p(e)$ where the function $p : E \to \mathcal{P}[F]$ is $\mathcal{P}[g] \circ f$ when y has value e. With this notation we can easily describe various helpful operations on resumptions. It can be extended to several elements on the right of the $|$.

Choice. $r ? r' = \lambda\sigma \in S_\perp . r(\sigma) \uplus r'(\sigma)$, $(r, r' \in R)$.

Sequence. $r^* r' = \lambda\sigma \in S_\perp . [\![\text{COND } (x : S_\perp, \langle x, r' \rangle, \langle (x)_1, ((x)_2^* r') \rangle) | x \in r(\sigma)]\!]$.

Parallelism.

$$r /\!\!/ r' = \lambda\sigma \in S_\perp [\![\text{COND } (x : S_\perp, \langle x, r' \rangle, \langle (x)_1, r' /\!\!/ (x)_2 \rangle) | x \in r(\sigma)]\!]$$
$$\uplus [\![\text{COND } (x : S_\perp, \langle x, r \rangle, \langle (x)_1, (x)_2 /\!\!/ r \rangle) | x \in r'(\sigma)]\!].$$

These recursive definitions are to be taken as shorthand versions of explicit definitions using the least fixed-point operation. The choice combinator is associative, commutative and idempotent; the sequence combinator is associative and the parallelism combinator is associative and commutative, but not idempotent in general.

We have used a slightly different conditional combinator than in § 2, namely, the combinator $\text{COND} : \mathbb{T} \times D \times D \to E$ defined by

$$\text{COND}(t, x, y) = \begin{cases} \perp & (t = \perp), \\ x & (t = \textit{true}), \\ y & (t = \textit{false}), \end{cases} \qquad (t \in \mathbb{T}, x, y \in D).$$

Strictly speaking we should write COND_D, but both here and later D will be understood from the context.

With the aid of the function $\mathscr{V}$ described in § 2 we can define the denotational semantics, $\mathfrak{N}$: Statements $\to R$ by structural induction on Statements:

$$\mathfrak{N}[\![(x_i := \tau)]\!] = \lambda\sigma \in S_\perp.\mathrm{COND}\,(EQ(\sigma, \sigma), [\![\langle x_1, \cdots, x_{i-1}, \mathscr{V}[\![\tau]\!](\sigma), x_{i+1}, \cdots, x_n\rangle]\!], \perp),$$

$$\mathfrak{N}[\![(\pi_1; \pi_2)]\!] = \mathfrak{N}[\![\pi_1]\!]^*\mathfrak{N}[\![\pi_2]\!],$$

$$\mathfrak{N}[\![(\pi_1 \textit{ or } \pi_2)]\!] = \mathfrak{N}[\![\pi_1]\!] ? \mathfrak{N}[\![\pi_2]\!],$$

$$\mathfrak{N}[\![(\textit{if } x_i \textit{ then } \pi_1 \textit{ else } \pi_2)]\!] = \lambda\sigma \in S_\perp.\mathrm{COND}\,(EQ((\sigma)_i, 0), \mathfrak{N}[\![\pi_1]\!](\sigma), \mathfrak{N}[\![\pi_2]\!](\sigma)),$$

$$\mathfrak{N}[\![(\textit{while } x_i \textit{ do } \pi)]\!] = Y(\lambda r \in R, \lambda\sigma \in S_\perp.\mathrm{COND}\,(EQ((\sigma)_i, 0), \mathfrak{N}[\![\pi]\!]^* r(\sigma), [\![\sigma]\!])),$$

$$\mathfrak{N}[\![(\pi_1 \textit{ par } \pi_2)]\!] = \mathfrak{N}[\![\pi_1]\!] /\!/ \mathfrak{N}[\![\pi_2]\!].$$

The predicate $EQ : D^2 \to \mathbb{T}$ used above is defined for discrete domains D by:

$$EQ(x, y) = \begin{cases} \perp & (x = \perp \text{ or } y = \perp), \\ \textit{true} & (x = y \neq \perp), \\ \textit{false} & (x \neq \perp, y \neq \perp, x \neq y), \end{cases} \qquad (x, y \in D).$$

Various ad hoc possibilities are available to deal with the fairness problem. For example one could define a parallelism combinator $/\!/_m$ which is like $/\!/$ except that it does not run a branch for more than m elementary operations.

This combinator is commutative but not associative; its use is equivalent to a local use of a nondeterministic oracle. It will, presumably, not give rise to a fully abstract semantics. One can also look at other cases, such as buffers [8] which need not even give rise to nondeterminism. Perhaps one could achieve some workable combination of nondeterminism, oracles and special cases, but we feel that some new insight will be needed.

Let us conclude with a semantics for Milner's language. We give only an abbreviated account here, following the general pattern laid down in the papers [10], [11]. Some inessential variations have been made for consistency's sake.

The language has *identifiers* with metavariable x and a class of *expressions* with metavariable ε. The expressions are given by:

$$\varepsilon ::= x \mid \varepsilon_1(\varepsilon_2) \mid (\varepsilon_1; \varepsilon_2) \mid (\pi x.\varepsilon) \mid (\textit{let rec } x \textit{ be } \varepsilon_1 \textit{ in } \varepsilon_2) \mid (\textit{let slave } x \textit{ be } \varepsilon_1 \textit{ in } \varepsilon_2) \mid (\varepsilon_1 \textit{ or } \varepsilon_2) \mid (\varepsilon_1 \textit{ par } \varepsilon_2) \mid (\textit{if } \varepsilon_0 \textit{ then } \varepsilon_1 \textit{ else } \varepsilon_2) \mid (\textit{while } \varepsilon_1 \textit{ do } \varepsilon_2) \mid (\varepsilon_1 \textit{ renew } \varepsilon_2)$$

The semantic domains (SFP objects) comprise basic values, B, which is not specified further, addresses L (a discrete ipo), $\mathbb{T}$, nondeterministic processes P and pairs of values W. The fundamental domain equations are:

$$V \cong B + L + \mathbb{T} + P + W,$$

$$W \cong V \times V,$$

$$P \cong V \to \mathscr{P}[L \times V \times P].$$

The solution is obtained as described above; the equation for V should be thought of as employing a generalized (five-way) separated sum rather than an iterated binary separated sum. We have five injection functions of B, L, etc., into V. These are all called "*in V*", and used in a postfix fashion. There are also five postfixed discriminator functions $: B$, $: L$, etc., defined similarly to the function $: S$ used above.

Finally there are five postfixed projection functions from V onto $B, L \cdots$ named $| B, | L \cdots$ where, for example,

$$(v|B) = \begin{cases} b, & v = (b \text{ in } V), \\ \perp & \text{(otherwise)}. \end{cases}$$

We have also the Cartesian pairing and tripling functions $\langle -, - \rangle$ and $\langle -, -, - \rangle$ and projection functions $(-)_1, (-)_2, (-)_3$; for convenience the Cartesian product in the equation for P is to be regarded as employing the generalized product.

In L there are two distinct addresses, ζ and ν; the latter is intended to address a process for generating a sequence of distinct addresses, also distinct from ζ and ν. In B there is a special value whose injection into V is denoted by "!".

One useful combinator is $K : D \to (E \to D)$, defined by $Kxy = y$. Here D and E are arbitrary domains which should really appear as suffices. Another is the least fixed-point combinator $Y : (D \to D) \to D$. Two more specialized ones are ID, in P, and QUOTE in $V \to P$ defined by:

$$\text{ID} = \lambda v \in V.[\![\langle \iota, v, \perp \rangle]\!],$$

$$\text{QUOTE} = \lambda v \in V.K(\text{ID } v).$$

Now we need various functions on processes.

Conditional. DECIDE in $P \to P \to P$ is defined by:

$$\text{DECIDE } pq = \lambda v \in V.\text{COND } ((v|\mathbb{T}), p!, q!).$$

Extension. EXTEND in $P \to (P \to P) \to P$ is defined recursively by:

$$\text{EXTEND } pf = \lambda v \in V.\biguplus[\![\text{COND } (EQ(t_1, \iota), ft_3t_2, [\![\langle t_1, t_2, \text{EXTEND } t_3 f \rangle]\!] | t \in pv]\!].$$

Serial Composition. $*$ in $P \to P \to P$ is defined by:

$$p*q = \text{EXTEND } p(Kq).$$

Choice. ? in $P \to P \to P$ is defined by

$$p \ ? \ q = \lambda v \in V.pv \uplus qv.$$

Parallel Composition. $/\!/$ in $P \to P \to P$ is defined recursively by:

$$\begin{aligned}(p/\!/q) = \lambda v \in V.\biguplus[\![&\text{COND } (EQ(s_1, \zeta), (q*\lambda u \in V.\text{ID}(\langle s_2, u \rangle \text{ in } V)) \\ &(v|W)_2, \text{COND } (EQ(t_1, \iota), (p*\lambda u \in V.\text{ID } (\langle u, t_2 \rangle \text{ in } V))(v|W)_1, \\ &[\![\langle s_1, s_2, \lambda u \in V.(s_3/\!/q)(\langle u, (v|W)_2 \rangle \text{ in } V) \rangle, \\ &\langle t_1, t_2, \lambda u \in V.(p/\!/t_3)\langle (v|W)_1, u \rangle \text{ in } V \rangle \rangle]\!])) \\ &|s \in p(v|W)_1, t \in q(v|W)_2]\!].\end{aligned}$$

Here we have used the extension of the $[\![\ |\ ]\!]$ notation mentioned above to several variables.

Renewal. : in $P \to P \to P$ defined by:

$$p : q = p^* \lambda v \in V.[\![\langle \iota, v, q \rangle]\!]$$

Binding. BIND in $L \to P \to P \to P$ defined recursively by:

$$\text{BIND } \alpha pq = \lambda v \in V. \biguplus [\![\text{COND } (EQ(t_1, \alpha), \text{EXTEND } q \text{ (BIND } \alpha t_3) t_2, [\![\langle t_1, t_2, \text{BIND } \alpha t_3 q \rangle]\!] | t \in pv]\!].$$

The domain of environments is Env = (Identifiers $\to P$). It is ranged over by r. Although Identifiers is a set, Env is an SFP object if given the pointwise ordering. We denote by $r[p/x]$ the environment r' differing from r only in that $r[\![x]\!] = p$. The semantic function, $\mathscr{E}$: Expressions $\to$ (Env $\to P$) can now be defined by structural induction on expressions by:

$$\mathscr{E}[\![x]\!]r = r[\![x]\!],$$

$$\mathscr{E}[\![\varepsilon_1(\varepsilon_2)]\!]r = (\mathscr{E}[\![\varepsilon_1]\!]r^*(\lambda v \in V.\mathscr{E}[\![\varepsilon_2]\!]r^* \text{ COND } (v : L, \lambda u \in V.[\![\langle v|L, u, ID\rangle]\!], v|P)))!,$$

$$\mathscr{E}[\![(\varepsilon_1; \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_1]\!]r^* \mathscr{E}[\![\varepsilon_2]\!]r,$$

$$\mathscr{E}[\![(\pi x.\varepsilon)]\!]r = \text{QUOTE } ((\lambda v \in V.\mathscr{E}[\![\varepsilon]\!]r[(\text{QUOTE } v)/x]!) \textit{ in } V),$$

$$\mathscr{E}[\![(\textit{let rec } x \textit{ be } \varepsilon_1 \textit{ in } \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_2]\!](Y(\lambda r' \in \text{ENV}.r[\mathscr{E}[\![\varepsilon_1]\!]r'/x])),$$

$$\mathscr{E}[\![(\textit{let slave } x \textit{ be } \varepsilon_1 \textit{ in } \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_1]\!]r^*(\lambda u \in V.[\![\langle \nu, !, (\lambda v \in V.\text{BIND } (v|L) (\mathscr{E}[\![\varepsilon_2]\!]r[(\text{QUOTE } v)/x])(u|P)!)\rangle]\!]).$$

$$\mathscr{E}[\![(\varepsilon_1 \textit{ or } \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_1]\!]r ? \mathscr{E}[\![\varepsilon_2]\!]r,$$

$$\mathscr{E}[\![(\varepsilon_1 \textit{ par } \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_1]\!]r /\!/ \mathscr{E}[\![\varepsilon_2]\!]r,$$

$$\mathscr{E}[\![(\textit{if } \varepsilon_0 \textit{ then } \varepsilon_1 \textit{ else } \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_0]\!]r^* \text{ DECIDE } (\mathscr{E}[\![\varepsilon_1]\!]r)(\mathscr{E}[\![\varepsilon_2]\!]r),$$

$$\mathscr{E}[\![(\textit{while } \varepsilon_1 \textit{ do } \varepsilon_2)]\!]r = Y(\lambda p \in P.\mathscr{E}[\![\varepsilon_1]\!]r^* \text{ DECIDE } (\mathscr{E}[\![\varepsilon_2]\!]p^*r)\text{ID}),$$

$$\mathscr{E}[\![(\varepsilon_1 \textit{ renew } \varepsilon_2)]\!]r = \mathscr{E}[\![\varepsilon_2]\!]r^*(\lambda v \in V.(\mathscr{E}[\![\varepsilon_1]\!]r : (v|P))!).$$

Now the denotational semantics, $\mathscr{D}$: Expressions $\to$ (Env $\to P$) is obtained by binding in an address-generating process GEN as mentioned above. Thus:

$$\mathscr{D}[\![\varepsilon]\!] = \lambda v \in \text{Env. (BIND } \nu(\mathscr{E}[\![\varepsilon]\!]r_0) \text{ GEN)},$$

where r_0 is a suitable standard environment; $\mathscr{D}[\![\varepsilon]\!]$ will be a process which does not interrogate any addresses, provided none of the processes assigned by r_0 do.

In [11], Milner considered a different binding combinator which involved a domain $Q \cong Q \to P$. It is straightforward to adapt that to the present context.

Acknowledgments. I would like to thank Robin Milner for his part in the development of this work. He contributed not only stimulation, but also general philosophy and concrete ideas. I have also profited from some conversations with Gilles Kahn, and from the many helpful suggestions of the referees. The research was carried out under the direction of Dr. R. M. Burstall.

REFERENCES

[1] J. W. DE BAKKER, *Recursive Procedures*, Mathematical Centre Tracts 24, Math. Centre, Amsterdam, 1971.

[2] J. W. DE BAKKER AND W. P. DE ROEVER, *A calculus for recursive program schemes*, Automata, Languages and Programming, M. Nivat, ed., North-Holland/American Elsevier; New York, 1972.

[3] G. BIRKHOFF, *What can lattices do for you? Trends in Lattice Theory*, J. C. Abbott, ed., Van Nostrand, New York, 1970.

[4] N. BOURBAKI, *Elements of Mathematics: General Topology*, part 1, Addison-Wesley, Reading, Mass., 1966.

[5] J. M. CADIOU AND J. J. LEVY, *Mechanizable proofs about parallel processes*, Proc. 14th Annual Symposium on Switching and Automata Theory, 1973, p. 34.

[6] P. M. COHN, *Universal Algebra*, Harper and Row, New York, 1965.

[7] P. HITCHCOCK AND D. M. R. PARK, *Induction rules and termination proofs*, Automata, Languages and Programming, M. Nivat, ed., North-Holland/American Elsevier, New York, 1972.

[8] G. KAHN, *The semantics of a simple language for parallel programming*, Proceedings of IFIP Congress '74, Stockholm, Sweden, North-Holland, Amsterdam, 1974.

[9] Z. MANNA, *The correctness of non-deterministic programs*, Artificial Intelligence, 1 (1970), pp. 1–26.

[10] R. MILNER, *An approach to the semantics of parallel programs*, Proceedings of the Convegno di Informatica Teorica, Instituto di Elaborazione delle Informazione, Pisa, 1973.

[11] ———, *Processes: A mathematical model of computing agents*, Logic Colloquium, '73, North-Holland, Amsterdam, 1973.

[12] P. MOSSES, *The mathematical semantics of Algol 60*, Programming Research Group Technical Monograph PRG-12, University of Oxford, 1974.

[13] G. D. PLOTKIN, *LCF considered as a programming language*, Theoret. Computer Sci., 1977, to appear.

[14] J. C. REYNOLDS, *Notes on a lattice-theoretic approach to the theory of computation*, Dept. Systems and Information Science, Syracuse University, Syracuse, N.Y., 1972.

[15] ———, *On the relation between direct and continuation semantics*, Automata, Languages and Programming—2nd Colloquium, Univ. of Saarbrücken, Lecture Notes, in Computer Science, vol. 14, Springer-Verlag, Berlin, 1974.

[16] D. SCOTT, *Continuous lattices*, Toposes, Algebraic Geometry and Logic, F. W. Lawvere, ed., Springer-Verlag Lecture Notes, vol. 274, Springer-Verlag, Berlin, 1970.

[17] ———, *Data types as lattices*, Logic Conference, Kiel 1974, A. Dold and B. Eckmann, eds., Springer-Verlag Lecture Notes, vol. 499, Springer-Verlag, Berlin, 1975.

[18] D. SCOTT AND C. STRACHEY, *Towards a mathematical semantics for computer languages*, Proceedings of the Symposium in Computers and Automata, Microwave Research Institute Symposia Series, vol. 21, 1971.

[19] C. P. WADSWORTH, *The relation between computational and denotational properties for Scott's D_∞-models of the lambda calculus*, this Journal, 5 (1976), pp. 488–521.

[20] M. WAND, *On the recursive specification of data types*, Category Theory Applied to Computation and Control, Lecture Notes in Computer Science, vol. 25, Springer-Verlag, Berlin, 1974.

[21] ———, *Fixed-point constructions in order-enriched categories*, Tech. Rep. 23, Computer Science Dept., Indiana University, Bloomington, Indiana, 1975.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

THE RELATION BETWEEN COMPUTATIONAL AND DENOTATIONAL PROPERTIES FOR SCOTT'S D_∞-MODELS OF THE LAMBDA-CALCULUS*

CHRISTOPHER P. WADSWORTH†

Abstract. A prominent feature of the lattice-theoretic approach to the theory of computation due to D. Scott is the construction of solutions for isomorphic domain equations. One of the simplest of these is a domain isomorphic to the space of all continuous functions from itself to itself, providing the first "mathematical" model for the lambda-calculus of Church and Curry.

However, solutions of such domain equations are not unique; in particular, the lambda-calculus has many models. So the question arises as to which one should choose for computational purposes. We consider the relation between equivalence of meaning in Scott's models and the usual notions of conversion and reduction. By extending the lambda-calculus to allow approximate (i.e., partially specified) expressions and approximate reductions, we show that every expression determines a set of approximate normal forms of which it is the limit in Scott's model. Two immediate corollaries give a characterization of those expressions whose value is the least element of the model and further justification for the result that various lambda-calculus fixed-point combinators are all equal to the lattice-theoretic least fixed-point operator.

We show also that this leads to a characterization of equivalence which has a natural counterpart for other languages; specifically, expressions have the same meaning in Scott's model just when either can serve in place of the other in any "program" without altering its "global" properties.

Key words. lambda-calculus, lattices, isomorphic domain equations, projections, theory of computation, denotational semantics, equivalence, termination, head normal form, approximations, approximate normal form, incompleteness, fixed-point operators

Introduction. The purpose of this paper is to give an overview of recent results about the λ-calculus models discovered by Scott [17] in 1969, and to discuss how they reflect, and can be interpreted in terms of, the assumptions underlying the lattice-theoretic approach to the theory of computation. A few of the longer, more technical proofs have been omitted to make the development more readable and will be given in forthcoming papers [23], [24].

Ultimately our study concerns the acceptability of (some) language definitions based on the lattice-theoretic approach. This theory provides solutions for *isomorphic domain equations*, which are needed to define denotational semantics for many programming languages. However, solutions of such domain equations are not unique, and several different construction methods are now available. As a result, languages defined in this way may have many possible semantics and the question arises as to which of these is appropriate for reasoning about programs written in the language.

* Received by the editors March 13, 1975, and in revised form November 23, 1975.

† Department of Systems and Information Science, Syracuse University, Syracuse, New York. Now at Department of Computer Science, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland. This work was supported by National Science Foundation under Grant GJ-41540 and previously by a U.K. Science Research Council Grant to C. Strachey, Programming Research Group, Oxford University Computing Laboratory, Oxford, England.

To answer this question requires consideration of the relation between a proposed semantics for a language and the computational behavior of its programs. We present our study as an *example* of investigations of this kind. For our purposes, we can think of the λ-calculus as a programming language, albeit a restricted one, whose denotational semantics is provided by the lattice-theoretic models. The models we study are based on solutions of

$$\mathbf{D} \cong [\mathbf{D} \to \mathbf{D}] \tag{1}$$

where $[\mathbf{D} \to \mathbf{D}]$ denotes some suitable notion of function space from a domain $\mathbf{D}$ to itself. This is one of the simplest examples of an isomorphic domain equation and was historically the first for which a solution, called $\mathbf{D}_\infty$ by Scott, was found. We explore the relation between equivalence of meaning in these models and the usual notions of conversion and reduction, which can be regarded as (part of) a possible implementation.[1]

At the same time, the conversion rules provide a *proof theory* for the λ-calculus, so questions of completeness arise. We shall see that we do not have, and cannot expect, completeness in the usual sense. More precisely, two expressions are said to be interconvertible when they can both be reduced to a common expression (not necessarily irreducible); then although interconvertible expressions will be equivalent in all the possible semantics we are considering, for each possible semantics in general there will also be noninterconvertible expressions which are equivalent. For our analysis, the problem is that those examples which we would like to regard as equivalent are independent of any semantics whereas the set of noninterconvertible expressions having the same meaning varies as the semantics varies. From this point of view, the reasonableness of a semantics becomes the question of which semantics induces the "correct" equivalences between noninterconvertible expressions. We shall see that Scott's $\mathbf{D}_\infty$-model exhibits an interesting "limit completeness" property which leads to a characterization of equivalence of meaning in $\mathbf{D}_\infty$ having an obvious and natural counterpart for programming languages.

Our investigation is not the first example of its kind. Though the technicalities show few similarities, a good example for comparison is provided by studies of recursive function calculi, beginning with Kleene and more recently in work of, e.g., Cadiou [6] and Rosen [16]. In these calculi one is interested in definitions of functions by recursion, such as

$$f(n) = \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } f(n-1) + 2n - 1. \tag{2}$$

Under a *semantic* explanation of recursion, (2) is regarded as an equation to be solved for f; as such it may have a unique solution, or many solutions, or no solutions at all. Definition (2) can also be explained *algorithmically* via a collection

[1] In practice, of course, reductions are often simulated by representing λ-expressions and other intermediate items of interest in various kinds of data structures—stacks, symbol tables, pointers, return links, etc.—so the idea of reduction is already an abstraction from "the real world". However, it may serve as a useful intermediary by deriving, independently of the models studied in this paper, relations between the results of evaluation-by-reduction and the computations of actual λ-calculus machines, several of which are described in the literature (e.g., in [10], where further references also may be found).

of calculation rules (the "copy-rule", simplification rules for conditionals, arithmetic, etc.) for transforming expressions; a set of ordered pairs $\langle n, m\rangle$ of integers may then be called a *computed function* of (2) if $f(\bar{n})$ can be transformed to $\bar{m}$ by some sequence of applications of the rules, where $\bar{n}$ and $\bar{m}$ denote numerals corresponding to the integers n and m, respectively. Under suitable restrictions on the language, it can be shown that equations such as (2) can always be solved uniformly—there is always a *least-defined* solution—and that the rules can be chosen and applied so that the algorithmically computed function agrees with this solution. This is well-known as the first recursion theorem of Kleene [9, § 66, Thm. 26]. The results in § 5 below provide a λ-calculus analogue of this result; indeed, when applied to the special case of several λ-calculus fixed-point operators they are essentially the same result (see also Morris [11]). Our study is thus exactly the same in nature as these earlier ones; the novel feature, as we have indicated, is that we consider here a language whose semantics requires solutions of domain equations.

After a quick review of the λ-calculus in § 1, we present its semantics in § 2 for arbitrary solutions, as complete lattices, of the isomorphism (1). In § 3 we extract the properties of Scott's particular solution, $\mathbf{D}_\infty$, we shall use and give some examples. Section 4 departs from the main study of models to introduce the equivalent notions of solvability and head normal form which play a vital role in interpreting the later results and understanding their proofs. In § 5 we generalize the ordinary notion of reduction to study "approximations" and develop their "limit" properties, which lead in § 6 to the mentioned characterization of equivalence of meaning in $\mathbf{D}_\infty$. Section 7 then discusses one of the more surprising properties of $\mathbf{D}_\infty$—the possibility of equivalences between normal forms and expressions with no normal form—and its implications.

Though much of the development is highly technical, nevertheless the results provide some new insights which should apply equally to more expressive and more realistic languages; some indications will be given in the conclusion.

1. The λ-calculus. We assume familiary with the basic theory as presented, e.g., in [7], but give a brief summary to fix our notation and terminology.

We assume denumerably many *variables* $x, y, z, \cdots, x', y', \cdots$, and define the set of well-formed expressions, called *terms*, inductively as follows:

1. Every variable is a term.
2. If M and N are terms, so is the *combination* (MN), the parts M and N being called its *rator* and *rand*, respectively.
3. If x is a variable and M is a term, then the *abstraction* $(\lambda x.M)$ is a term, the parts x and M being called its *bv* and *body*, respectively.

We shall use $\equiv$ for syntactic identity of terms and, to reduce the proliferation of parentheses, we adopt the conventions that

(a) omitted parentheses in combinations associate to the left, e.g.,

$$xyzw \equiv (((xy)z)w),$$

(b) the scope of the dot " . " in an abstraction extends as far to the right as possible, i.e., to the first unmatched ") " or to the end of the term if that occurs first, and

(c) consecutive abstractions may be collapsed to a single one, e.g.,

$$\lambda xyz.M \equiv (\lambda x.\ (\lambda y.\ (\lambda z.M)))$$

An occurrence of a variable x in a term M is *free* if it is not inside the body M' of some part of M of the form $\lambda x.M'$, and *bound* otherwise. FV(M) denotes the set of variables occurring free in M. M is *closed* if FV(M) is empty, otherwise M is *open*; closed terms are sometimes also called *combinators*.

The notion of a *context* is the "dual" of the notion of a subterm and is useful for a uniform treatment of results for both open and closed terms. A context, $\mathbf{C}[\ ]$, consists of all parts of a term except that one subterm is missing (indicated by the empty brackets); $\mathbf{C}[M]$ then denotes the result of filling the missing subterm with M. Thus, the notation $\mathbf{C}[M]$ distinguishes a particular occurrence of M as a subterm. Note that a statement that terms A, B differ only in an occurrence of M, N, respectively, as a subterm, can then be expressed as the existence of a context $\mathbf{C}[\ ]$ such that $A \equiv \mathbf{C}[M]$ and $B \equiv \mathbf{C}[N]$. We shall see several such uses of contexts below when we compare the applicative properties of terms and their substitution instances. Many contexts we meet will be *head contexts*, of the general form

$$(\lambda x_1 x_2 \cdots x_n.[\ ])M_1 M_2 \cdots M_n A_1 A_2 \cdots A_m, \quad n \geqq 0, \quad m \geqq 0.$$

In particular examples, $x_1, x_2, \cdots, x_n$ will typically be a list of free variables of the terms being considered and $M_1, \cdots, M_n, A_1, \cdots, A_m$ will be closed terms; the choice of $M_1, \cdots, M_n$ determines corresponding substitution instances, and the choice of $A_1, \cdots, A_m$ prescribes arguments for application of these substitution instances as functions.

We shall write $[N/x]M$ for the *substitution* of N for (free occurrences of) x in M. We omit a formal definition but assume it is given so that bound variables in M are changed when necessary to prevent capture of free variables of N; e.g.,

$$[yz/x](\lambda y.x(\lambda x.xy)) \equiv \lambda w.yz(\lambda x.xw),$$

where w is some variable different from x, y, and z. This ensures that substitution is well-defined for *all* terms M and N; see [7, pp. 89–104] for a full discussion.

Terms are "evaluated" by eliminating abstractions as much as possible, according to two replacement rules and an auxiliary rule allowing renaming of bound variables:

1. *α-conversion*: Provided $y \notin \mathrm{FV}(M)$, a term of the form $\mathbf{C}[\lambda x.M]$ may be converted to $\mathbf{C}[\lambda y.M']$, where $M' \equiv [y/x]M$ and $\mathbf{C}[\ ]$ is any context.
2. *β-conversion*: A term of the form $R \equiv (\lambda x.M)N$ is called a *β-redex* and $R' \equiv [N/x]M$ is called its *contractum*. In any term, the operation of replacing an occurrence of R by R' is called a *β-contraction*; in the other direction, replacement of an occurrence of R' by R is called a *β-expansion* or *β-abstraction*. A sequence of (possibly zero) β-contractions is called a *β-reduction*, written $X\ \beta\textbf{-red}\ X'$; when β-abstractions may also be included in a sequence of replacements, the sequence is called a *β-conversion*, written $X\ \beta\textbf{-cnv}\ X'$.
3. *η-conversion*: A term of the form $\lambda x.Mx$, with $x \notin \mathrm{FV}(M)$, is called an *η-redex*, and then M is its contractum. The definitions of *η-contraction*, etc., are analogous to those for β-contraction, etc., in 2.

When it is of no interest which rules are being applied, we write simply X **red** X' or X **cnv** X'.

A term is said to be *in normal form* if it does not contain a redex as a subterm. Terms in normal form are said to be *distinct* if they are not α-interconvertible. A term N in normal form is said to be a *normal form of* M iff M **cnv** N. (Again, we may prefix β- and/or η- to these notions as appropriate; when used without a prefix, we always mean β-η-normal form.) It is readily seen that every term in normal form can be written in the form

$$\lambda x_1 x_2 \cdots x_n . z N_1 N_2 \cdots N_m, \qquad n \geqq 0, \quad m \geqq 0,$$

with each N_i also in normal form; conversely, every such term is in β-normal form (and in β-η-normal form if also $N_m \not\equiv x_n$ or if x_n occurs free in $zN_1N_2 \cdots N_{m-1}$).

Not all terms have a normal form. The two best-known examples are $\Delta\Delta$, where $\Delta \equiv \lambda x.xx$, and the so-called paradoxical combinator

$$Y_\lambda \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

A third example is the term

$$J \equiv Y_\lambda(\lambda f.\lambda x.\lambda y.x(fy))$$

which we shall see behaves very much like the identity $I \equiv \lambda x.x$.

For our discussion of interpretations and models, we summarize the theory as a formal system. As such the λ-calculus is an equational calculus; there is one predicate symbol "$=$" and the formulae consist of all equations $M = N$ between terms. The axioms and rules of inference are the usual ones for equality, plus the three conversion rules:

A1. $=$ is a *substitutive equivalence relation*:

(ρ) $\quad M = M$

(σ) $\quad \dfrac{M = N}{N = M}$

(τ) $\quad \dfrac{M = L,\ L = N}{M = N}$

(Subst) $\quad \dfrac{M = N}{\mathbf{C}[M] = \mathbf{C}[N]}$, for all contexts $\mathbf{C}[\ ]$.

A2. *conversion rules*:

(α) $\quad \lambda x.M = \lambda y.[y/x]M$, provided $y \notin \mathrm{FV}(M)$,

(β) $\quad (\lambda x.M)N = [N/x]M$,

(η) $\quad \lambda x.Mx = M$, provided $x \notin \mathrm{FV}(M)$.

Then, formally, M **cnv** N means that $M = N$ is provable from these axioms, and M **red** N means that $M = N$ is provable without the use of the symmetry rule (σ). Prefixes α-, β-, or η- on **cnv** or **red** denote probability without some of the conversion rules; e.g., M β-**red** N means that $M = N$ is provable without the use of (σ), (α) and (η).

Note that since $(\lambda x.Mx)X$ β-**red** MX, when $x \notin \mathrm{FV}(M)$, for all terms X, the η-rule is equivalent to the requirement of (functional) *extensionality*:

$$\text{(Ext)} \quad \frac{MX = NX \text{ for all } X}{M = N}$$

More essentially, however, the η-rule, when included, expresses a limitation on possible interpretations; it says that *all* terms can be regarded as functions (which, incidentally, we regard as equivalent if they have the same graph).

THEOREM 1.1 (The Church–Rosser Theorem). *If X* **cnv** *Y, there is a term Z such that X* **red** *Z and Y* **red** *Z. Hence, if a term has two normal forms X and Y, then X α-***cnv** *Y.*

THEOREM 1.2 (The Standardization Theorem). *If M has a normal form, then M can always be reduced to normal form by* normal-order reduction, *defined as the reduction in which each step is determined by contraction of the* leftmost *redex* (i.e., the redex whose left-hand end is furthest to the left).

Two terms M and N will be said to be *separable* iff there is a (head) context $\mathbf{C}[\ ]$ such that $\mathbf{C}[M]$ **cnv** $I \equiv \lambda x.x$ and $\mathbf{C}[N]$ **cnv** $K \equiv \lambda x.\lambda y.x$.

THEOREM 1.3 (Böhm [5]). *If M and N have distinct β-η-normal forms, then M and N are separable.*

Theorems 1.1 and 1.2 are well-known; in particular, Theorem 1.1 establishes the *consistency* of the λ-calculus system based on A1, A2. (In the absence of a negation operation, a formal system for the λ-calculus is said to be inconsistent if *all* equations between terms are provable, for which it suffices that $I = K$ is provable, since $II(KA)B$ β-**red** A and $KI(KA)B$ β-**red** B for all terms A and B.)

Theorem 1.3 is, for distinct normal forms, a form of converse of the Church–Rosser Theorem. The latter shows that distinct normal forms cannot be proved equal by the conversion rules; Theorem 1.3 shows that if one were ever to postulate, as an extra axiom, the equality of two distinct normal forms, the resulting system would be inconsistent. So the truth of equations between terms having a normal form is completely resolved by the theory of conversion—if any two such terms with distinct normal forms have the same value in a model, then all terms have the same value.

2. Lattice models of the λ-calculus. Because of the type-free style of application allowed in the λ-calculus, the domain of any interpretation must include (at least up to isomorphism) a significant portion of its own function space. In this section we consider interpretations based on arbitrary solutions of the isomorphism

$$\mathbf{D} \cong [\mathbf{D} \to \mathbf{D}] \tag{2.1}$$

with $\mathbf{D}$ a *complete lattice*[2] and $[\mathbf{D} \to \mathbf{D}]$ the lattice of *continuous* functions from $\mathbf{D}$ to $\mathbf{D}$ under the "pointwise" partial ordering. For a general orientation and the

[2] For those who prefer, directed complete partial orderings (partially ordered sets with a least element in which every directed subset has a least upper bound) may be used throughout without significantly affecting our development or results. However, we prefer to work with complete lattices for simplicity and ease of comparison to Scott's papers (though the more sophisticated notions associated with continuous lattices will not be used here).

definitions of monotonicity, directed set, continuous function, projections, etc., we refer the reader to the referenced papers of Scott. We shall use the symbols $\sqsubseteq$, $\bigsqcup$, $\bot$, and $\top$ to denote, respectively, the *partial ordering*, the *least upper bound* (l.u.b.) operation, the *least element*, and the *greatest element* of complete lattices.

Anticipating the notation of the particular solution to be studied later, we let $\mathbf{D}_\infty$ be *any* solution of (2.1), as a complete lattice, with more than one element, and we express the isomorphism by two (continuous) functions

$$\mathbf{D}_\infty \underset{\Psi}{\overset{\Phi}{\rightleftarrows}} [\mathbf{D}_\infty \to \mathbf{D}_\infty], \tag{2.2}$$

$$\Psi(\Phi(x)) = x, \quad \text{for all } x \in \mathbf{D}_\infty, \tag{2.3}$$

$$\Phi(\Psi(f)) = f, \quad \text{for all } f \in [\mathbf{D}_\infty \to \mathbf{D}_\infty]. \tag{2.4}$$

Let **VAR**, **EXP** and **ENV** denote the sets of all variables, terms and environments, respectively. By an environment, ρ, we mean an association of values in $\mathbf{D}_\infty$ with all variables, so $\rho : \mathbf{VAR} \to \mathbf{D}_\infty$ and **ENV** is the set of all functions from **VAR** to $\mathbf{D}_\infty$. The interpretation of variables is extended to all terms by a mapping

$$\mathscr{V} : \mathbf{EXP} \to [\mathbf{ENV} \to \mathbf{D}_\infty].$$

We shall use the emphatic brackets $[\![$ and $]\!]$ to enclose terms and write $\mathscr{V}[\![M]\!](\rho)$ for the *value*, or *denotation*, of M relative to the environment ρ. $\mathscr{V}$ is defined by structural induction, with one clause for each case in the syntax of terms:

(S1) $\mathscr{V}[\![x]\!](\rho) = \rho[\![x]\!]$,

(S2) $\mathscr{V}[\![MN]\!](\rho) = \Phi(\mathscr{V}[\![M]\!](\rho))(\mathscr{V}[\![N]\!](\rho))$,

(S3) $\mathscr{V}[\![\lambda x.M]\!](\rho) = \Psi(\lambda \mathbf{d} \in \mathbf{D}_\infty . \mathscr{V}[\![M]\!](\rho[\mathbf{d}/x]))$,

where $\rho[\mathbf{d}/x] \equiv \rho' \in \mathbf{ENV}$ is given by $\rho'[\![x]\!] = \mathbf{d}$ and $\rho'[\![x']\!] = \rho[\![x']\!]$ for $x' \not\equiv x$.

In (S1), the value of a variable x is ascertained by looking up its denotation in ρ, which, in view of the functional nature of ρ, is achieved by application of ρ to x. In (S2), the value of the rator M is interpreted as a function by application of Φ; this function is an element of $[\mathbf{D}_\infty \to \mathbf{D}_\infty]$, so can be meaningfully applied to the value of the rand N.

Clause (S3) is a little more involved. An abstraction $\lambda x.M$ is naturally interpreted as a function from $\mathbf{D}_\infty$ to $\mathbf{D}_\infty$. When applied to any argument $\mathbf{d} \in \mathbf{D}_\infty$, the result of this function is the value of the body M in the environment ρ' identical to ρ in all respects, except that the bound variable x is now associated with the argument $\mathbf{d}$. As $\mathbf{d}$ varies over $\mathbf{D}_\infty$, $\mathscr{V}[\![M]\!](\rho[\mathbf{d}/x])$ determines a continuous function, which is rendered as an element of $\mathbf{D}_\infty$ by the isomorphism Ψ. (That this function is continuous is a consequence of the continuity of the isomorphism pair Φ, Ψ and the fact that expressions fashioned out of variables and continuous functions by *typed* abstraction and *typed* application are continuous in all their free variables.)

It is important to notice the distinction between the language being defined and the notation used to define it (the meta-language). In particular, note the two different uses of the λ-notation in (S3); on the left is the symbol "λ" of the

type-free λ-calculus, on the right we have the typed λ-notation used to write down an expression for the meaning of the type-free notation. The second usage is convenient but is not essential to a presentation of the definition of $\mathscr{V}$; it can be avoided at the cost of introducing a name for the function occurring as the argument of Ψ (i.e., by writing:

$$\text{(S3')} \qquad \mathscr{V}[\![\lambda x.M]\!](\rho) = \Psi(f), \quad \text{where } f(\mathbf{d}) = \mathscr{V}[\![M]\!](\rho[\mathbf{d}/x]).$$

To establish that the above interpretation of terms gives a *model* for the λ-calculus, we must say when equations between terms hold in $\mathbf{D}_\infty$, and show that the axioms about equality and conversion are then satisfied. We define terms as being *semantically equivalent*, or *equal in* $\mathbf{D}_\infty$, when they have the same value for all associations of values with their free variables:

$$M =_{\mathbf{D}_\infty} N \quad \text{iff} \quad \mathscr{V}[\![M]\!](\rho) = \mathscr{V}[\![N]\!](\rho) \quad \text{for all } \rho \in \mathbf{ENV}.$$

That this gives a substitutive equivalence relation is immediate from the corresponding properties of equality of lattice elements. The validity of the conversion rules then follows from the properties (2.3) and (2.4) of the isomorphism, obvious results about $\mathscr{V}[\![M]\!](\rho)$ being independent of values in ρ for variables not occurring free in M, and a preliminary *substitution lemma*: *For all terms* M *and* N, *variables* x, *and environments* ρ,

$$\mathscr{V}[\![[N/x]M]\!](\rho) = \mathscr{V}[\![M]\!](\rho[\mathscr{V}[\![N]\!](\rho)/x]).$$

The latter asserts that extending environments models substitution correctly; the proof is a tedious but straightforward induction on the structure of M.

In outline we have proved

THEOREM 2.1. *The interpretation* (S1)–(S3) *and the relation* $=_{\mathbf{D}_\infty}$ *provide a model for the* λ*-calculus system based on* α-β-η*-conversion*

$$M\ \alpha\text{-}\beta\text{-}\eta\text{-}\mathbf{cnv}\ N \qquad \textit{implies} \qquad M =_{\mathbf{D}_\infty} N.$$

Since we shall always have in mind the fixed interpretation above, it is convenient now to simplify our notation by

(i) allowing the terms themselves to stand for their values in $\mathbf{D}_\infty$, and

(ii) identifying elements of $\mathbf{D}_\infty$ with their image under the isomorphism Φ, Ψ.

For closed terms the ambiguity in (i) is in any case not very great. Using the semantic clauses (S1)–(S3) to fully expand $\mathscr{V}[\![M]\!](\rho)$, we obtain an expression (of the meta-language) which is independent of ρ, e.g.,

$$\begin{aligned}\mathscr{V}[\![\lambda x.(\lambda y.yx)x]\!](\rho) &= \Psi(\lambda \mathbf{d}_0 \in \mathbf{D}_\infty.\Phi(\Psi(\lambda \mathbf{d}_1 \in \mathbf{D}_\infty.\Phi(\mathbf{d}_1)(\mathbf{d}_0)))(\mathbf{d}_0)) \\ &= \lambda \mathbf{d}_0 \in \mathbf{D}_\infty.(\lambda \mathbf{d}_1 \in \mathbf{D}_\infty.\mathbf{d}_1\mathbf{d}_0)\mathbf{d}_0\end{aligned}$$

by property (2.4) and the identification convention (ii) for Φ, Ψ. Apart from a change of bound variable names, this last expression differs from the original term only in the type indications for its bound variables. If M does contain free variables, the ambiguity is only slightly greater; a similar expansion then shows that $\mathscr{V}[\![M]\!](\rho)$ depends only on the values of these free variables in the environment ρ.

Thus the use of the terms themselves to denote their values is sufficiently precise if we say that

1. free variables are regarded as ranging over $\mathbf{D}_\infty$,
2. when a term is applied as a function we understand that its image under Φ is intended,
3. when we write an abstraction $\lambda x.M$ we understand that x is restricted to ranging over $\mathbf{D}_\infty$, and by the abstraction we really mean the image of the corresponding function under Ψ.

By virtue of Theorem 2.1, the conversion rules preserve values so we may use them to manipulate terms used in this way as expressions for elements of $\mathbf{D}_\infty$. Whenever we do need to be careful about distinguishing between terms and their values, or between elements of $\mathbf{D}_\infty$ and their image in $[\mathbf{D}_\infty \to \mathbf{D}_\infty]$, or for emphatic reasons, we can fall back on the more formal notation and write in all the appropriate $\mathscr{V}, \rho, \Phi, \Psi$, etc., but most of the time we can tolerate the ambiguity.

In general the converse of the implication in Theorem 2.1 cannot hold for *arbitrary* solutions of the isomorphism (2.2), but for terms having a normal form there is a positive result. From the observations at the end of § 1, we know that distinct normal forms must have distinct values provided the model is nontrivial (not all terms have the same value). The latter is immediate from $I \neq_{\mathbf{D}_\infty} K$. (Otherwise, since $I(Kx)y$ β-**red** x and $KI(Kx)y$ β-**red** y, we would have $x = y$ for all $x, y \in \mathbf{D}_\infty$, contradicting the assumption that $\mathbf{D}_\infty$ contains more than one element.)

With $\mathbf{D}_\infty$ being a lattice rather than a set a slightly stronger result holds. Just as lattice equality determined an equivalence relation between terms, so $\sqsubseteq$ induces a *quasi-ordering*:

$$M \sqsubseteq N \quad \text{iff} \quad \mathscr{V}[\![M]\!](\rho) \sqsubseteq \mathscr{V}[\![N]\!](\rho) \quad \text{for all } \rho \in \mathbf{ENV}.$$

(As a relation between terms, $\sqsubseteq$ fails to be a partial ordering only in that $M \sqsubseteq N \sqsubseteq M$ implies that M and N have the same value but not that they are the same term.) The relation $\sqsubseteq$ is easily seen to be substitutive:

$$M \sqsubseteq N \quad \text{implies} \quad C[M] \sqsubseteq \mathbf{C}[N] \quad \text{for all contexts } \mathbf{C}[\ ].$$

Then, since the terms I and K are incomparable under $\sqsubseteq$ (otherwise, we would have the same contradiction as above), it follows that all pairs of separable terms are incomparable; in particular:

THEOREM 2.2. *If M and N have distinct β-η-normal forms, then M and N are incomparable under $\sqsubseteq$.*

For terms without normal forms the situation is not so straightforward. Their properties are dependent on the deeper structure of particular models, and Theorem 2.2 may fail if either M or N fails to have a normal form.

Everything we have said so far holds for *all* models of the λ-calculus (or at least for all extensional lattice models), and we cannot expect the conversion rules to be complete for an arbitrary model. At the same time it is clear that we would not want completeness in this sense anyway. At first one's intuition might have been that a "natural model" for the λ-calculus is one in which terms have the same value just when they are interconvertible. However, the rules of conversion are only the *minimum* one should expect of an equational theory of "type-free"

functions and there are several respects in which they are deficient; for instance, they do not allow any inductive arguments. More immediately, we shall see examples of terms which are not interconvertible but which exhibit the same computational behavior. We would like to regard such terms as having the same meaning.

3. A particular model: The use of projections. We turn our attention now to the particular models discovered by Scott in 1969. First we give a brief outline of the construction in order to extract the properties of $\mathbf{D}_\infty$ we need in a convenient form (the laws of projection and application below).

Let $\mathbf{D}_0$ be any complete lattice and, inductively, $\mathbf{D}_{n+1} = [\mathbf{D}_n \to \mathbf{D}_n]$. The key to the construction lies in making this hierarchy of function spaces *cumulative* by "embedding" the lower-type spaces in the higher-type ones. These embeddings are described formally by a sequence of *projection pairs*

$$\mathbf{D}_n \underset{\psi_n}{\overset{\phi_n}{\rightleftarrows}} \mathbf{D}_{n+1}, \quad n = 0, 1, 2, \cdots,$$

$$\phi_{n+1}(x) = \phi_n \circ x \circ \psi_n, \qquad x \in \mathbf{D}_{n+1},$$

$$\psi_{n+1}(x') = \psi_n \circ x' \circ \phi_n, \qquad x' \in \mathbf{D}_{n+2}.$$

Then the *inverse limit* of the $\mathbf{D}_n$'s, i.e.,

$$\mathbf{D}_\infty = \{\langle x_n \rangle_{n=0}^{\infty} : x_n = \psi_n(x_{n+1}),\ x_n \in \mathbf{D}_n\}$$

is a complete lattice (under the "componentwise" partial ordering) and gives an isomorphism

$$\mathbf{D}_\infty \underset{\Psi}{\overset{\Phi}{\rightleftarrows}} [\mathbf{D}_\infty \to \mathbf{D}_\infty]. \tag{3.1}$$

There are two known "parameters" in this construction—the choice of the initial domain $\mathbf{D}_0$ and the choice of the initial projection pair ϕ_0, ψ_0—which can be varied and one still obtains a solution of (3.1). In this and later sections our results hold for *any* $\mathbf{D}_0$ with more than one element (we shall see in § 6 why, other than this, the choice of $\mathbf{D}_0$ doesn't matter) but with ϕ_0, ψ_0 *fixed* as Scott's original projection pair:

$$\phi_0(x) = \lambda y \in \mathbf{D}_0.x, \qquad x \in \mathbf{D}_0,$$

$$\psi_0(x') = x'(\bot_{\mathbf{D}_0}), \qquad x' \in \mathbf{D}_1.$$

In fact the construction does more than solve the isomorphism (3.1) for the solutions have an internal structure too. There are also projection pairs

$$\mathbf{D}_n \underset{\psi_{\infty n}}{\overset{\phi_{n\infty}}{\rightleftarrows}} \mathbf{D}_\infty$$

embedding each $\mathbf{D}_n$ as the subspace

$$\mathbf{D}_n^{(\infty)} = \{\phi_{n\infty}(x) : x \in \mathbf{D}_n\} \subseteq \mathbf{D}_\infty$$

and inducing projection functions

$$P_n \equiv \phi_{n\infty} \circ \psi_{\infty n} : \mathbf{D}_\infty \to \mathbf{D}_n^{(\infty)}.$$

We can now "forget" the finite-type spaces $\mathbf{D}_n$ used in the construction and work entirely within $\mathbf{D}_\infty$, with the subspaces $\mathbf{D}_n^{(\infty)}$ and the projections P_n. These projections satisfy various equations (e.g., see [20, p. 64] or [15, pp. 114–115]) and determine essentially all the structure of the resulting λ-calculus models. For this it is convenient to redefine the subscript notation to stand for the projections; we now write, for $x, y \in \mathbf{D}_\infty$ and $f \in [\mathbf{D}_\infty \to \mathbf{D}_\infty]$,

$$\begin{aligned} x(y) &\quad \text{for} \quad \Phi(x)(y), \\ f &\quad \text{for} \quad \Psi(f), \\ x_n &\quad \text{for} \quad P_n(x), \end{aligned}$$

Transliterating the properties of the projections P_n into this notation gives the following:

Laws of projection.

(P1) $x_n \sqsubseteq x_m \sqsubseteq x \quad \text{for } n \geqq m,$

(P2) $\bot_n = \bot \quad (\bot \text{ now denotes } \bot_{\mathbf{D}_\infty}),$

(P3) $x = \bigsqcup_{n=0}^{\infty} x_n,$

(P4) $(x_n)_m = x_{\min(n,m)},$

(P5) (Additivity) $(\bigsqcup X)_n = \bigsqcup \{x_n : x \in X\} \quad \text{for } X \subseteq \mathbf{D}_\infty.$

Laws of application.

(P6) (Extensionality) $x(z) = y(z) \text{ for all } z \in \mathbf{D}_\infty \leftrightarrow x = y,$
$x(z) \sqsubseteq y(z) \text{ for all } z \in \mathbf{D}_\infty \leftrightarrow x \sqsubseteq y,$

(P7) $x(y) = \bigsqcup_{n=0}^{\infty} x_{n+1}(y_n),$

(P8) $\bot(y) = \bot,$

(P9) $x_0(y) = x_0 = x(\bot)_0,$

(P10) $x_{n+1}(y) = x(y_n)_n.$

(Where parentheses are omitted, application takes precedence over projection; e.g., in (P10), $x(y_n)_n$ when fully parenthesized is $(x(y_n))_n$.)

Three derived laws which follow from the above are

(P11) $x_{n+1}(y) = x_{n+1}(y)_m \quad \text{for all } m \geqq n,$

(P12) $x_{n+1}(y) = x_{n+1}(y_m) \quad \text{for all } m \geqq n,$

(P13) $x = y \leftrightarrow x_n = y_n \quad \text{for all } n \geqq 0.$

Informally, we can read applications of these projection functions as entailing a *loss of information*, in the following sense. We have

$$\mathbf{D}_0^{(\infty)} \subseteq \mathbf{D}_1^{(\infty)} \subseteq \cdots \subseteq \mathbf{D}_n^{(\infty)} \subseteq \cdots \subseteq \mathbf{D}_\infty$$

and, for each $x \in \mathbf{D}_\infty$, the projections $x_n \in \mathbf{D}_n^{(\infty)}$ form an increasing sequence

$$x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots \sqsubseteq x$$

of *approximations* to x; moreover, since

$$x_n = \bigsqcup \{w : w \in \mathbf{D}_n^{(\infty)},\ w \sqsubseteq x\}$$

follows easily from the above, we can think of x_n as the *best* approximation to x available in the subspace $\mathbf{D}_n^{(\infty)}$. Properties (P3) and (P13) then state that every element of $\mathbf{D}_\infty$ is determined by these "finite" approximations.

In a similar way, (P7) states that application of an element of $\mathbf{D}_\infty$ as a function is determined by the functional behavior of its projections. Note, however, that (P7) does *not* imply that the terms in the l.u.b.-expression on the right are the *best* approximations $x(y)_n$ to the result of the application on the left (in general we only have $x_{n+1}(y_n) \sqsubseteq x(y)_n$), but only that their *join* gives the correct result. (Note also that (P7) is not a definition of application but a *derived property* satisfied by the projection functions together with the Φ-part of the isomorphism; in the strict notation (P7) reads

$$\text{(P7')} \qquad \Phi(x)(y) = \bigsqcup_{n=0}^{\infty} \Phi(P_{n+1}(x))(P_n(y)) \quad .)$$

Properties (P9)–(P12) tell us how the individual projections of elements behave as functions. Property (P9) states that 0th projections are constant functions on $\mathbf{D}_\infty$; this is a consequence of the choice of the constant-function embedding $\phi_0 : \mathbf{D}_0 \to \mathbf{D}_1$ given above. For $(n+1)$st projections, the definition of $\mathbf{D}_{n+1}^{(\infty)}$ as the embedding of $\mathbf{D}_{n+1} \equiv [\mathbf{D}_n \to \mathbf{D}_n]$ in $\mathbf{D}_\infty$ implies that elements of $\mathbf{D}_{n+1}^{(\infty)}$ are essentially functions from $\mathbf{D}_n^{(\infty)}$ to $\mathbf{D}_n^{(\infty)}$; that is, $\mathbf{D}_{n+1}^{(\infty)} \cong [\mathbf{D}_n^{(\infty)} \to \mathbf{D}_n^{(\infty)}]$. Thus, the application $x_{n+1}(y)$ will always give a result in the subspace $\mathbf{D}_n^{(\infty)}$ (property (P11) above) and is independent of information contained in its argument y which is not contained in the best approximation y_n to y in $\mathbf{D}_n^{(\infty)}$ (property (P12) above). Further, considering the expression $x(y_n)_n$ on the right of (P10) as an operation on y, x is being applied to the nth projection $y_n \in \mathbf{D}_n^{(\infty)}$ of the argument and the nth projection $x(y_n)_n \in \mathbf{D}_n^{(\infty)}$ of the result is being taken; but from what we just said, this is how the *best* approximation $x_{n+1} \in \mathbf{D}_{n+1}^{(\infty)} \cong [\mathbf{D}_n^{(\infty)} \to \mathbf{D}_n^{(\infty)}]$ acts, so the application $x_{n+1}(y)$ gives the same result (property (P10) above).

With these laws we can do many useful calculations about the values of terms in $\mathbf{D}_\infty$. Three typical examples are

THEOREM 3.1. (a) (Scott [20]) $\Delta\Delta = \bot$,
(b) (Park [13]) $Y_\lambda = Y$,
(c) $J =_{\mathbf{D}_\infty} I$,

where $Y : [\mathbf{D}_\infty \to \mathbf{D}_\infty] \to \mathbf{D}_\infty$ *defined by*

$$Y(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot)$$

is the lattice-theoretic least fixed-point operator *and the terms* Δ, Y_λ, *I and J are as given in* § 1.

Note how use has been made of the ambiguity between terms and their values in stating the theorem. In (a) we really mean that $\mathscr{V}[\![\Delta\Delta]\!](\rho) = \bot$ for all ρ. For (b),

from lattice theory we know that the function Y maps continuous functions to their least fixed-points. Y itself also is continuous, so, by virtue of the isomorphism, we can think of Y as an element of $\mathbf{D}_\infty$; that is, we can faithfully "identify" Y with the element $Y_\infty \in \mathbf{D}_\infty$ given by

$$Y_\infty \equiv \Psi(\lambda x \in \mathbf{D}_\infty . Y(\Phi(x))) = \Psi(Y \circ \Phi)$$

Then, properly stated, (b) reads: for all ρ, $\mathscr{V}[\![Y_\lambda]\!](\rho) = Y_\infty$.

Proof. We give the calculation for (b) as an illustration. Part (a) can be proved by a similar calculation, but notice it follows immediately from (b), since $\Delta\Delta$ **cnv** $Y_\lambda I$ and $\perp$ is clearly the least fixed-point of I.

For (b), since Y_λ is, by β-conversion, a fixed-point operator, we have $Y \sqsubseteq Y_\lambda$. For the converse, by extensionality it suffices to show that $Y_\lambda(f) \sqsubseteq Y(f)$ for arbitrary f. Let $X \equiv \lambda x.f(xx)$. Then it suffices to show that

$$X(X_n) \sqsubseteq f^{n+2}(\perp) \quad \text{for all } n \geqq 0, \tag{3.2}$$

for then

$$Y_\lambda(f) \textbf{ cnv } X(X) = X\Big(\bigsqcup_{n=0}^{\infty} X_n\Big) = \bigsqcup_{n=0}^{\infty} X(X_n), \quad \text{by (P3) and continuity,}$$

$$\sqsubseteq \bigsqcup_{n=0}^{\infty} f^{n+2}(\perp) = Y(f).$$

We prove (3.2) by induction on n. For $n = 0$, we have

$$X(X_0) = f(X_0(X_0)) = f(X_0) \sqsubseteq f(f(\perp))$$

by definition of X, (P9), and monotonicity, since $X_0 = X(\perp)_0 \sqsubseteq X(\perp) = f(\perp(\perp)) = f(\perp)$. For $n \geqq 0$, we have

$$X(X_{n+1}) = f(X_{n+1}(X_{n+1})) = f(X_{n+1}(X_n)), \qquad \text{by definition of } X \text{ and (P12),}$$

$$\sqsubseteq f(X(X_n)) \sqsubseteq f(f^{n+2}(\perp)),$$

$$\text{by } X_{n+1} \sqsubseteq X \text{ and induction hypothesis.}$$

For part (c), let $F \equiv \lambda f.\lambda x.\lambda y.x(fy)$, so $J \equiv Y_\lambda F$. That I is a fixed-point of F, and hence $J \sqsubseteq I$, is immediate since FI β-η-**red** I. We prove the stronger result:

THEOREM 3.2. *I is the only fixed-point of F in $\mathbf{D}_\infty$.*

Proof. Let J be any fixed-point of F, so J satisfies the equation

$$J(x)(y) = x(J(y)). \tag{3.3}$$

We prove inductively that $J_n = I_n$ for all $n \geqq 0$, for which, by extensionality, it suffices to prove that $J_n(x)(y) = I_n(x)(y)$ for arbitrary x, y.

For $n = 0$ we have, using (3.3), (P9), (P8) and (P2),

$$J_0(x)(y) = J(\perp)_0(y) = J(\perp)(\perp)_0 = \perp(J(\perp))_0 = \perp,$$

$$I_0(x)(y) = I(\perp)_0(y) = \perp_0(y) = \perp.$$

For $n = 1$ we have, using also (P10) with $n = 0$,

$$J_1(x)(y) = J(x_0)_0(y) = J(x_0)(\perp)_0 = x_0(J(\perp))_0 = x_0,$$

$$I_1(x)(y) = I(x_0)_0(y) = x_0(y) = x_0.$$

Now assume, inductively, that $J_{n+1} = I_{n+1}$ for some $n \geqq 0$. Then

$$\begin{aligned} J_{n+2}(x)(y) &= J(x_{n+1})_{n+1}(y) = J(x_{n+1})(y_n)_n \quad \text{by (P10) twice} \\ &= x_{n+1}(J(y_n))_n = x_{n+1}(J(y_n)_n) \quad \text{by (3.3), (P11, P12) with } m = n \\ &= x_{n+1}(y_n) \quad \text{since, using the induction hypothesis,} \\ &\qquad J(y_n)_n = J_{n+1}(y) = I_{n+1}(y) = I(y_n)_n = y_n \end{aligned}$$

$$I_{n+2}(x)(y) = I(x_{n+1})_{n+1}(y) = x_{n+1}(y) = x_{n+1}(y_n). \qquad \square$$

The second part of Theorem 3.1 extends to a whole sequence of λ-calculus fixed-point combinators, defined by

$$Y^{(0)} \equiv Y_\lambda, \qquad Y^{(i+1)} \equiv Y^{(i)}(G), \qquad i \geqq 0,$$

where $G \equiv \lambda y.\lambda f.f(yf)$. First, note the following intimate connection between G and arbitrary fixed-point operators:

LEMMA 3.3. (a) *In any extensional model of the λ-calculus, an element ∇ is a fixed-point operator iff ∇ is a fixed-point of G:*

$$\nabla f = f(\nabla f) \quad \textit{iff} \quad G\nabla = \nabla.$$

(b) *For the lattice-theoretic least fixed-point operator Y,*

$$G(Y) =_{\mathbf{D}_\infty} Y =_{\mathbf{D}_\infty} Y(G).$$

Proof. Part (a), and hence the first half of (b), is immediate from the definition of G by β-reduction and extensionality. For the second half of (b),

$$G^n(\bot)(f) = f^n(\bot), \qquad f \in \mathbf{D}_\infty, \qquad \text{for all } n \geqq 0,$$

is easily proved by induction on n. Hence, using the definition of $\sqcup$ on function spaces,

$$Y(G)(f) \equiv \left(\bigsqcup_{n=0}^{\infty} G^n(\bot)\right)(f) = \bigsqcup_{n=0}^{\infty} G^n(\bot)(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot) \equiv Y(f)$$

from which the result follows by extensionality. $\square$

(Similar to Theorem 3.2, it might now be asked whether Y is the only fixed-point of G in $\mathbf{D}_\infty$, but this fails—there are other (continuous) fixed-point operators on $\mathbf{D}_\infty$ besides Y. However, it turns out that Y is the only one which is λ-definable, because Y_λ is a maximal term under the ordering $\sqsubseteq$. (A proof of the latter will be included in [24].) In other words, it can be shown that *every* λ-calculus term having the fixed-point property has the least fixed-point operator Y as its value in $\mathbf{D}_\infty$. For the particular terms $Y^{(0)}, Y^{(1)}, \cdots$, however, we need not bother here with this interesting, but technically quite difficult, generalization.)

COROLLARY 3.4. *For all $i \geqq 0$, $Y^{(i)} =_{\mathbf{D}_\infty} Y$.*

Proof. The case $i = 0$ is just Theorem 3.1(b). If, inductively, $Y^{(i)} =_{\mathbf{D}_\infty} Y$, then

$$Y^{(i+1)} \equiv Y^{(i)}(G) =_{\mathbf{D}_\infty} Y(G) =_{\mathbf{D}_\infty} Y \qquad \square$$

Historically, the equivalence of these fixed-point combinators in $\mathbf{D}_\infty$, discovered originally by Park [13], provided the first example of the incompleteness

of the conversion rules for these models, for it was known previously [4, p. 195] that the first two, $Y^{(0)}$ and $Y^{(1)}$, are not interconvertible. The motivation for wishing to regard all these fixed-point combinators equivalent will be discussed at the end of § 5.

Part (c) of Theorem 3.1 is a more unexpected example of incompleteness, for it gives an example of a term without a normal form which is equivalent to one in normal form. We shall return to discuss this example further in § 7, where we shall see it is not so unreasonable as it seems at first sight. For now, we note that the phenomenon is not special to the particular normal form I (it would perhaps be even more surprising if it was!):

COROLLARY 3.5. *For any normal form N, there is a term X with no normal form such that $N =_{\mathbf{D}_\infty} X$.*

Proof. Choose an occurrence of a variable x in N which is not the rator of a combination. (Such an occurrence always exists, e.g., choose the rightmost occurrence of a variable in N.) Let X be the term obtained by replacing this occurrence of x by the term $J(x)$. Then $N =_{\mathbf{D}_\infty} X$, since $J(x) =_{\mathbf{D}_\infty} I(x)$ **cnv** x, and the replaced occurrence of x not being the rator of a combination is sufficient to guarantee that X does not have a normal form. □

4. Solvability and head normal forms. Classical studies of the λ-calculus generally emphasize the significance of normal forms and tend to divide the terms into two classes—those with normal forms, whose "values" are perfectly defined, and those with no normal form, which are regarded as "meaningless" or "undefined". Such a division is too "discrete" (indeed we shall find reason to doubt the semantic significance of the distinction at all). Instead, just as Scott argues for data objects in general, we must recognize varying degrees of definedness; the "value" of a term may be defined in some respects but not in others. More especially, we cannot consistently regard a term as undefined just because it fails to have a normal form; some can be used to give defined results in nontrivial ways, and in fact it would be inconsistent to identify *all* terms not having a normal form:

Example 1. Let A be any term without a normal form, and let $M \equiv \lambda x.xIA$ and $N \equiv \lambda x.xKA$. Both M and N fail to have a normal form, but MK β-**red** I and NK β-**red** K. Identifying M and N would therefore imply $I = K$, which would render the λ-calculus inconsistent.

The concepts of solvability and head normal form are the beginnings of our analysis of the semantic significance of terms. We think of them primarily as allowing distinctions to be drawn between terms without normal forms, but the notions apply equally to all terms. Solvability was first studied in connection with the λ-definability of partial recursive functions by Barendregt [2], [3], to which we refer the interested reader for a fuller discussion.

We consider first the closed terms and how they behave when applied as functions:

Example 2. Consider the terms

$$M_1 \equiv \Delta\Delta, \quad \text{where } \Delta \equiv \lambda x.xx,$$

$$M_2 \equiv \Delta T, \quad \text{where } T \equiv \lambda x.xxx,$$

$$M_3 \equiv \Delta D, \quad \text{where } D \equiv \lambda x.\lambda y.xx.$$

(In fact M_3 β-**red** $Y_\lambda K$.) These three terms, none of which have a normal form, are particularly "hereditarily undefined". No matter how they may be applied as functions their computations will continue indefinitely: for all $k \geqq 0$ and all terms $X_1, X_2, \cdots, X_k$, the applications $M_i X_1 X_2 \cdots X_k$ fail to have a normal form, for each of M_1, M_2, M_3 above.

On the other hand, not all terms without normal forms behave in this way as functions. For example, the terms of Example 1 do not, nor does the paradoxical combinator Y_λ or the term J given in § 1; e.g.,

$$Y_\lambda(KA) \quad \beta\textbf{-red} \quad A, \quad \text{for any term } A,$$

$$J(KA)B \quad \beta\textbf{-red} \quad A, \quad \text{for any terms } A, B.$$

We define a *closed* term M as being *solvable* iff there is an integer $k \geqq 0$ and terms $X_1, X_2, \cdots, X_k$ such that $MX_1X_2 \cdots X_k$ has a normal form.

Terms having a normal form are always solvable; indeed, more strongly:

LEMMA 4.1. *If M is a closed term having a normal form N and X is* any *term, there exist $k \geqq 0$ and terms $X_1, X_2, \cdots, X_k$ such that*

$$MX_1X_2 \cdots X_k \quad \beta\textbf{-red} \quad X \tag{4.1}$$

Proof. We can write the normal form N in the form $N \equiv \lambda x_1 x_2 \cdots x_n.zN_1N_2 \cdots N_m$. Since M, and therefore N, is closed, we have $z \equiv x_i$ for some i. Choosing $k = n$, $X_i \equiv \lambda y_1 y_2 \cdots y_m.X$, and $X_j \equiv I$ for $j \neq i$, the result follows. □

The following corollary is then straightforward from the property IX **red** X, and indicates the reason for the terminology "solvability":

COROLLARY 4.2. *A closed term M is solvable iff the equation* (4.1) *with $X \equiv I$ can be solved for $X_1, X_2, \cdots, X_k$ (for some $k \geqq 0$) iff for* all *terms X the equation* (4.1) *can be solved.*

For *open* terms we must consider also their substitution instances. For example, $x(\Delta\Delta)$ can never yield a normal form by being applied as a function, but its substitution instance $[KS/x](x(\Delta\Delta))$ is solvable provided S is solvable. We therefore define an open term as being solvable iff there is a substitution of closed terms for its free variables such that the resulting (closed) term is solvable. Or, uniformly for all terms M, M is *solvable* iff there is a *head* context $\mathbf{C}[\ ]$ such that $\mathbf{C}[M]$ has a normal form. Corollary 4.2 then extends to open terms in the same way.

The concept of head normal form provides a syntactical characterization of the solvable terms. First:

Example 2 (cont.). If we inspect the reductions of M_1, M_2, M_3 we find:

1. M_1 reduces only to itself.
2. Every term to which M_2 reduces is of the form $((\lambda x.xxx)T)TT \cdots T$.
3. Every term to which M_3 reduces is, after α-conversion, of the form $\lambda x_1 x_2 \cdots x_n.((\lambda x.\lambda y.xx)D)$.

These terms have the general form: either they consist of a β-redex, or of a β-redex followed by a finite number of arguments, or of a finite number of abstractions on terms of the latter form. That is, they are of the form

$$\lambda x_1 x_2 \cdots x_n.((\lambda x.M)N)X_1X_2 \cdots X_m, \qquad n \geqq 0, \quad m \geqq 0. \tag{4.2}$$

A term of the form (4.2) will be said to be *not in head normal form* and the redex $(\lambda x.M)N$ is then called its *head redex.* The reduction of a term in which each step is determined by contraction of the head redex (when it exists) will be called *head reduction.*[3] □

On the other hand, the terms of Example 1 are not of the form (4.2), nor are the terms

$$\lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))), \qquad \lambda x.\lambda y.x(Jy)$$

to which Y_λ and J, respectively, are reducible. None of these terms are in (or have) a normal form, but they are in *head normal form,* of the general form

$$\lambda x_1 x_2 \cdots x_n.zX_1X_2 \cdots X_m, \qquad n \geq 0, \quad m \geq 0, \quad z \text{ a variable}, \tag{4.3}$$

where $X_1, X_2, \cdots, X_m$ are arbitrary terms. (We leave it to the reader to show that every term can be written uniquely in one of the forms (4.2) or (4.3) for suitable n, m, X_i, etc.)

In (4.3) the variable z is known as the *head variable* and the term X_i as its ith *main argument.* Two head normal forms will be called *similar* iff they have the same head variable (after α-conversion, if necessary, so that bound variables agree) and the *difference* between the number of main arguments and the number of initial bound variables (i.e., $m-n$ in (4.3)) is the same for both. (The use of the difference here is a technical point connected with η-conversion. Most similar head normal forms we meet will be *strongly similar,* i.e., will have the same number of main arguments and the same number of initial bound variables. Note that in any case similar head normal forms can always be converted to strongly similar ones, by applying η-abstractions to the one with fewer initial bound variables.)

Analogous to normal-order reduction of terms to normal form, it can be shown that a term has a head normal form iff its head reduction terminates, and all head normal forms of a term are similar (strongly similar if η-conversions are excluded). (In fact, note that a head redex, when it exists, is also the leftmost β-redex, so the head reduction of a term consists of some initial segment of its normal-order reduction, or the whole of the latter if the term does not have a head normal form.)

It is instructive to compare (4.3) with the structure of normal forms given in § 1. In a head normal form (4.3) the main arguments $X_1, X_2, \cdots, X_m$ can be arbitrary terms, whereas a normal form requires these inner subterms also to be in normal form. Thus, a head normal form is a kind of "weak" normal form, which we might say is in normal form "at the first level".

It is easy to see that terms having a head normal form are solvable, for the proof of Lemma 4.1 uses only the fact that N is in head normal form. The converse can also be proved by syntactic means (by analyzing head reductions) but we shall see an easier model-theoretic proof in the next section.

[3] This definition of head reduction should not be confused with that of Curry [8, pp. 32, 157]. For a head redex, Curry's definition requires $n = 0$ above; here it is not appropriate to limit ourselves to this case, because the binding of free variables does not affect (un)solvability.

The upshot of this discussion is that only those terms without normal forms which are in fact unsolvable can be regarded as being "undefined" (or better now: "totally undefined"); by contrast, all other terms without normal forms are at least partially defined. Essentially the reason is that unsolvability is preserved by application and composition—if U is unsolvable, so are UX and $U \cdot X$ for all terms X—which we have seen is not true in general for the property of failing to have a normal form.

To say that (some) terms without normal forms have a value which is not undefined seems at first a little disturbing, for after all every reduction of any term without a normal form will fail to terminate. In fact it is our traditional conception of termination—the result of a program is defined if its execution terminates and undefined if not—which is at fault here, as nonterminating programs cannot always be regarded as being "totally undefined". Consider, for instance, a language with output statements, so that programs whether they terminate or not may be producing intermediate output, possibly even infinite output (e.g.,

let $n := 0$;
while *true* **do begin output** $Prime(n)$;
$n := n+1$ **end**

producing a list of the prime numbers). In such latter cases we might say the program computes its (infinitary) result "bit-by-bit" (the result of the program being the "union" of the partial outputs); then only those nonterminating programs which produce no intermediate output should be thought of as being "totally undefined".

For understanding the interpretation of λ-calculus terms in $\mathbf{D}_\infty$ this analogy with languages which allow intermediate output is a helpful one to have in mind, in the following sense. Consider "evaluating" a term M by reduction. If we find that M has a head normal form (4.3), we can output the "first-level" information about its initial bound variables, its head variable, and the number of its main arguments; then proceed recursively to evaluate (in parallel) the main arguments $X_1, X_2, \cdots, X_m$. If this process stops in finite time the original term M has a normal form and this is the information which will have been output. The process may fail to terminate in two ways: Either at some stage it may find a (sub)term without a head normal form, in which case the relevant component of the result is "totally undefined" as no information about it will ever be output, or the process may recurse indefinitely producing infinite output. (The latter would happen, e.g., for Y_λ and J.) For either case the total output gives, intuitively, *all* the information that can be computed about the "value" of the whole term M. These ideas harmonize very nicely with the general intuitions underlying Scott's approach to the theory of computation and will be pursued further in the next section.

So far we have considered only what happens when terms (or their substitution instances) are applied as functions. If functions are applied *to them*, then, even for an unsolvable term U as argument, we can always obtain, say, a normal form as result by using a constant function. But such usages are trivial as the use of any term in place of U would give the same result; more generally, we shall see that

unsolvable terms can never have a nontrivial effect on the outcome of a reduction (Corollary 5.5).

Thus the interpretation of unsolvable terms as being "least defined" is a good one. It will not be surprising therefore when we find that they are exactly the terms with value $\perp$ in the $\mathbf{D}_\infty$-model. It has been shown by Barendregt [2, p. 91] that it is consistent to identify all unsolvable terms (again, an easier proof will be seen below), and from a computational point of view it is desirable to do so as they behave so alike.

5. Approximate reduction. Our discussion so far has established models only for the equational part of the theory of the λ-calculus. In consequence we know that the process of reduction preserves meanings, but there is no direct interpretation of the reduction-relation itself in these models. In this section we shall see that none is needed, for the relation can be treated metatheoretically; that is, its properties will be seen to be implicit in the internal structure of the models.

We shall take up the idea of "evaluation-by-reduction" in a little more general formulation than at the end of the previous section, so as not to be dependent on any one method of reduction. Based on suggestions originally due to Scott,[4] we shall extend the ordinary λ-calculus to allow *partial terms* and *partial*, or *approximate*, *reductions*, then we can investigate *limits* of better and better approximate reductions and tie these in with the notion of limit already present in the lattice-models. The method is similar to the usual arguments justifying the use of *least* fixed-points in work on the theory of computation (cf. the comments on recursive function calculi in the Introduction and the treatment of **while**-statements in [19] or recursive procedures in [1]). The results will show that although the theory of conversion is too weak for proof purposes—not all true equations between terms which hold in $\mathbf{D}_\infty$ can be proved—there is a limiting sense in which the reduction rules are complete for purposes of evaluation.

We shall base our limiting process on β-reduction. The ideas can also be applied to η-reduction but we do not need to do so here as the latter plays rather a subsidiary role in evaluation; in any case if we can prove anything about limits using fewer approximations, the results remain true when more approximations are included.

Suppose we start reducing a term M, using any method and order of reduction we care to choose. If M has a normal form and the reduction actually reaches one, everything is fine and we can take the normal form obtained as the "value" of M. If not, conventionally the possibility is not considered of attributing a value to a reduction which apparently fails to terminate; one simple concludes that the initial term M does not appear to have a normal form. However, we need not be so pessimistic, for there is something that can be said, based on the intermediate terms in the reduction:

Example 3. Suppose M has been reduced to a term M', say

$$M' \equiv \lambda x.\lambda y.y(x(\underline{(\lambda z.P)Q}))(xy)(\underline{(\lambda w.R)S})$$

[4] In conversation, October 1970.

where P, Q, R, S are terms. At this stage we cannot tell whether M has a normal form or not, because there are still at least the two underlined redexes in M'. However, from the form of M', we can say that if M', and hence M, is going to have a normal form, this must be of the form

$$\lambda x.\lambda y.y(x(?))(xy)(?)$$

because any further (β)-reduction of M' can affect only the parts where we have written "?". Now adjoin to the λ-calculus the special constant symbol Ω to stand for such "undetermined" parts of normal forms. Then we shall say

$$A' \equiv \lambda x.\lambda y.y(x(\Omega))(xy)(\Omega)$$

is a *partial* normal form of M—partial because it tells us something of the structure of the normal form of M (if such exists), but does not give complete information. The obvious interpretation of Ω in the lattice-models is as the least element $\perp$:

(S4) $$\mathscr{V}[\![\Omega]\!](\rho) = \perp.$$

Then $\Omega \sqsubseteq X$ for all terms X, from which the substitutivity property gives $A' \sqsubseteq M'$ ($=M$). Thus, A' is certainly *one* approximation to M, and we shall now call it an *approximate normal form* of M. (Of course, it is possible that A' is actually equal to M in $\mathbf{D}_\infty$, if the subterms which have been replaced by Ω turn out to have value $\perp$, but in general this will not be the case.)

Clearly this idea of approximating any parts remaining to be evaluated (i.e., β-redexes) by Ω can be applied to all terms in all possible reductions of M. In this way we obtain a whole set of approximations, each giving some, in general incomplete, information about M. The obvious question is whether, passing to their values in $\mathbf{D}_\infty$, their join gives *all* the information "contained" in the value of the starting term M.

Notice that the answer is immediate for any term which has a (proper) normal form, for then its normal form is one of its approximate normal forms, and for some obvious terms with value $\perp$, e.g., for the term $\Delta\Delta$ which reduces only to itself so that Ω is its only approximate normal form. More informative is:

Example 4. All terms to which Y_λ reduces are of the form $\lambda f.f^n(XX)$, where $X \equiv \lambda x.f(xx)$ and $n \geqq 0$, so the approximate normal forms of Y_λ are $\{\lambda f.f^n(\Omega) : n \geqq 0\}$. Thus there is an exact parallel here with the terms in the usual l.u.b.-expression for the least fixed-point operator. (This of course was part of our motivation for studying approximate reduction for arbitrary terms.)

To formalize the above we first introduce some further terminology. Expressions of this λ-Ω-calculus will be called λ-Ω-*terms*, or simply terms when no confusion is likely. A term will be said to be *in approximate normal form* if it does not contain a β-redex, and in *proper* normal form if additionally it does not contain any occurrences of Ω. A term A will be called a *direct approximant* of a term M iff A is in approximate normal form and matches M except at occurrences of Ω in A, and will be called the *best* direct approximant of M if Ω occurs in A only at subterms corresponding to the (outermost) β-redexes in M. (In Example 3, A' was the best direct approximant of M'; other direct approximants are, e.g.,

$\lambda x.\lambda y.y(\Omega)(\Omega y)(\Omega)$ and $\lambda x.\lambda y.\Omega(xy)(\Omega)$.) A term A will be called an *approximate normal form of* M iff A is a direct approximant of a term to which M is reducible.

(In fact, these definitions are slightly wider than in our earlier discussion for they allow subterms larger than the outermost β-redexes to be replaced by Ω in forming direct approximants; but clearly every direct approximant is $\sqsubseteq$ the best one (hence the terminology), so use of the wider definition does not affect limits.)

From (S4) and the properties of $\perp$, it is immediate that two Ω-*conversion* rules are valid in $\mathbf{D}_\infty$:

$$(\Omega_1) \qquad \Omega X = \Omega,$$

$$(\Omega_2) \qquad \lambda x.\Omega = \Omega.$$

We leave it to the reader to show

LEMMA 5.1. *If* M β-**red** M' *and* B, B' *are the best direct approximants of* M, M', *respectively, then* $B \sqsubseteq B'$.

So the best approximants of successive terms in a reduction form an increasing sequence. Unfortunately for any particular reduction of a term M such a sequence may converge to a limit smaller than the value of M. (In particular, normal-order reduction is sometimes inadequate in this sense; consider $M \equiv \lambda x.x(\Delta\Delta)(R)$ where R is any β-redex with value different from $\perp$.)

Considering *all* reductions, however, remedies the deficiency. We shall write $\mathscr{A}(M)$ for the set of approximate normal forms of M. (It follows easily from Lemma 5.1 and the Church–Rosser Theorem that $\mathscr{A}(M)$ is always a directed set.) In the statement of the limit theorem which follows we write in the $\mathscr{V}$ and ρ to emphasize the distinction between syntactic and semantic concepts:

THEOREM 5.2. *For all terms* M *and environments* ρ,

$$\mathscr{V}[\![M]\!](\rho) = \bigsqcup \{\mathscr{V}[\![A]\!](\rho) : A \in \mathscr{A}(M)\}.$$

Proof. Unfortunately there is room only for a few hints of the proof here; the full proof will be given in [23]. The main step consists of the development of a formal *typed calculus* for carrying out the kind of calculations with projections we did informally in § 3, via the notion[5] of a *type-assignment* for terms appropriate to these models. The "types" are integers and a type-assignment consists of an association of (arbitrary) integers with *every* subterm of a term; the intended interpretation is that the corresponding projection of the value of the subterm is to be taken. The properties of projections, in particular (P3), and continuity then imply, by structural induction on M, that $\mathscr{V}[\![M]\!](\rho)$ is equal to the l.u.b. of (the values of) all such typed-terms representing type-assignments for M.

Corresponding to the properties (P9) and (P10) of projections, two forms of *typed* β*-reduction* can be defined and shown to preserve the values of terms with type-assignments. The "well-foundedness" of (P9) and (P10) (i.e., application of an $(n+1)$st projection always decreases the level of the projections involved by one, and application of a 0th projection allows the argument to be replaced by Ω without changing the result) can then be used to show that every typed-term T, representing a type-assignment for M, can be reduced to, and hence is equivalent

[5] Suggested to us by J. M. E. Hyland, personal communication, January 1972.

to, a typed-term T' in approximate normal form. The result then follows by showing that the untyped term corresponding to T' is always one of the approximate normal forms of M. □

With this theorem we can explain our remarks about the "limiting completeness" of the reduction rules. The rules are adequate for generating the (r.e.) set of terms to which M reduces. As these are generated, so more and more of the approximate normal forms of M are obtained. The theorem asserts that if we are prepared to generate enough of the terms to which M reduces, then we can calculate an expression whose value approximates the value of M as close as we like—intuitively, we can "compute" (canonical representations for) the values of terms in $\mathbf{D}_\infty$.

In another, related sense, the theorem can be regarded as asserting that every term has a "normal form"; it's just that this may be an infinite expression (as was hinted at toward the end of § 4). Several methods for specifying *infinite normal forms* have been suggested (e.g., see [12]); when done in the right way we should be able to say that terms have the same meaning (in $\mathbf{D}_\infty$) iff they have the same infinite normal form.

As one immediate consequence of Theorem 5.2, we obtain the following complete characterization of those terms which have value $\perp$ (for all environments) in these models:

COROLLARY 5.3. *The following three conditions are equivalent*:

(a) *M is unsolvable.*

(b) *M does not have a head normal form.*

(c) $M =_{\mathbf{D}_\infty} \Omega$.

Proof. We prove (c) implies (a) and (b) implies (c). That (a) implies (b), or equivalently that not-(b) implies not-(a), follows by a proof similar to that of Lemma 4.1.

For (c) implies (a), we give the proof for closed terms M. (The extension to terms with free variables—in which case, for solvability, one must consider closed substitution instances—is straightforward using the rule (Ω_2).) Suppose $M =_{\mathbf{D}_\infty} \Omega$ but M is solvable. Then there exist terms $X_1, X_2, \cdots, X_k$ $(k \geqq 0)$ such that $MX_1X_2 \cdots X_k$ **cnv** I. But then, by using (Ω_1), we have

$$I =_{\mathbf{D}_\infty} MX_1X_2 \cdots X_k =_{\mathbf{D}_\infty} \Omega X_1X_2 \cdots X_k =_{\mathbf{D}_\infty} \Omega,$$

which is a contradiction.

For (b) implies (c), suppose M does not have a head normal form, so every term M' to which M reduces is not in head normal form. For any term M' not in head normal form, all direct approximants of M' are of the form

$$A' \equiv \lambda x_1 x_2 \cdots x_n.\Omega X_1 X_2 \cdots X_m, \qquad n \geqq 0, \quad m \geqq 0. \tag{5.1}$$

But $A' =_{\mathbf{D}_\infty} \Omega$ by (Ω_1) and (Ω_2). Hence all approximate normal forms of M are equal to Ω in $\mathbf{D}_\infty$, so $M =_{\mathbf{D}_\infty} \Omega$ by Theorem 5.2. □

The last two results are pleasing *semantic* properties of the models of § 3. They are quite natural in themselves and desirable from our earlier discussion, but we can find additional technical support via several formal (syntactic) results relating terms and their approximate normal forms when used as parts of larger expressions. In particular we have:

THEOREM 5.4. *For all terms M and contexts* $\mathbf{C}[\ ]$,

(a) $\mathbf{C}[M]$ *has a normal form iff* $\mathbf{C}[A]$ *has the same normal form for some* $A \in \mathscr{A}(M)$.

(b) $\mathbf{C}[M]$ *has a head normal form iff* $\mathbf{C}[A]$ *has a similar head normal form for some* $A \in \mathscr{A}(M)$.

Proof. That the normal forms in (a) (head normal forms in (b)) will be the same (similar) follows from the separability of distinct normal forms (dissimilar head normal forms). (For dissimilar head normal forms, this is straightforward; for distinct normal forms, see Theorem 1.3.) Part (a) is then proved by arguments based on the relative lengths of the normal-order reductions of terms and their direct approximants; we leave the full proof to [23]. Part (b) can be proved by analogous arguments about the lengths of various head reductions, but there is an alternative proof using our results above. By Corollary 5.3, it suffices to show

$$\mathbf{C}[M] \neq_{\mathbf{D}_\infty} \Omega \quad \text{iff} \quad \mathbf{C}[A] \neq_{\mathbf{D}_\infty} \Omega \quad \text{for some } A \in \mathscr{A}(M).$$

But this follows easily from continuity and Theorem 5.2. □

COROLLARY 5.5. *Suppose U is unsolvable and* $\mathbf{C}[\ ]$ *is any context. Then* $\mathbf{C}[U]$ *has a normal form (a head normal form) iff* $\mathbf{C}[M]$ *has the same normal form (a similar head normal form) for all terms M.*

Proof. The "if" part is trivial. We give the proof of the "only if" part for normal forms. Suppose $\mathbf{C}[U]$ has a normal form N. By Theorem 5.4(a), there is an approximate normal form A' of U such that $\mathbf{C}[A']$ **red** N. Since U is unsolvable, A' must be of the form (5.1) in the proof of Corollary 5.3. Hence

$$\mathbf{C}[\lambda x_1 x_2 \cdots x_n.\Omega X_1 X_2 \cdots X_m] \quad \mathbf{red} \quad N.$$

It is easily shown that normal-order reductions to (proper) normal form are not affected by applications of Ω-conversion or replacements of Ω by arbitrary terms, from which the result follows. □

Intuitively, Corollary 5.5 shows why unsolvable terms should be regarded as being "least defined". Theorem 5.4 shows that the "interesting" computational behavior of terms is determined by their approximate normal forms, so Theorem 5.2 can be interpreted as showing that the values of terms in $\mathbf{D}_\infty$ "contain" just the information relevant to their use in computations.

In fact, Theorem 5.4 can be extended to apply also to the *approximate* normal forms of $\mathbf{C}[M]$. In the case of a function application FM (i.e., taking $\mathbf{C}[\ ] \equiv F[\ ]$ so $\mathbf{C}[M] \equiv FM$) where F is any term, this specializes to: *For all* $A' \in \mathscr{A}(FM)$, *there is an* $A \in \mathscr{A}(M)$ *such that* $A' \in \mathscr{A}(FA)$. Or, equivalently:

$$\mathscr{A}(FM) = \bigcup \{\mathscr{A}(FA) : A \in \mathscr{A}(M)\}.$$

In words: To obtain an approximate normal form of a function application FM requires only an approximate normal form of the argument M. This is just a λ-calculus (syntactic) analogue of the general considerations motivating the continuity of functions in Scott's theory of computation.

Finally we close this section by considering again the fixed-point combinators of § 3. For $i = 0$ and $i = 1$ (and it seems likely for all $i \geqq 0$) *all* terms to which $Y^{(i)}$ reduces are of the form

$$\lambda f.f^n(\lambda x_1 x_2 \cdots x_k.RX_1X_2 \cdots X_m), \qquad n, k, m \geqq 0,$$

where R is a β-redex and X_j is a term, for $j = 1, 2, \cdots, m$. (For $Y^{(0)}$, we have $k = m = 0$ and $R \equiv XX$ where $X \equiv \lambda x.f(xx)$; see Example 4. For $Y^{(1)}$, see [4, p. 195].) So the approximate normal forms of $Y^{(i)}$ consist essentially (i.e., after use of (Ω_1) and/or (Ω_2) if $m \neq 0$ and/or $k \neq 0$) of

$$\{\lambda f.f^n(\Omega) : n \geqq 0\}$$

Thus, $Y^{(0)}$ and $Y^{(1)}$ have the same set of approximate normal forms—which, in view of Theorem 5.4, is why we would like to regard them as equivalent—and these approximate normal forms correspond exactly to the terms in the l.u.b.-expression for the least fixed-point operator—which is why we would like them to have the least fixed-point operator as their meaning.

6. A characterization of $=_{\mathbf{D}_\infty}$. The results of § 5 provide motivation for several semantic properties of Scott's λ-calculus models—roughly, they show that terms are given the right meanings from a computational point of view. In this section we show that equivalence of these meanings is also natural, by exhibiting a direct connection between $=_{\mathbf{D}_\infty}$ (and the ordering $\sqsubseteq$) and a syntactic relation definable from the conversion rules.

From our discussion so far, two particular equivalence relations between terms are worth noting:

1. *approximate normal form equivalent*: $M \sim_a N$

$$M \sim_a N \equiv_{\text{def}} \mathscr{A}(M) = \mathscr{A}(N)$$

2. *normal form equivalent* (first studied by Morris [11]): $M \sim_n N$

$$M \sim_n N \equiv_{\text{def}} \text{for all contexts } \mathbf{C}[\ ],\ \mathbf{C}[M] \text{ has a normal form iff } \mathbf{C}[N] \text{ has (the same) normal form.}$$

Then Theorems 5.4 and 5.2 imply

$$M \sim_a N \Rightarrow M \sim_n N,$$

$$M \sim_a N \Rightarrow M =_{\mathbf{D}_\infty} N,$$

Further, it can be shown that

$$M \sim_n N \Rightarrow M =_{\mathbf{D}_\infty} N,$$

but the reverse implication does not hold (e.g., consider I and J with the null context $\mathbf{C}[\ ] \equiv [\ ]$)—essentially what happens is that the property of having a normal form is too strong. However, if we replace "normal form" by "head normal form" throughout, as suggested by § 4, then implications can be established in both directions. That is, if we define *head normal form equivalence*, $M \sim_h N$, analogous to $M \sim_n N$, then

$$M \sim_h N \Leftrightarrow M =_{\mathbf{D}_\infty} N$$

From right to left this is an easy consequence of Corollary 5.3 and substitutivity. In the other direction the full proof is rather intricate but can be derived quite straightforwardly with the aid of a lemma we shall state without proof (Lemma 6.2 below).

First, it is convenient to introduce two alternative formulations and corresponding quasi-orderings. We define *M is solvably extended by N* as

$$M \lesssim_s N \equiv_{\text{def}} \text{ for all contexts } \mathbf{C}[\ ], \text{ if } \mathbf{C}[M] \text{ is solvable, so is } \mathbf{C}[N].$$

and write $M \sim_s N$ if each is solvably extended by the other. We define *M is semi-separable from N* by

$$M \not\lesssim N \equiv_{\text{def}} \text{ there is } a \text{ (head) context } \mathbf{C}[\ ] \text{ such that } \mathbf{C}[M] \textbf{ red } I \text{ but } \mathbf{C}[N] \text{ is unsolvable,}$$

and call two terms *semi-separable* if either is semi-separable from the other. (The latter is a weak version of "separable" defined in § 1; the prefix "semi-" is used in the same sense as its usage in connection with decision procedures in computability theory.)

LEMMA 6.1. *For all terms M and N,*

(a) $M \sim_h N \Leftrightarrow M \sim_s N$,

(b) $M \not\lesssim N \Leftrightarrow M \not\lesssim_s N$.

Proof. The proof is immediate from the definitions, by Corollaries 5.3, 4.2, respectively. □

LEMMA 6.2. *Suppose A is in approximate normal form and N is any term. Then*

$$A \not\sqsubseteq N \Rightarrow A \not\lesssim N.$$

Proof. The proof is achieved by a generalization of the construction used by Bohm [5] in the proof of Theorem 1.3 and will be given in [24]. □

Now we can establish the characterization of $=_{\mathbf{D}_\infty}$. In fact it is more convenient to do so for the inequality $\neq_{\mathbf{D}_\infty}$, along with a characterization of $\not\sqsubseteq$:

THEOREM 6.3.[6] *The following three conditions are equivalent:*

(a) *M and N are semi-separable* $(M \not\lesssim N)$,

(b) *M and N are not solvably equivalent* $(M \not\lesssim_s N)$,

(c) $M \neq_{\mathbf{D}_\infty} N$ $(M \not\sqsubseteq N)$.

Proof. By Lemma 6.1(b), it suffices to show (a) is equivalent to (c).

To show (c) implies (a), suppose $M \not\sqsubseteq N$. Then $A \not\sqsubseteq N$ for some $A \in \mathscr{A}(M)$, by Theorem 5.2 and the properties of l.u.b's. By Lemma 6.2, there is a context $\mathbf{C}[\ ]$ such that $\mathbf{C}[N]$ is unsolvable and $\mathbf{C}[A]$ **red** I, whence also $\mathbf{C}[M]$ **red** I by Theorem 5.4(a), which shows M is semi-separable from N.

To show (a) implies (c), suppose $M \not\lesssim N$ and let $\mathbf{C}[\ ]$ be a context such that $\mathbf{C}[M]$ **red** I and $\mathbf{C}[N] \equiv U$ is unsolvable. Then $M \sqsubseteq N$ would imply $I \sqsubseteq U$, contradicting U being unsolvable (because of Corollary 5.3). □

For closed terms, Theorem 6.3 specializes to:

THEOREM 6.4. *If M and N are both closed terms, the following three conditions are equivalent:*

(a) $M \not\sqsubseteq N$,

(b) *there is a closed term F such that FM* **red** *I and FN is unsolvable,*

(c) *there exist closed terms* $X_1, X_2, \cdots, X_k$ $(k \geqq 0)$ *such that*

$$MX_1X_2 \cdots X_k \textbf{ red } I$$

while $NX_1X_2 \cdots X_k$ *is unsolvable.*

[6] Essentially this same result has also been proved independently by J. M. E. Hyland.

Proof. For (c) implies (b), simply set $F \equiv \lambda z.zX_1X_2 \cdots X_k$. That (b) implies (a) follows from Theorem 6.3 since, when (b) holds, $\mathbf{C}[\ ] \equiv F[\ ]$ is then a semi-separating context. For (a) implies (c), suppose $M \not\subseteq N$, so that M is semi-separable from N, by Theorem 6.3. Let

$$\mathbf{C}[\ ] \equiv (\lambda x_1 x_2 \cdots x_n.[\ ])M_1M_2 \cdots M_nA_1A_2 \cdots A_m$$

be a head context such that $\mathbf{C}[M]$ **red** I and $\mathbf{C}[N] \equiv U$ is unsolvable. Since M is closed, we have

$$\mathbf{C}[M] \equiv (\lambda x_1 x_2 \cdots x_n.M)M_1M_2 \cdots M_nA_1A_2 \cdots A_m \quad \beta\textbf{-red} \quad MA_1A_2 \cdots A_m$$

and similarly for N. Hence (c) follows by setting $k = m$ and $X_i \equiv A_i$ for $i = 1, 2, \cdots, k$. □

We can understand the content of Theorem 6.4 further with reference to the lattice structure of $\mathbf{D}_\infty$. For distinct elements $a, b \in \mathbf{D}_\infty$, say $a \not\subseteq b$ for definiteness, there are always continuous functions which (partially) distinguish between them: for arbitrary $d \neq \bot$,

(6.1) $a \not\subseteq b \Rightarrow$ there is a continuous $f : \mathbf{D}_\infty \to \mathbf{D}_\infty$ such that $f(a) = d$, $f(b) = \bot$.

(E.g., the function

$$f(x) = \textbf{if } x \not\subseteq b \textbf{ then } d \textbf{ else } \bot$$

has this property and is continuous.) Now call an element of $\mathbf{D}_\infty$ λ*-definable* iff it is the value $\mathscr{V}[\![M]\!](\rho)$ of a *closed* term M (since M is closed, the choice of ρ doesn't matter), and suppose a and b are λ-definable elements, say

$$a = \mathscr{V}[\![M]\!](\rho), \quad b = \mathscr{V}[\![N]\!](\rho).$$

Since $M \not\subseteq N$ iff $a \not\subseteq b$ (by definition of $\not\subseteq$), (6.1) implies that when $M \not\subseteq N$ there is a continuous function f which distinguishes between the values of M and N, and by the isomorphism Φ, Ψ this function f can itself be considered an element of $\mathbf{D}_\infty$. Theorem 6.4 expresses the stronger property that, for distinct λ-definable elements, there is always a λ*-definable* function which distinguishes between them. A second consequence, for closed terms M and N, is that

$$M =_{\mathbf{D}_\infty} N \quad \text{iff} \quad MZ =_{\mathbf{D}_\infty} NZ \quad \text{for all } \textit{closed} \text{ terms } Z$$

(which incidentally implies we have models satisfying the ω-rule of Barendregt [2, p. 48], for the restriction that M and N be closed terms can be removed with the aid of Theorem 6.3). So the λ-definable subspace has pleasant "closure" properties and the term structure of the model (i.e., the ordering $\sqsubseteq$ and the equality $=_{\mathbf{D}_\infty}$ between terms) depends only on the λ-definable subspace of $\mathbf{D}_\infty$.

We can also see, with hindsight, why the construction of $\mathbf{D}_\infty$ yields essentially the same model for arbitrary choice of the initial domain $\mathbf{D}_0$ (but not for arbitrary choice of the initial projection pair ϕ_0, ψ_0—the characterization theorems above hold only with the ϕ_0, ψ_0 mentioned in § 3). It is not difficult to show that, of the elements of $\mathbf{D}_0$ (regarded as the subspace $\mathbf{D}_0^{(\infty)}$ of $\mathbf{D}_\infty$), $\bot$ is the only one which is λ-definable. (In fact, $\bot$ is the only λ-definable element of the union of all the "finite-type" spaces $\mathbf{D}_n^{(\infty)}$, $n = 0, 1, 2, \cdots$.) For the choice of $\mathbf{D}_0$ this means that, although $\mathbf{D}_0$ must include two distinct elements, $\bot$ and $\top$, for a nontrivial model, additional elements are redundant as they will not be λ-definable and hence cannot affect the term structure of the resulting model.

Further insight is provided by recalling the intuitive interpretation of partial orderings in Scott's theory, in terms of "information content". For domain elements a and b, $a \not\sqsubseteq b$ was intended to mean "a gives some information where b gives either no information or different information". The results of § 5 showed that (the values of) terms having different computational behavior have differences in their "information content"; Theorem 6.3 gives the converse, that when there is a difference in the "information content" of (the values of) terms in $\mathbf{D}_\infty$, this can be detected in their computational behavior. Roughly, when $M \not\sqsubseteq N$, their semi-separating context searches the terms for the information contained in M but not in N. If and when this is found, i.e., for M, the appropriate part of M is extracted and "standardized" to give the identity as result. In the corresponding part of N one finds no information (an unsolvable term) or different information (a term with head normal form dissimilar to that in the corresponding part of M); in either case the standardizing stage for M can be chosen so that when applied to this part of N the resulting term is unsolvable. (In fact, the omitted proof of Lemma 6.2 is just a formalization of this informal sketch; the tricky part involves showing that all the selections and manipulations can be done within the λ-calculus.)

A more technical consequence concerns the possibility of giving additional axioms for the λ-calculus in an attempt to axiomatize the model completely. Although neither the relation $=_{\mathbf{D}_\infty}$ not its negation $\neq_{\mathbf{D}_\infty}$ are recursively enumerable (in fact, $=_{\mathbf{D}_\infty}$ is Π^0_2-complete), effective relative axiomatizations are possible. Since unsolvable terms are equal in $\mathbf{D}_\infty$, consider the extended theory λ^* in which we postulate the equality of all unsolvable terms:

$$\text{(Uns)} \qquad \frac{M, N \text{ both unsolvable}}{M = N}$$

as an additional inference rule. It is easily shown that any equality of semi-separable terms is inconsistent with (Uns), whence it follows from Theorem 6.3 that $M \neq_{\mathbf{D}_\infty} N$ iff $M = N$ would render the theory λ^* inconsistent. This implies that the *inequality* $\neq_{\mathbf{D}_\infty}$ (and similarly the relation $\not\sqsubseteq$) can be effectively axiomatized *relative to* λ^*; roughly, one constructs a proof of $M \neq_{\mathbf{D}_\infty} N$ by tracing backwards a proof of inconsistency of $\lambda^* + M = N$ (e.g., a proof of $I = K$ in $\lambda^* + M = N$). Such a relative axiomatization is of some interest as (Uns) is the only noneffective rule in λ^*—in fact, (Uns) is equivalent to the halting problem (implied by results in [2]). In a certain sense, this is the best one can do as regards axiomatizing the inequality $\neq_{\mathbf{D}_\infty}$ (because $=_{\mathbf{D}_\infty}$ is a Π^0_2-relation).

It also follows that $=_{\mathbf{D}_\infty}$ is the *largest, consistent* model of the theory λ^*. This has the interesting interpretation: given that we wish to consider all unsolvable terms equivalent, terms have different values in $\mathbf{D}_\infty$ iff this is necessary for consistency. Or, equivalently, terms are equal in $\mathbf{D}_\infty$ iff there is no reason to distinguish between them, where we regard the property of terms having different computational behavior (in the "global" sense we have discussed) as a necessary and sufficient reason for distinguishing between them.

7. Rationalization for $I = J$. The possibility of equivalences between normal forms and terms with no normal form is one of the more unexpected, almost

pathological, properties of the models we have studied and deserves a separate discussion. At first, this possibility was surprising because we felt there was a close connection between the property of having a normal form and termination of programs. In this way we hoped to find a model-theoretic characterization of "normal form" and hence suggest a semantic analogue of termination. In § 4, we saw reason to doubt the analogy, so our original motivation here no longer applies.

However, this alone cannot be said to imply that we should necessarily expect or desire that $I = J$; to rationalize this, more is needed. Of course, the general consequences of the characterization in § 6 provide some motivation, but there are several more specific arguments also.

First, under a suitable formalization of "infinite normal form", the term J will have an infinite normal form—roughly, this will be

$$J = \lambda x_0.\lambda x_1.x_0(\lambda x_2.x_1(\lambda x_3.x_2(\lambda x_4.x_3(\cdots))))$$

—while $I \equiv \lambda x.x$ is a finite normal form. So $I = J$ is a λ-calculus analogue of, e.g., $3 = 2.999\cdots$, in the sense that it is an example of a finite expression which is equivalent to an infinite one, just as $3 = 2.999\cdots$ is an example of a real number which has both a finite and an infinite decimal notation.

A second insight concerns the operation $\mathbf{H}$ of (a single) η-expansion viewed as an operation within the λ-calculus. $\mathbf{H}$ can be expressed by the equation

$$\mathbf{H}(x) = \lambda y.x(y), \qquad y \not\equiv x,$$

and is, of course, an identity operation in any extensional model. Suppose now that instead of performing just one η-expansion, we consider an infinite η-expansion operator obtained by recursively applying η-expansion to the newly introduced variable y:

$$\mathbf{H}_\infty(x) = \lambda y.x(\mathbf{H}_\infty(y)), \qquad y \not\equiv x.$$

This equation is now just that which is satisfied by the term J, so J represents an infinite number of applications of an identity operation. Iterating an identity operation an *infinite* number of times will not always given an identity operation (often it will give "undefined") but this is the case for the particular iteration involved in J. Perhaps the best way of expressing this is that when we write a recursive definition we can always compute results step-by-step via the recursion, but sometimes a solution can also be found as a closed formula (non-recursive).

Clearly $\mathbf{H}_\infty$ can be generalized to other operators, e.g., to ones which perform any finite number of η-expansions at the top level before recursing on some or all of the newly introduced variables; all such examples will have the identity function as their value in $\mathbf{D}_\infty$. In a certain sense, this method and that of Corollary 3.5 exhaust all possibilities of equivalences between normal forms and terms with no normal form,[7] so all such examples are essentially just variations on the $I = J$ one.

[7] More precisely, we mean here that, given any term N in normal form, by applying infinite η-expansion operators to some or all of its subterms (not necessarily the same operator for each subterm) we can obtain many terms equivalent to N which have no normal form. Under a suitable formalization of "infinite η-expansion operator", we conjecture that every term not having a normal form but equivalent to N in $\mathbf{D}_\infty$ can be obtained in this way, up to α-β-conversion. However, we shall not pursue this formally here as the ideas are more clearly seen in the general discussion of "infinite conversions" to follow, but see also the treatment of infinite normal forms in Nakajima [12].

It is also interesting here to compare the characterization of $=_{\mathbf{D}_\infty}$ with the more naturally occurring equivalence $\sim_n$ defined in § 6. All the above examples will fail for $\sim_n$ (because no normal form can be normal form equivalent to a term without a normal form), but it can be shown that, again, they essentially exhaust all examples of terms for which $M =_{\mathbf{D}_\infty} N$ but $M \not\sim_n N$.

An alternative formulation allows comparison with other calculi. We can say that I and J are equal by "infinite conversion" or are "interconvertible in the limit"; we illustrate this view in Fig. 1, where we have applied suitably chosen η-expansions to I and α-β-reductions to J. As these conversions proceed, it can be seen that the part where the terms differ is pushed deeper and deeper inside the terms. In other words, although I and J are not finitely interconvertible, they can be transformed, by means of the conversion rules alone, to terms which match each other up to arbitrarily large, finite "depth".

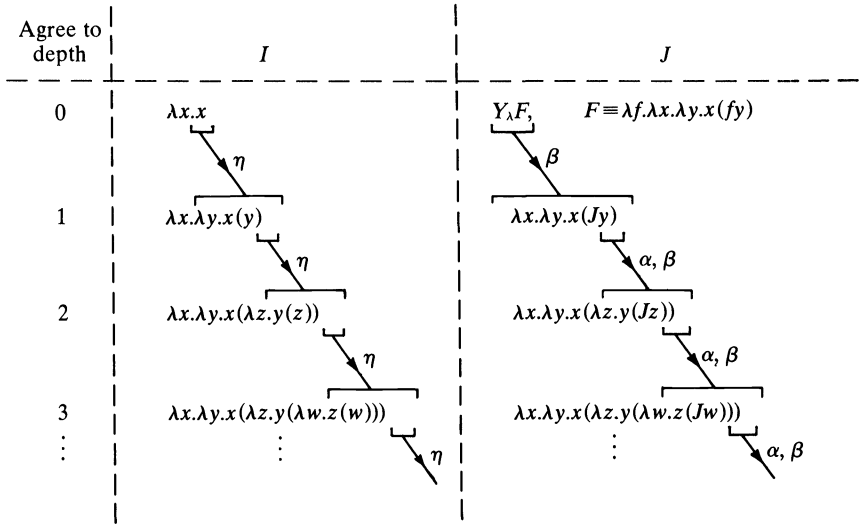


FIG. 1

This phenomenon can be generalized to give a full characterization of $=_{\mathbf{D}_\infty}$ along the same lines by taking account of the rule (Uns) equating unsolvable terms. ((Uns) was not needed in Fig. 1 as no subterm was unsolvable.) We have: *$M =_{\mathbf{D}_\infty} N$ iff M and N can be transformed by α-β-η-*Uns*-conversion to terms which agree to arbitrary depth.* More formally:

$$M =_{\mathbf{D}_\infty} N \quad \textit{iff} \quad (\forall k \geqq 0)(\exists M', N')(M' \simeq_k N' \quad \textit{and} \quad \lambda^* \vdash M = M', N = N')$$

where $\simeq_k$ symbolizes the notion of terms matching to depth k. By formalizing the notion of "depth" which we have indicated only rather loosely above, this formulation can be made the basis for an axiomatization of $=_{\mathbf{D}_\infty}$ with only two noneffective rules—the rule (Uns) and an infinitistic rule corresponding to the universal quantifier for k. (Again there is a similar axiomatization of the ordering $\sqsubseteq$).

By comparison, consider functions on the integers. One can give two definitions for the identity function, the explicit definition

$$f = \lambda n \in \mathbf{int}.n$$

and the recursive definition

$$g = \lambda n \in \mathbf{int}.\ \mathbf{if}\ n = 0\ \mathbf{then}\ 0\ \mathbf{else}\ g(n-1)+1$$

Although these two definitions clearly define the same function, one cannot give a recursive function calculus with a finite set of (effective) "conversion rules" such that the equivalence of f and g can be shown by transforming them into identical expressions. (Proofs of $f = g$ normally require the use of inductive methods.) However if we consider the rule

(R_k) Replace n by $\mathbf{if}\ n = k\ \mathbf{then}\ k\ \mathbf{else}\ n$

where n is an integer variable and k is a numeral, then, just as for I and J in Fig. 1, we can obtain "conversions" of f and g to expressions which agree to arbitrary depth. This is shown in Fig. 2, where, for g, we use the "copy-rule" to handle recursion plus the usual simplification rules of elementary arithmetic.

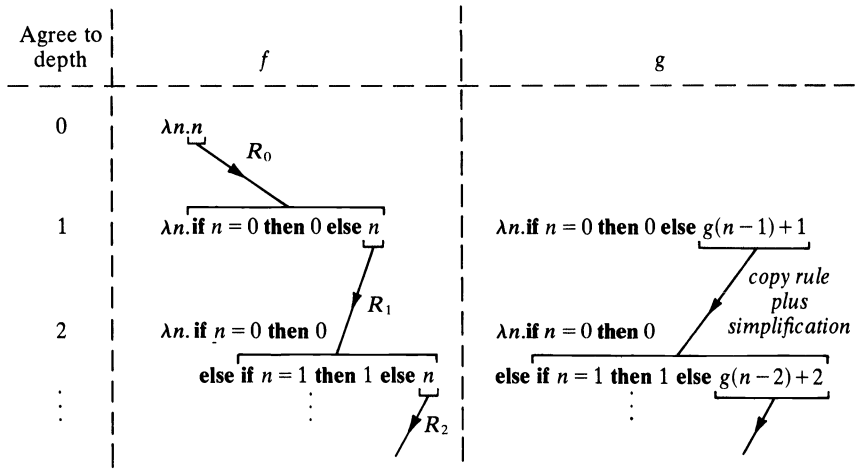


FIG. 2

Taken together, we feel these arguments show that equivalences such as $I =_{\mathbf{D}_\infty} J$ are not unreasonable and, indeed, should be regarded as natural *whenever one accepts the validity of η-conversion.*

The proviso is, of course, crucial and at least inconvenient when dealing with most computer languages. We mentioned in § 1 that η-conversion embodies the view that *every* object is a function. There are no constants as such in the "pure" λ-calculus, one effect of which is that λ-calculus computations never quite touch ground level (e.g., by giving an integer, a truth value, or some other primitive object, as the result of a computation); if needed, constants must be represented as functions. Though this is not the most transparent approach for constants, it is a

possible one and should have a corresponding "natural semantics". The properties of $\mathbf{D}_\infty$ can then be read as showing that when one is happy to represent everything as a function, the $\mathbf{D}_\infty$-model is a natural one, in the sense that the meanings of terms and their equivalences have the appropriate consequences for computational purposes.

A better approach, however, is to include primitive objects, or *atoms*, directly in the interpretation. Then one might study various extended λ-calculi which include constants in their syntax and some additional replacement rules (called δ-rules) for these. Models for such "applied" λ-calculi can be found via solutions of

$$\mathbf{E} \cong \mathbf{A} + [\mathbf{E} \to \mathbf{E}]$$

where $\mathbf{A}$ is some primitive lattice of atoms. The interpretation of terms runs much as before, with some given interpretation for the constants. (The only ambiguity concerns an application $a(x)$ in which a is an atom in $\mathbf{A}$; this is given a standard meaning, e.g., $\bot$, or $\top$, or a special *error*-atom.) Any solution for $\mathbf{E}$ will provide models for α-β-conversion but not, of course, for η-conversion (because distinct atoms behave the same as functions).

Particular solutions, $\mathbf{E}_\infty$, are known in which many of the results for $\mathbf{D}_\infty$ remain true, e.g., the analogues of Corollary 3.4 and Theorem 5.2. Whether all unsolvable terms will have value $\bot$ in $\mathbf{E}_\infty$ depends on the choice between "separated" and "coalesced" sums; that is, on whether one wishes to distinguish between the least element $\bot_{\mathbf{E}}$ and the least function $\bot_{[\mathbf{E}\to\mathbf{E}]}$. If so, this is mirrored in the syntactic aspects by calling an unsolvable term *strongly* unsolvable if also it is not reducible to an abstraction; then, with separated sums, terms will have value $\bot$ iff they are strongly unsolvable.

Equivalences such as $I = J$ will fail to hold. In fact, it can be shown that no normal form can be equal in $\mathbf{E}_\infty$ to a term without a normal form. This is to be expected as there is no reason to regard I and J as equivalent when η-conversion is denied; indeed, they can be distinguished by application to a constant.

The relation $=_{\mathbf{E}_\infty}$ will clearly be different from $=_{\mathbf{D}_\infty}$. One obvious conjecture is: $M =_{\mathbf{E}_\infty} N$ iff for all contexts $\mathbf{C}[\ ]$, $\mathbf{C}[M]$ has a normal form just when $\mathbf{C}[N]$ has the same normal form (here, normal form would mean β-δ-normal form). But with constants present, since whole programs are generally intended to give basic objects rather than functions as results, a more natural characterization would be

$$M =_{\mathbf{E}_\infty} N \Leftrightarrow \text{for all contexts } \mathbf{C}[\ ],\ \mathbf{C}[M] \text{ is reducible to a constant just when } \mathbf{C}[N] \text{ is reducible to the same constant.}$$

It seems likely that such a characterization can be established if sufficient constants (one for each atom?) and δ-rules are included in an extended calculus.

8. Conclusion. It is worth reflecting on how our results might generalize to other languages. Clearly there should always be some connection between the denotational semantics of a language and the properties of the computations which can be evoked by its programs. When such a connection has been established, we can then work with either the denotational or an implementation-oriented definition of the language. If we regard the denotational semantics as the

primary definition, the connection would show that the implementation is not only correct but is "effectively complete", in the sense that it can compute and make use of every "bit" of information determined by the denotations. Alternatively, if we regard the implementation as given, such a connection establishes the denotational meanings as a valid basis for reasoning about the "global" properties of programs under the implementation (conceptually a simpler task than working with the implementation directly).

For proofs based on a denotational semantics, methods will be needed for performing inductions on the structure of domains, particularly domains obtained as solutions of isomorphic domain equations. As with the properties of λ-calculus models, many program properties will not be dependent on the internal structure of particular solutions, but it seems likely that some will be so dependent. In our study the laws of projections on $\mathbf{D}_\infty$ embody the relevant structure, but it remains to be seen whether a similar use of projections will provide a convenient framework for inductions on solutions of other domain equations.

The characterization of semantic equivalence of terms in $\mathbf{D}_\infty$ has an obvious counterpart for other languages in the form of a *substitutivity test*: program fragments may reasonably be regarded as equivalent just when either can serve in place of the other in any program without affecting the "input-output" behavior of the program. To emphasize the sense in which a program's behavior might be affected, we can generalize the notion of "semi-separable" as follows. Let π, π' be (well-formed) program fragments of the same syntactic category of a language **L**. Call π and π' *semi-separable* iff there are programs Π and Π' of **L**, differing only in an occurrence of π and π', respectively, such that the execution of Π halts giving some standard output (e.g., 0) while that of Π' would continue indefinitely giving no intermediate output, or vice versa. Then, given a semantics for **L**, one can conjecture that π and π' are semantically distinct just when they are semi-separable. If this holds we can reasonably claim that the given semantics for **L** is the right one. (Of course, with the benefit of hindsight, these suggestions are not surprising.)

Finally, note the closure properties of the definable subspace of $\mathbf{D}_\infty$. In general, any solution of a domain equation will contain nondefinable elements and it would be distressing if two definable elements were equivalent in all the ways they may be combined with definable elements yet could be distinguished by making use of the nondefinable elements; for in the semantics based on such a domain there would be programs which have different meanings yet have the same computational behavior. (Of course, the results of § 6 show this does not occur for the λ-calculus semantics based on $\mathbf{D}_\infty$.)

Such a situation is not necessarily irretrievable, however. One possibility is simply that a language may be lacking in generality or expressive power; this could be patched either by extending the language directly or by regarding computational equivalence as being determined with respect to an extended language. A second possibility is that the definition of semantic equivalence could be modified; instead of defining semantic equivalence with reference to *all* environments, one can consider only environments in which the denotation of every variable is a definable element of the domains used in the semantics (with similar modifications for equivalence of functional abstractions). However, the structure of the

domains may not then be as directly applicable to equivalence proofs as it was for the λ-calculus examples in § 3. A recent paper of Plotkin [14] points out these and other difficulties and expands on the possibilities.

Acknowledgments. I wish to thank Professors Dana Scott, Christopher Strachey, and John Reynolds for many helpful suggestions and encouragement. I am especially indebted to Martin Hyland for the notion of type-assignment used in the proof of Theorem 5.2.

REFERENCES

[1] J. W. DE BAKKER AND W. P. DE ROEVER, *A calculus for recursive program schemes*, Proc. IRIA Colloquium on Automata, North-Holland, Amsterdam, 1972.

[2] H. P. BARENDREGT, *Some extensional term models for combinatory logics and λ-calculi*, Ph.D. thesis, Utrecht Univ., the Netherlands, 1971.

[3] ———, *Solvability in λ-calculi*, Proc. 1972 Orléans Congrès de Logique, J. P. Calais, J. Derrick and G. Sabbagh, eds., to appear.

[4] C. BÖHM, *The CUCH as a formal and descriptive language*, Formal Language Description Languages, T. B. Steel, ed., North-Holland, Amsterdam, 1966, pp. 179–197.

[5] ———, *Alcune proprieta delle forme β-η-normali nel λ-K-calcolo*, Consiglio Nazionalle delle Recerche, no. 696, Rome, 1968.

[6] J. M. CADIOU, *Recursive definitions of partial functions and their computations*, Ph.D. thesis, Stanford Univ., Stanford, Calif., 1972.

[7] H. B. CURRY AND R. FEYS, *Combinatory Logic*, vol. 1, North-Holland, Amsterdam, 1958.

[8] H. B. CURRY, J. R. HINDLEY AND J. SELDIN, *Combinatory Logic*, vol. 2, North-Holland, Amsterdam, 1972.

[9] S. C. KLEENE, *Introduction to Metamathematics*, Van Nostrand, New York, 1952.

[10] C. MCGOWAN, *Correctness results for lambda-calculus interpreters*, Ph.D. thesis, Cornell Univ., Ithaca, N.Y., 1971.

[11] J. H. MORRIS, *Lambda-calculus models of programming languages*, Ph.D. thesis, Project MAC, Mass. Inst. of Tech., Cambridge, Mass., 1968.

[12] R. NAKAJIMA, *Infinite norms for λ-calculus*, Symposium on λ-calculus and Computer Science Theory, Consiglio Nazionalle delle Ricerche, Rome, 1975.

[13] D. M. R. PARK, *The Y-combinator in Scott's lambda-calculus models*, Symposium on Theory of Programming, Univ. of Warwick, unpublished, 1970.

[14] G. D. PLOTKIN, *LCF considered as a programming language*, J. Theoret. Computer Sci., to appear.

[15] J. C. REYNOLDS, *Notes on a lattice-theoretic approach to the theory of computation*, Dept. of Systems and Information Science, Syracuse Univ., Syracuse, N.Y., 1972.

[16] B. K. ROSEN, *Tree-manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20 (1973), pp. 160–187.

[17] D. SCOTT, *Lattice-theoretic models for the λ-calculus*, Princeton Univ., Princeton, N.J., unpublished, 1969.

[18] ———, *Outline of a mathematical theory of computation*, Proc. 4th Ann. Princeton Conf. on Information Sciences and Systems, Princeton Univ., Princeton, N.J., 1970, pp. 169–176.

[19] ———, *The lattice of flow diagrams*, Semantics of Algorithmic Languages, E. Engeler, ed., Springer Lecture Notes in Mathematics, no. 188, Springer-Verlag, New York, 1971, pp. 311–366.

[20] ———, *Data types as lattices*, Notes, Amsterdam, unpublished, 1972.

[21] ———, *Lattice theory, data types and semantics*, Formal Semantics of Programming Languages, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 65–106.

[22] ———, *Lattice-theoretic models for various type-free calculi*, Proc. 4th Internat. Congr. for Logic, Methodology, and the Philosophy of Science (Bucharest), North-Holland, Amsterdam, 1973.

[23] C. P. WADSWORTH, *Approximate reduction and lambda-calculus models*, this Journal, to appear.

[24] ———, *A general form of a theorem of Böhm and its application to Scott's models for the* λ*-calculus*, in preparation.

SIAM J. COMPUT.
Vol. 5, No. 3, September 1976

DATA TYPES AS LATTICES

To the Memory of Christopher Strachey, 1916–1975

DANA SCOTT†

Abstract. The meaning of many kinds of expressions in programming languages can be taken as elements of certain spaces of "partial" objects. In this report these spaces are modeled in one universal domain $\mathbf{P}\omega$, the set of all subsets of the integers. This domain renders the connection of this semantic theory with the ordinary theory of number theoretic (especially general recursive) functions clear and straightforward.

Key words. programming language semantics, lattice, continuous lattice, algebraic lattice, computability, retract, combinatory logic, lambda calculus, recursion theorem, enumeration degrees, continuous function, fixed-point theorem

Introduction. Investigations begun in 1969 with Christopher Strachey led to the idea that the denotations of many kinds of expressions in programming languages could be taken as elements of certain kinds of spaces of "partial" objects. As these spaces could be treated as function spaces, their structure at first seemed excessively complicated—even impossible. But then the author discovered that there were many more spaces than we had first imagined—even wanted. They could be presented as lattices (or as some prefer, semilattices), and the main technique was to employ topological ideas, in particular the notion of a continuous function. This approach and its applications have been presented in a number of publications, but that part of the foundation concerned with *computability* (in the sense of recursion theory) was never before adequately exposed. The purpose of the present paper is to provide such a foundation and to simplify the whole presentation by a de-emphasis of abstract ideas. An Appendix and the references provide a partial guide to the literature and an indication of connections with other work.

The main innovation in this report is to model everything within one "universal" domain $\mathbf{P}\omega = \{x | x \subseteq \omega\}$, the domain of all subsets of the set ω of nonnegative integers. The advantages are many: by the most elementary considerations $\mathbf{P}\omega$ is recognized to be a lattice and a topological space. In fact, $\mathbf{P}\omega$ is a *continuous lattice*, even an *algebraic lattice*, but in the beginning we do not even need to define such an "advanced" concept; we can save these ideas for an *analysis* of what has been done here in a more direct way. Next by taking the set of integers as the basis of the construction, the connection with the ordinary theory of number-theoretic (especially, general recursive) functions can be made clear and straightforward.

The model $\mathbf{P}\omega$ can be intuitively viewed as the domain of *multiple-valued* integers; what is new in the presentation is that functions are not only multiple-valued but also "multiple-argumented". This remark is given a precise sense in § 2 below, but the upshot of the idea is that multiple-valued integers are regarded as

* Received by the editors May 23, 1975, and in revised form March 17, 1976.
† Mathematics Department, Oxford University, Oxford, England OX2 6PE.

objects in themselves—possibly infinite—and as something more than just the collection of single integers contained in them. This combination of the finite with the infinite into a single domain, together with the idea that a continuous function can be reduced to its *graph* (in the end, a set of integers), makes it possible to view an $x \in \mathbf{P}\omega$ at one time as a value, at another as an argument, then as an integer, then as a function, and still later as a functional (or combinator). The "paradox" of self-application (as in $x(x)$) is solved by allowing the *same* x to be used in two *different* ways. This is done in ordinary recursion theory via Gödel numbers (as in $\{e\}(e)$), but the advantage of the present theory is that not only is the function concept the *extensional* one, but it includes *arbitrary* continuous functions and not just the computable ones.

Section 1 introduces the elementary ideas on the topology of $\mathbf{P}\omega$ and the continuous functions including the fixed-point theorem. Section 2 has to do with computability and definability. The language LAMBDA is introduced as an extension of the pure λ-calculus by four arithmetical combinators; in fact, it is indicated in § 3 how the whole system could be based on *one* combinator. What is shown is that computability in $\mathbf{P}\omega$ according to the natural definition (which assumes that we already know what a recursively enumerable set of integers is) is equivalent to LAMBDA-definability. The main tool is, not surprisingly, the first recursion theorem formulated with the aid of the so-called *paradoxical combinator* $\mathbf{Y}$. The plan is hardly original, but the point is to work out what it all means in the model.

Along the way we have to show how to give every λ-term a denotation in $\mathbf{P}\omega$; the resulting principles of λ-calculus that are thereby verified are summarized in Table 1. Of these the first three, (α), (β), and (ξ), are indeed valid in the model; however, rule (η), which is a stronger version of extensionality, *fails* in the $\mathbf{P}\omega$ model. This should not be regarded as a disadvantage since the import of (η) is to suppose every object is a function. A quick construction of these special models is indicated at the end of § 5. Since $\mathbf{P}\omega$ is partially ordered by $\subseteq$, there are also laws involving this relation. Law (ξ^*) is an improvement of (ξ); while (μ) is a form of monotonicity for application.

TABLE 1

Some laws of λ*-calculus*

(α)	$\lambda x.\tau = \lambda y.\tau[y/x]$
(β)	$(\lambda x.\tau)(y) = \tau[y/x]$
(ξ)	$\lambda x.\tau = \lambda x.\sigma$ iff $\forall x.\tau = \sigma$
(η)	$y = \lambda x.y(x)$
(ξ^*)	$\lambda x.\tau \subseteq \lambda x.\sigma$ iff $\forall x.\tau \subseteq \sigma$
(μ)	$x \subseteq y$ and $u \subseteq v$ imply $u(x) \subseteq v(y)$

Section 3 has to do with enumeration and degrees. Gödel numbers for LAMBDA are defined in a very easy way which takes advantage of the notation of combinators. This leads to the second recursion theorem, and results on incompleteness and undecidability follow along standard lines. *Relative recursiveness* is also very easy to define in the system, and we make the tie-in with *enumeration degrees* which correspond to finitely generated combinatory subalgebras of $\mathbf{P}\omega$. Finally a theorem of Myhill and Shepherdson is interpreted as a most satisfactory completeness property for definability in the system.

Sections 4 and 5 show how a calculus of *retracts* leads to quite simple definitions of a host of useful domains (as lattices). Section 6 investigates the classification of other subsets (nonlattices) of $\mathbf{P}\omega$; while § 7 contrasts partial (multiple-valued) functions with total functions, and interprets various theories of functionality. Connections with category theory are mentioned.

What is demonstrated in this work is how the language LAMBDA, together with its interpretation in $\mathbf{P}\omega$, is an extremely convenient vehicle for *definitions* of computable functions on complex structures (all taken as subdomains of $\mathbf{P}\omega$). It is a "high-level" programming language for recursion theory. It is *applied* combinatory logic, which in usefulness goes far beyond anything envisioned in the standard literature. What has been shown is how many interesting predicates can be *expressed* as equations between continuous functions. What is needed next is a development of the *proof theory* of the system along the lines of the work of Milner, which incorporates the author's extension of McCarthy's rule of recursion induction to this high-level language. Then we will have a flexible and practical "mathematical" theory of computation.

1. Continuous functions. The domain $\mathbf{P}\omega$ of all subsets of the set ω of nonnegative integers is a complete lattice under the partial ordering $\subseteq$ of set inclusion, as is well known. We use the usual symbols $\cup, \cap, \bigcup, \bigcap$ for the finite and infinite lattice operations of union and intersection. $\mathbf{P}\omega$ is of course also a Boolean algebra; and for complements we write

$$\sim x = \{n | n \notin x\}$$

where it is understood that such variables as i, j, k, l, m, n range over integers *in* ω, while u, v, w, x, y, z range over *subsets of* ω.

The domain $\mathbf{P}\omega$ can also be made into a topological space—in many ways. A common method is to match each $x \subseteq \omega$ with the corresponding characteristic function in $\{0, 1\}^{\omega}$, and to take the induced product topology. In this way $\mathbf{P}\omega$ is a totally disconnected compact Hausdorff space homeomorphic to the Cantor "middle-third" set. This is *not* the topology we want; it is a *positive-and-negative* topology which makes the function $\sim x$ continuous. We want a weaker topology: the topology of positive "information", which has the advantage that all continuous functions possess fixed points. (The equation $x = \sim x$ is impossible.) The topology that we do want is exactly that appropriate to considering $\mathbf{P}\omega$ to be a continuous lattice. But all this terminology of abstract mathematics is quite unnecessary, since the required definitions can be given in very elementary terms.

To make the topology "visible", we introduce a standard enumeration $\{e_n | n \in \omega\}$ of all *finite* subsets of ω. Specifically we set

$$e_n = \{k_0, k_1, \cdots, k_{m-1}\},$$

provided that $k_0 < k_1 < \cdots < k_{m-1}$ and $n = \sum_{i<m} 2^{k_i}$. Thus n is the code number for e_n, and the elements of e_n are the exponents in the binary expansion of the integer n. This is a one-to-one enumeration of finite subsets, where $k \in e_n$ always implies $k < n$, the function max (e_n) is (primitive) recursive in n, and the relations $k \in e_n$, $e_n \subseteq e_m$, $e_n = e_m \cup e_k$ are all (primitive) recursive in k, m, n.

Topologically speaking the finite sets e_n are *dense* in the space $\mathbf{P}\omega$, for each $x \in \mathbf{P}\omega$ is the "limit" of its finite subsets in the sense that

$$x = \bigcup \{e_n | e_n \subseteq x\}.$$

To make this precise we need a rigorous definition of *open subset* of $\mathbf{P}\omega$.

DEFINITION. A *basis* for the neighborhoods of $\mathbf{P}\omega$ consists of those sets of the form:

$$\{x \in \mathbf{P}\omega | e_n \subseteq x\},$$

for a given e_n. An arbitrary *open subset* is then a union of basic neighborhoods.

It is easy to prove that an open subset $U \subseteq \mathbf{P}\omega$ is just a set of "finite character"; that is, a set such that for all $x \in \mathbf{P}\omega$ we have $x \in U$ *if and only if* some finite subset of x also belongs to U. An alternate approach would define directly what we mean by a continuous function using the idea that such functions must preserve limits.

DEFINITION. A function $f : \mathbf{P}\omega \to \mathbf{P}\omega$ is *continuous* iff for all $x \in \mathbf{P}\omega$ we have:

$$f(x) = \bigcup \{f(e_n) | e_n \subseteq x\}.$$

Again it is an easy exercise to prove that a function is continuous in the sense of this definition iff it is continuous in the usual topological sense (namely: inverse images of open sets are open). For giving proofs it is even more convenient to have the usual ε-δ formulation of continuity.

THEOREM 1.1 (The characterization theorem). *A function* $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous iff for all* $x \in \mathbf{P}\omega$ *and all* ε_m *we have*:

$$e_m \subseteq f(x) \quad \textit{iff} \quad \exists e_n \subseteq x . e_m \subseteq f(e_n).$$

Note that open sets and continuous functions have *a monotonicity property*:

whenever $x \subseteq y$ *and* $x \in U$, *then* $y \in U$; *and*
whenever $x \subseteq y$, *then* $f(x) \subseteq f(y)$.

This gives a precise expression to the "positive" character of our topology. However, note too that openness and continuity mean rather more than just monotonicity. In particular, a continuous function is completely determined by the pairs of integers such that $m \in f(e_n)$, as can be seen from the definition. (Hence, there are only a continuum number of continuous functions, but more than a continuum number of monotonic functions.) This brings us to the definition of the *graph* of a continuous function.

To formulate the definition, we introduce a standard enumeration (n, m) of all pairs of integers. Specifically we set

$$(n, m) = \tfrac{1}{2}(n + m)(n \dot{+} m \dot{+} 1) \dot{+} m.$$

This is the enumeration along the "little diagonals" going from left to right, and it produces the ordering:

$$(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), (3, 0), (2, 1), \cdots.$$

Note that $n \leqq (n, m)$ and $m \leqq (n, m)$ with equality possible only in the cases of $(0, 0)$ and $(1, 0)$. This is a one-to-one enumeration, and the inverse functions are (primitive) recursive—but we do not require at the present any notation for them.

DEFINITION. The *graph* of a continuous function $f : \mathbf{P}\omega \to \mathbf{P}\omega$ is defined by the equation:

$$\mathbf{graph}(f) = \{(n, m) | m \in f(e_n)\};$$

while the *function* determined by any set $u \subseteq \omega$ is defined by the equation:

$$\mathbf{fun}(u)(x) = \{m | \exists e_n \subseteq x.(n, m) \in u\}.$$

THEOREM 1.2 (The graph theorem). *Every continuous function is uniquely determined by its graph in the sense that*:

(i) $$\mathbf{fun}\,(\mathbf{graph}(f)) = f.$$

Conversely, every set of integers determines a continuous function and we have:

(ii) $$u \subseteq \mathbf{graph}(\mathbf{fun}(u)),$$

where equality holds just in case u satisfies:

(iii) *whenever* $(k, m) \in u$ *and* $e_k \subseteq e_n$, *then* $(n, m) \in u$.

Besides functions of one variable we need to consider also functions of several variables. The official definition for one variable given above can be extended simply by saying $f(x, y, \cdots)$ is continuous iff it is continuous in *each* of $x, y, \cdots$. Those familiar with the product topology can prove that for our special positive topology on $\mathbf{P}\omega$ this is equivalent to being continuous on the product space (continuous in the variables *jointly*). Those interested only in elementary proofs can calculate out directly from the definition (with the aid of 1.1) that continuity behaves under combinations by substitution [as in: $f(g(x, y), h(y, x, y))$].

THEOREM 1.3 (The substitution theorem). *Continuous functions of several variables on* $\mathbf{P}\omega$ *are closed under substitution.*

The other general fact about continuous functions that we shall use constantly concerns fixed points, whose existence can be proved using a well-known method.

THEOREM 1.4 (The fixed-point theorem). *Every continuous function* $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *has a least fixed point given by the formula*:

$$\mathbf{fix}(f) = \bigcup\{f^n(\varnothing) | n \in \omega\},$$

where $\varnothing$ *is the empty set and* f^n *is the n-fold composition of f with itself.*

Actually **fix** is a *functional* with continuity properties of its own. We shall not give the required definitions here because they can be more easily derived from the construction of the model given in the next section.

For those familiar with the abstract theory of topological spaces we give in conclusion two general facts about continuous functions with values in $\mathbf{P}\omega$ which indicate the scope and generality of our method.

THEOREM 1.5 (The extension theorem). *Let X and Y be arbitrary topological spaces where $X \subseteq Y$ as a subspace. Then every continuous function $f: X \to \mathbf{P}\omega$ can be extended to a continuous function $\bar{f}: Y \to \mathbf{P}\omega$ defined by the equation*:

$$\bar{f}(y) = \bigcup\{\bigcap\{f(x) | x \in X \cap U\} | y \in U\},$$

where $y \in Y$, and U ranges over the open subsets of Y.

THEOREM 1.6 (The embedding theorem). *Every T_0-space X with a countable basis $\{U_n | n \in \omega\}$ for its topology can be embedded in $\mathbf{P}\omega$ by the continuous function $\varepsilon: X \to \mathbf{P}\omega$ defined by the equation*:

$$\varepsilon(x) = \{n | x \in U_n\}.$$

Technically the T_0-hypothesis is what is needed to show that ε is one-to-one. The upshot of these two theorems is that in looking for (reasonable) topological structures we can confine attention to the subspaces of $\mathbf{P}\omega$ and to continuous functions defined on *all* of $\mathbf{P}\omega$. Thus the emphasis on a single space is justified structurally. What we shall see in the remainder of this work is that the use of a single space is also justified practically because the required subspaces and functions can be *defined* in very simple ways by a natural method of equations.

In order to make the plan of the work clearer, the proofs of the theorems have been placed in an Appendix when they are more than simple exercises.

2. Computability and definability. The purpose of the first section was to introduce in a simple-minded way the basic notions about the topology of $\mathbf{P}\omega$ and its continuous functions. In this section we wish to present the details of a powerful language for defining *particular* functions—especially computable functions—and initiate the study of the *use* of these functions. This study is then extended in different ways in the following sections.

Before looking at the language, a short discussion of the "meaning" of the elements $x \in \mathbf{P}\omega$ will be helpful from the point of view of motivation. Now in itself $x \in \mathbf{P}\omega$ is a *set*, but this does not reveal its *meaning*. Actually x has no "fixed" meaning, because it can be *used* in strikingly different ways; we look for meaning here solely in terms of use. Nevertheless it is possible to give some coherent guidelines.

In the first place it is convenient to let the singleton subsets $\{n\} \in \mathbf{P}\omega$ stand for the corresponding integers. In fact, we shall enforce by convention the *equation* $n = \{n\}$ as a way of simplifying notation. In this way, $\omega \subseteq \mathbf{P}\omega$ as a subset. (Note that our convention conflicts with such set-theoretical equations as $5 = \{0, 1, 2, 3, 4\}$. What we have done is to abandon the usual set-theoretical conventions in favor of a slight redefinition of *set of integers* which produces a more helpful convention for present purposes.) So, if we choose, a singleton "means" a single integer. The next

question is what a "large" set $x \in \mathbf{P}\omega$ could mean. Here is an answer: if we write:

$$x = \{0, 5, 17\} = 0 \cup 5 \cup 17,$$

we are thinking of x as a *multiple* integer. This is especially useful in the case of multiple-valued function where we can write:

$$f(a) = 0 \cup 5 \cup 17.$$

Then "$m \in f(a)$" can be interpreted as "m is *one* value of f at a." Now $a \in \mathbf{P}\omega$, too, and so it is a multiple integer also. This brings us to an important point.

A multiple integer is (often) *more* than just the (random) collection of its elements. From the definition of continuity, $m \in f(a)$ is equivalent to $m \in f(e_n)$ with $e_n \subseteq a$. We may not be able to reduce this to $m \in f(\{n\})$ with $n \in a$ without additional assumptions on f. Indeed we shall take advantage of the feature of continuous functions whereby the elements of an argument a can join in *cooperation* in determining $f(a)$. Needless to say, continuity implies that the cooperation cannot extend beyond *finite* configurations, and so we can say that a *is* the union (or limit) of its finite subsets. However, finitary cooperation will be found to be quite a powerful notion.

Where does this interpretation leave the empty set $\varnothing$? When we write "$f(a) = \varnothing$" we can read this as "f has *no* value at a", or "f is *undefined* at a". In this case $f(a)$ exists (as a set), but it is not "defined" as an *integer*. Single- (or singleton) valued functions are "well-defined", but multiple-valued functions are rather "over-defined".

How does this interpretation fit in with monotonicity? In case $a \subseteq b$ and $m \in f(a)$, then we must have $m \in f(b)$. We can read "$a \subseteq b$" as "b is an *improvement* of a" is *better-defined* than a". The point of monotonicity is that the better we define an argument, the better we define a value. 'Better" does not imply "well" (that is, singleton-valuedness), and overdefinedness may well creep in. This is not the fault of the function; it is our fault for not choosing a different function.

As a subspace $\omega \subseteq \mathbf{P}\omega$ is *discrete*. This implies that *arbitrary* functions $p : \omega \to \omega$ are *continuous*. Note that $p : \omega \to \mathbf{P}\omega$ as well, because $\omega \subseteq \mathbf{P}\omega$. By Theorem 1.5 we can extend p continuously to $\bar{p} : \mathbf{P}\omega \to \mathbf{P}\omega$. The formula given produces this function:

$$\bar{p}(x) = \begin{cases} \bigcap\{p(n) \mid n \in \omega\} & \text{if } x = \varnothing; \\ p(n) & \text{if } x = n \in \omega; \\ \omega & \text{otherwise.} \end{cases} \tag{2.1}$$

This is a rather abrupt extension of p (the maximal extension); a more gradual, continuous extension (the minimal extension) is determined by this equation:

$$\hat{p}(x) = \bigcup\{p(n) \mid n \in x\}. \tag{2.2}$$

The same formulae work for *all* multiple-valued functions $p : \omega \to \mathbf{P}\omega$. Functions like $f = \hat{p}$ are exactly characterized as being those continuous functions $f : \mathbf{P}\omega \to \mathbf{P}\omega$ which in addition are *distributive* in the sense of these equations:

$$f(x \cup y) = f(x) \cup f(y) \quad \text{and} \quad f(\varnothing) = \varnothing.$$

The sets $\varnothing$ and ω play special roles. When we consider them as *elements* of $\mathbf{P}\omega$ we shall employ the notation:

$$\bot = \varnothing \quad \text{and} \quad \top = \omega.$$

The element $\bot$ is the most "undefined" integer, and $\top$ is the most "overdefined". All others are in between.

One last general point on meaning: suppose $x \in \mathbf{P}\omega$ and $k \in x$. Then $k = (n, m)$ for suitable (uniquely determined) integers n and m. That is to say, every element of x can be regarded as an *ordered pair*; thus, x can be used as a *relation*. Such an operation as

$$x; y = \{(n, l) | \exists m.(n, m) \in x, (m, l) \in y\} \tag{2.3}$$

is then a continuous function of two variables that treats both x and y as relations. On the other hand we could define a quite different continuous function such as

$$x + y = \{n + m | n \in x, m \in y\} \tag{2.4}$$

which treats both x and y *arithmetically*. The only reason we shall probably never write $(x + y)$; x again is that the values of this perfectly well-defined continuous function are, for the most part, quite uninteresting. There is, however, no theoretical reason why we cannot use the same set with several different "meanings" in the same formula. Of course if we do so, it is to be expected that we will show the point of doing this in the special case. We turn now to the definition of the general language for defining all such functions.

The *syntax* and *semantics* of the language LAMBDA are set out in Table 2. The syntax is indicated on the left, and the meanings of the combinations are shown on the right as subsets of $\mathbf{P}\omega$. This is the basic language and could have been given (less understandably) in terms of combinators (see Theorem 2.4). It is, however, a very primitive language, and we shall require many definitions before we can see why such functions as in (2.3) and (2.4) are themselves definable.

TABLE 2

The language LAMBDA

$0 = \{0\}$
$x+1 = \{n+1 \mid n \in x\}$
$x-1 = \{n \mid n+1 \in x\}$
$z \supset x, y = \{n \in x \mid 0 \in z\} \cup \{m \in y \mid \exists k.k+1 \in z\}$
$u(x) = \{m \mid \exists e_n \subseteq x.(n, m) \in u\}$
$\lambda x.\tau = \{(n, m) \mid m \in \tau[e_n/x]\}$

The definition has been left somewhat informal in hopes that it will be more understandable. In the above, τ is any term of the language. LAMBDA is *type-free* and allows any combination to be made by substitution into the given functions. There is one primitive constant (0); there are two unary functions $(x+1, x-1)$; there is one binary function $(u(x))$ and one ternary function $(z \supset x, y)$; finally there is one variable binding operator $(\lambda x.\tau)$. The first three equations have obvious sense. In the fourth, $z \supset x, y$ is McCarthy's *conditional expression* (a test for *zero*). Next $u(x)$ defines *application* (u is treated as a graph and x as a set), and $\lambda x.\tau$ is *functional abstraction* (compare the definition of **fun**). In defining $\lambda x.\tau$, we use $\tau[e_n/x]$ as a shorthand for *evaluating* the term τ when the variable x is given the value e_n.

Note that the functions are all multiple-valued. Thus we have such a result as:

$$(6 \cup 10)+1 = 7 \cup 11. \tag{2.5}$$

The partial character of subtraction has expression as:

$$0-1 = \perp. \tag{2.6}$$

We shall see how to define + and − in LAMBDA later. The conditional could also have been defined by cases:

$$z \supset x, y = \begin{cases} \perp & \text{if } z = \perp; \\ x & \text{if } z = 0; \\ y & \text{if } 0 \notin z \neq \perp; \\ x \cup y & \text{if } 0 \in z \neq 0. \end{cases} \tag{2.7}$$

We say that a LAMBDA-term τ defines a function of its free variables (at least). Other results depend on this fundamental proposition:

THEOREM 2.1 (The continuity theorem). *All LAMBDA definable functions are continuous.*

Once that is proved, we can use Theorem 1.2 to establish:

THEOREM 2.2 (The conversion theorem). *The three basic principles* (α), (β), (ξ) *of* λ*-conversion are all valid in the model.*

By "model" here we of course understand the interpretation of the language where the semantics gives terms denotations in $\mathbf{P}\omega$ according to the stated definition. Through this interpretation, more properly speaking, $\mathbf{P}\omega$ becomes a model for the axioms (α), (β), (ξ). Two well-known results of the calculus of λ-conversion allow the reduction of functions of several variables to those of *one*, and the reduction of all the primitives to *combinators* (constants)—all this with the aid of the binary operation of application.

THEOREM 2.3 (The reduction theorem). *Any continuous function of* k*-variables can be written as*

$$f(x_0, x_1, \cdots, x_{k-1}) = u(x_1) \cdots (x_{k-1}),$$

where u *is a suitably chosen element of* $\mathbf{P}\omega$.

THEOREM 2.4 (The combinator theorem). *The LAMBDA-definable func-*

tions can be generated (from variables) by iterated application with the aid of these six constants:

$$\mathbf{0} = 0$$

$$\mathbf{suc} = \lambda x.x + 1$$

$$\mathbf{pred} = \lambda x.x - 1$$

$$\mathbf{cond} = \lambda x \lambda y \lambda z.z \supset x, y$$

$$\mathbf{K} = \lambda x \lambda y.x$$

$$\mathbf{S} = \lambda u \lambda v \lambda x.u(x)(v(x))$$

But the result that makes all this model building and combinatory logic *mathematically* interesting concerns the so-called *paradoxical combinator* defined by the equation:

$$\mathbf{Y} = \lambda u.(\lambda x.u(x(x)))(\lambda x.u(x(x))). \tag{2.8}$$

THEOREM 2.5 (The first recursion theorem). *If u is the graph of a continuous function f, then* $\mathbf{Y}(u) = \mathbf{fix}(f)$, *the least fixed point of f.*

There are two points to note here: the fixed point is LAMBDA-definable if f is; and $\mathbf{Y}$ defines a *continuous* operator. The word "recursion" is attached to the theorem because fixed points are employed to solve recursion equations. It would not be correct to call the fixed-point theorem (Theorem 1.4) the recursion theorem since it only shows that fixed points *exist* and not how they are *definable* in a language. The *second* recursion theorem (in Kleene's terminology) is related, but it involves Gödel numbers as introduced in § 3.

From this point on we see no need to distinguish continuous functions from elements of $\mathbf{P}\omega$; a continuous function will be *identified* with its graph. Note that u is a graph iff $u = \lambda x.u(x)$, which is equivalent to Theorem 1.2 (iii). For this reason (functions are graphs) we propose the name *Graph Model* for this model of the λ-calculus. (There is more to LAMBDA than just λ, however.)

The identification of functions with graphs entails that the function *space* of all continuous functions from $\mathbf{P}\omega$ into $\mathbf{P}\omega$ is to be identified (one-to-one) with the subspace

$$\mathbf{FUN} = \{u | u = \lambda x.u(x)\} \subseteq \mathbf{P}\omega.$$

The identification is *topological* in that the subspace topology agree with the product topology on the function space. This is the topology of pointwise convergence and is closely connected with the lattice structure on the function space which is also defined pointwise (that is, argumentwise). In the notation of λ-abstraction we can express this as the extension of the axiom of extensionality called (ξ^*) in Table 1 of the Introduction. The laws in Table 1 are not the only ones valid in the model, however. We may also note such argumentwise distribution

laws as:

$$(f \cup g)(x) = f(x) \cup g(x); \tag{2.9}$$

$$(\lambda x.\tau) \cup (\lambda x.\sigma) = \lambda x.(\tau \cup \sigma); \tag{2.10}$$

$$(f \cap g)(x) = f(x) \cap g(x); \tag{2.11}$$

$$(\lambda x.\tau) \cap (\lambda x.\sigma) = \lambda x.(\tau \cap \sigma). \tag{2.12}$$

In the above f and g must be graphs. It is also true that if $\mathscr{F} \subseteq \mathbf{P}\omega$, then

$$\bigcup\{f | f \in \mathscr{F}\}(x) = \bigcup\{f(x) | f \in \mathscr{F}\}, \tag{2.13}$$

but the same does not hold for $\bigcap$.

We state now a sequence of minor results which show why some simple functions and constants are LAMBDA-definable.

$$\bot = (\lambda x.x(x))(\lambda x.x(x)); \tag{2.14}$$

$$x \cup y = (\lambda z.0) \supset x, y; \qquad (\textit{Hint}: 0, 1 \in \lambda z.0) \tag{2.15}$$

$$\top = \mathbf{Y}(\lambda x.0 \cup (x+1)); \tag{2.16}$$

$$x \cap y = \mathbf{Y}(\lambda f \lambda x \lambda y.x \supset (y \supset 0, \bot), f(x-1)(y-1)+1)(x)(y). \tag{2.17}$$

The elements $\bot$ and $\top$ are graphs, by the way, and we can characterize them as the only fixed points of the combinator $\mathbf{K}$:

$$a = \lambda x.a \quad \text{iff} \quad a = \bot \quad \text{or} \quad a = \top. \tag{2.18}$$

Next we use the notation $\langle x_0, x_1, \cdots, x_{n-1}\rangle$ for the function $\hat{p}$ where $p : \omega \to \mathbf{P}\omega$ is defined by:

$$p(i) = \begin{cases} x_i & \text{if } i < n; \\ \bot & \text{if } i \geqq n. \end{cases}$$

$$\langle\rangle = \bot \tag{2.19}$$

$$\langle x\rangle = \lambda z.z \supset x, \bot \tag{2.20}$$

$$\langle x, y\rangle = \lambda z.z \supset x, (z-1 \supset y, \bot) \tag{2.21}$$

$$\langle x_0, x_1, \cdots, x_n\rangle = \lambda z.z \supset x_0, \langle x_1, \cdots, x_n\rangle(z-1). \tag{2.22}$$

Obviously we should formalize the subscript notation so that $u_x = u(x)$; then we find:

$$\langle x_0, x_1, \cdots, x_{n-1}\rangle_i = \begin{cases} x_i & \text{if } i < n; \\ \bot & \text{if } i \geq n. \end{cases} \tag{2.23}$$

This gives us the method of LAMBDA-defining *finite* sequences (in a quite natural way), and the next step is to consider *infinite* sequences. But these are just the functions $\hat{p}$ where $p : \omega \to \mathbf{P}\omega$ is arbitrary. What we need then is a condition expressible in the language equivalent to saying $u = \hat{p}$ for some p. This is the same as

$$u = \lambda x.\bigcup\{u_i | i \in x\},$$

but the $\bigcup$- and set-notation is not part of LAMBDA. We are forced into a recursive definition:

$$\$ = \mathbf{Y}(\lambda s \lambda u \lambda z.z \supset u_0, s(\lambda t.u_{t+1})(z-1)). \tag{2.24}$$

This equation generalizes (2.22) and we have:

$$\$(u) = \lambda x.\bigcup\{u_i | i \in x\}. \tag{2.25}$$

Thus the combinator \$ "revalues" an element as a distributive function. This suggests introducing the λ-notation for such functions by the equation:

$$\lambda n \in \omega.\tau = \$(\lambda z.\tau[z/n]). \tag{2.26}$$

With all these conventions LAMBDA-notation becomes very much like ordinary mathematical notation without too much strain.

Suppose that f is any continuous function and $a \in \mathbf{P}\omega$. We can define $p : \omega \to \mathbf{P}\omega$ in the ordinary way by *primitive recursion* where:

$$p(0) = a;$$

$$p(n+1) = f(n)(p(n)).$$

The question is: can we give a LAMBDA-definition for $\hat{p}$ (in terms of f and a as constants, say)? The answer is clear, for we can prove:

$$\hat{p} = \mathbf{Y}(\lambda u \lambda n \in \omega.n \supset a, f(n-1)(u(n-1))). \tag{2.27}$$

This already shows that a large part of the definitions of recursion theory can be given in this special language. Of course, *simultaneous* (primitive) recursions can be transcribed into LAMBDA with the aid of the ordered tuples of (2.22), (2.23) above. But we can go further and connect with partial (and general) recursive functions. We state first a definition.

DEFINITION. A continuous function f of k-variables is *computable* iff the relationship

$$m \in f(e_{n_0})(e_{n_1}) \cdots (e_{n_{k-1}})$$

is recursively enumerable (r.e.) in the integer variables $m, n_0, n_1, \cdots, n_{k-1}$.

If q is a *partial* recursive function in the usual sense, then we can regard it as a mapping $q : \omega \to \omega \cup \{\perp\}$, where $q(n) = \perp$ means that q is undefined at n. Saying that q is partial *recursive* is just to say that $m \in q(n)$ is r.e. as a relation in n and m. It is easy to see that this is in turn equivalent to the recursive enumerability of the relationship $m \in \hat{q}(e_n)$; and so our definition is formally a generalization of the usual one. But it is also intuitively reasonable. To "compute" $y = f(x)$, in the one-variable case, we proceed by enumeration. First we begin the enumeration of all finite subsets $e_n \subseteq x$. For each of these f starts up an enumeration of the set $f(e_n)$; so we sit back and observe which $m \in f(e_n)$ by enumeration. The totality of all such m for all $e_n \subseteq x$ forms in the end the set y.

THEOREM 2.6 (The definability theorem). *For a k-ary continuous function f, the following are equivalent*:

(i) *f is computable*;

(ii) $\lambda x_0 \lambda x_1 \cdots \lambda x_{k-1}.f(x_0)(x_1) \cdots (x_{k-1})$ *as a set is r.e.*;

(iii) $\lambda x_0 \lambda x_1 \cdots \lambda x_{k-1}.f(x_0)(x_1) \cdots (x_{k-1})$ *is LAMBDA-definable.*

As a hint for the proof we may note that the method of (2.27) shows that all primitive recursive functions p have the corresponding $\hat{p}$ LAMBDA-definable. Next we remark that a nonempty r.e. set is the range of a primitive recursive function; but the range of p is $\hat{p}(\top)$, which is clearly LAMBDA-definable. That any LAMBDA-definable set (graph) is r.e. is obvious from the definition of the language itself. More details are given in the Appendix.

We may draw some interesting conclusions from the definability theorem. In the first place, we see that the countable collection $\mathbf{RE} \subseteq \mathbf{P}\omega$ of r.e. sets is *closed* under application and LAMBDA-definability. Indeed it forms a model for the λ-calculus (axioms (α), (β), (ξ^*) at least) and it also contains the arithmetical combinators. (Clearly there will be *many* intermediate submodels.) In the second place, we can see now how very easy it is to interpret λ-calculus in ordinary arithmetical recursion theory by means of quite elementary operations on r.e. sets. Thus the equivalence of λ-definability with partial recursiveness seems not to be all that good a piece of evidence for Church's Thesis. In his 1936 paper (a footnote on p. 346) Church says about λ-definability:

> The fact, however, that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of the reasons adduced below for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it.

The point never struck the present author as an especially telling one, and the reduction of λ-calculus to r.e. theory shows that the divergence between the theories is not at all wide. Of course it is a pleasant surprise to see how many complicated things can be defined in *pure* λ-calculus (without arithmetical combinators), but this fact cuts the wrong way as evidence for the thesis (we want stronger theories, not weaker ones). Post systems (or even first-order theories) are much better to mention in this connection, since they are obviously *more* inclusive in giving enumerations than Turing machines or Herbrand–Gödel recursion equations. But the equivalence proofs are all so easy! What one would like to see is a "natural" definition where the equivalence with r.e. is not just a mechanical exercise involving a few tricks of coding.

In the course of the development in this section we have stated many equations which are not found in Table 1, and which involve new combinators. In conclusion we would like to mention an equation about $\mathbf{Y}$ which holds in the model, which can be stated in pure λ-calculus, and which cannot be proved by ordinary reduction (though we shall not try to justify this last statement here). In order to shorten the calculations, we note from definition (2.8) that $\mathbf{Y}(u) = \mathbf{Y}(\lambda y.u(y))$; so by Theorem 2.5 this also equals $u(\mathbf{Y}(u))$.

$$\mathbf{Y}(\lambda f \lambda x.g(x)(f(x))) = \lambda x.\mathbf{Y}(g(x)). \tag{2.28}$$

Call the left side f' and the right f''. Now

$$f'' = \lambda x.g(x)(\mathbf{Y}(g(x)) = \lambda x.g(x)(f''(x)),$$

thus $f' \subseteq f''$, because f' is a least fixed point. On the other hand $f' = \lambda x.g(x)(f'(x))$, so $f'(x) = g(x)(f'(x))$. Thus $f''(x) \subseteq f'(x)$, because $f''(x)$ is a least fixed point. As this holds for all x, we see that $f'' \subseteq f'$; and so they are equal. There must be many other such equations.

3. Enumeration and degrees. A great advantage of the combinators from the formal point of view is that (bound) variables are eliminated in favor of "algebraic" combinations. The disadvantage is that the algebra is not all that pretty, as the combinations tend to get rather long and general laws are rather few. Nevertheless as a technical device it is mildly remarkable that we can have a notation for all r.e. sets requiring so few primitives. In the model defined here the reduction to one combinator rests on a lemma about conditionals:

$$\mathbf{cond}(x)(y)(\mathbf{cond}(x)(y)) = y. \tag{3.1}$$

Recall that **cond** (or $\supset$) is a test for zero, so that:

$$\mathbf{cond}(x)(y)(0) = x. \tag{3.2}$$

This suggests that we lump all combinators of Theorem 2.4 into this one:

$$\mathbf{G} = \mathbf{cond}(\langle \mathbf{suc}, \mathbf{pred}, \mathbf{cond}, \mathbf{K}, \mathbf{S}\rangle)(0). \tag{3.3}$$

We can then readily prove:

THEOREM 3.1 (The generator theorem). *All LAMBDA-definable elements can be obtained from* **G** *by iterated application.*

A distributive function f is said to be *total* iff $f(n) \in \omega$ for all $n \in \omega$. As they come from obvious primitive recursive functions, we do not stop to write out LAMBDA-definitions of these three total functions:

$$\mathbf{apply} = \lambda n \in \omega.\lambda m \in \omega.(n, m) + 1 \tag{3.4}$$

$$\mathbf{op}((n, m)) = n \tag{3.5}$$

$$\mathbf{arg}((n, m)) = m. \tag{3.6}$$

The point of these auxiliary combinators concerns our Gödel numbering of the r.e. sets. The number 0 will correspond to the generator **G**; while $(n, m)+1$ will correspond to the application of the nth set to the mth. This is formalized in the combinator **val** which is defined as the least fixed point of the equation:

$$\mathbf{val} = \lambda k \in \omega.k \supset \mathbf{G}, \mathbf{val}(\mathbf{op}(k-1))(\mathbf{val}(\mathbf{arg}(k-1))). \tag{3.7}$$

This function accomplishes the enumeration as follows:

THEOREM 3.2 (The enumeration theorem). *The combinator* **val** *enumerates the LAMBDA-definable elements in that* $\mathbf{RE} = \{\mathbf{val}(n) | n \in \omega\}$. *Further*:

(i) $\mathbf{val}(0) = \mathbf{G}$,

(ii) $\mathbf{val}(\mathbf{apply}(n)(m)) = \mathbf{val}(n)(\mathbf{val}(m))$.

As a principal application of the Enumeration Theorem we may mention the following: suppose u is *given* as LAMBDA-definable. We look at its definition and rewrite it in terms of combinators—eventually in terms of **G** alone. Then using 0 and apply we write down the name of an integer corresponding to the combination—say, n. By Theorem 3.2 we see that we have *effectively found* from

the definition an integer such that $\mathbf{val}\, n = u$. This remark can be strengthened by some numerology.

(3.8) $\mathbf{apply}(0)(0) = 1$ and $\mathbf{val}(1) = 0$;

(3.9) $\mathbf{apply}(0)(1) = 3$ and $\mathbf{val}(3) = \langle \mathbf{suc}, \cdots \rangle$;

(3.10) $\mathbf{apply}(3)(1) = 12$ and $\mathbf{val}(12) = \mathbf{suc}$.

Thus, define as the least fixed point:

(3.11) $$\mathbf{num} = \lambda n \in \omega . n \supset 1, \mathbf{apply}(12)(\mathbf{num}(n-1)),$$

and derive the equation for all $n \in \omega$:

(3.12) $$\mathbf{val}(\mathbf{num}(n)) = n.$$

We note that **num** is a primitive recursive (total) function. The combinator **num** allows us now to effectively find a LAMBDA-definition, corresponding to a given LAMBDA-definable element u, of an element v such that uniformly in the integer variable n we have $\mathbf{val}(v(n)) = u(n)$. Further, v is a primitive recursive (total) function. This is the technique involved in the proof of Kleene's well-known result:

THEOREM 3.3 (The second recursion theorem). *Take a LAMBDA-definable element v such that*:

(i) $$\mathbf{val}(v(n)) = \lambda m \in \omega . \mathbf{val}(n)(\mathbf{apply}(m)(\mathbf{num}(m))),$$

and then define a combinator by:

(ii) $$\mathbf{rec} = \lambda n \in \omega . \mathbf{apply}(v(n))(\mathbf{num}(v(n))).$$

Then we have a primitive recursive function with this fixed-point property:

(iii) $$\mathbf{val}(\mathbf{rec}(n)) = \mathbf{val}(n)(\mathbf{rec}(n)).$$

Note that if u is LAMBDA-definable, then we find first an n such that $\mathbf{val}(n) = u$. Next we calculate $k = \mathbf{rec}(n)$. This effectively gives us an integer such that $\mathbf{val}(k) = u(k)$. Gödel numbers represent expressions (combinations in **G**), and **val** maps the numbers to the values denoted by the expressions in the model. The k just found thus represents an expression whose value is defined in terms of its *own* Gödel number. In recursion theory there are many applications of this result. Another familiar argument shows:

THEOREM 3.4 (The incompleteness theorem). *The set of integers n such that* $\mathbf{val}(n) = \bot$ *is not r.e.; hence, there can be no effectively given formal system for enumerating all true equations between LAMBDA-terms.*

(A critic may sense here an application of Church's thesis in stating the metatheoretic consequence of the nonresult.) A few details of the proof can be given to see how the notation works. First let v be a (total) primitive recursive function such that:

$$\mathbf{val}(v(n)) = n \cap \mathbf{val}(n).$$

and note that:

$$n \cap \mathbf{val}(n) = \bot \quad \text{iff} \quad n \notin \mathbf{val}(n).$$

Call the set in question in Theorem 3.4 the set b. If it were r.e., then so would be:

$$\{n \in \omega | v(n) \in b\} = (\lambda n \in \omega . v(n) \cap b \supset n, n)(\top).$$

That would mean having an integer k such that:

$$\mathbf{val}(k) = \{n \in \omega | v(n) \in b\}.$$

But then:

$$\begin{aligned} k \in \mathbf{val}(k) \quad &\text{iff} \quad v(k) \in b \\ &\text{iff} \quad \mathbf{val}(v(k)) = \bot \\ &\text{iff} \quad k \notin \mathbf{val}(k), \end{aligned}$$

which gives us a contradiction. This is the usual diagonal argument.

The relationship $\mathbf{val}(n) = \mathbf{val}(m)$ means that the expressions with Gödel numbers n and m have the *same* value in the model. (This is not only not r.e. but is a complete Π^0_2-predicate.) A total mapping can be regarded as a *syntactical* transformation on expressions defined via Gödel numbers. Such a mapping p is called *extensional* if it has the property:

$$\mathbf{val}(p(n)) = \mathbf{val}(p(m)) \quad \textit{whenever} \quad \mathbf{val}(n) = \mathbf{val}(m).$$

The Myhill–Shepherdson theorem shows that extensional, syntactical mappings really depend on the *values* of the expressions. Precisely we have:

THEOREM 3.5 (The completeness theorem for definability). *If a (total) extensional mapping p is LAMBDA-definable, then there is a LAMBDA-definable q such that* $\mathbf{val}(p(n)) = q(\mathbf{val}(n))$ *for all* $n \in \omega$.

Of course q is uniquely determined (because the values of q are given at least on the finite sets). Thus any attempt to define something *new* by means of some strange mapping on Gödel numbers is bound to fail as long as it is *effective* and *extensional.* The main part of the argument is concentrated on showing these mappings to be *continuous*; that is why q exists.

The preceding results indicate that the expected results on r.e. sets are forthcoming in a smooth and unified manner in this setting. Some knowledge of r.e. theory was presupposed, but analysis shows that the knowledge required is slight. The notion of primitive recursive functions should certainly be well understood together with standard examples. Partial functions need not be introduced separately since they are naturally incorporated into LAMBDA (the theory of multiple-valued functions). As a working definition of r.e. one can take either "empty or the range of a primitive recursive function" or, more uniformly, "a set of the form $\{m | \exists n . m + 1 = p(n)\}$ where p is primitive recursive". A few obvious closure properties of r.e. sets should then be proved, and then an adequate foundation for the discussion of LAMBDA will have been provided. The point of introducing LAMBDA is that further closure properties are more easily expressed in a theory where equations can be variously interpreted as involving numbers, functionals, etc., without becoming too heavily involved in intricate Gödel numbering and encodings. Another useful feature of the present theory concerns the ease with which we can introduce *relative recursiveness.*

As we have seen, $\{\mathbf{val}(n)|n \in \omega\}$ is an enumeration of all r.e. sets. Suppose we add a new set a as a new *constant.* What are the sets enumerable in a? Answer: $\{\mathbf{val}(n)(a)|n \in \omega\}$, since in combinatory logic a parameter can always be factored out as an extra argument. Another way to put the point is this: for b to be *enumeration reducible* to a it is necessary and sufficient that $b = u(a)$ where $u \in \mathbf{RE}$. This is *word for word* the definition given by Rogers (1967, pp. 146–147). What we have done is to put the theory of enumeration operators (Friedberg–Rogers and Myhill–Shepherdson) into a general setting in which the language LAMBDA not only provides definitions but also the basis of a calculus for demonstrating properties of the operators defined. The algebraic style of this language throws a little light on the notion of enumeration degree. In the first place we can identify the degree of an element with the set of all objects reducible to it (rather than just those equivalent to it) and write

$$\mathbf{Deg}(a) = \{u(a)|u \in \mathbf{RE}\}.$$

The set-theoretical inclusion is then the same as the partial ordering of degrees. What kind of a partially ordered set do we have?

THEOREM 3.6 (The subalgebra theorem). *The enumeration degrees are exactly the finitely generated combinatory subalgebras of* $\mathbf{P}\omega$.

By "subalgebras" here we of course mean subsets containing **G** and closed under application (hence, they contain all of **RE**, the least subalgebra). Part of the assertion is that every *finitely* generated subalgebra has a *single* generator (under application). This fact is an easy extension of Theorem 3.1. Not very much seems to be known about enumeration degrees. *Joins* can obviously be formed using the pairing function $\langle x, y\rangle$ on sets. Each degree is a countable set; hence, it is trivial to obtain the existence of a sequence of degrees whose *infinite join* is not a degree (not finitely generated). The intersection of subalgebras is a subalgebra—but it may not be a degree even starting with degrees. There are no minimal degrees above **RE**, but there are minimal *pairs* of degrees. Also for a given degree there are only countably many degrees minimal over it; but the question of whether the partial ordering of enumeration degrees is dense seems still to be open.

Theorem 3.6 shows that the semilattice of enumeration degrees is naturally extendable to a *complete* lattice (the lattice of all subalgebras of $\mathbf{P}\omega$), but whether there is anything interesting to say about this complete lattice from the point of view of structure is not at all clear. Rogers has shown (1967, pp. 151–153) that Turing degrees can be defined in terms of enumeration degrees by restricting to special elements. In our style of notation we would define the space:

$$\mathbf{TOT} = \{u|u = \$(u) \text{ and } \forall n \in \omega.u(n) \in \omega\},$$

the space of all graphs of total functions. Then the system $\{\mathbf{Deg}(u)|u \in \mathbf{TOT}\}$ is isomorphic to the system of Turing degrees. Now there are many other interesting subsets of $\mathbf{P}\omega$. Whether the degree structure of these various subsets is worth investigation is a question whose answer awaits some new ideas.

Among the subsets of $\mathbf{P}\omega$ with natural mathematical structure, we of course have **FUN**, which is a semigroup under $\circ = \lambda u\lambda v\lambda x.u(v(x))$. It is, however, a rather

complicated semigroup. We introduce for its study three new combinators:

(3.13) $\mathbf{R} = \lambda x.\langle 0, x\rangle;$

(3.14) $\mathbf{L} = \lambda x.x_1(x_2);$

(3.15) $\bar{u} = \lambda x.x_0 \supset \langle 1, u, x_1\rangle, \overline{u(x_1)}(x_2).$

THEOREM 3.7 (The semigroup theorem). *The countable semigroup* $\mathbf{RE} \cap \mathbf{FUN}$ *of computable enumeration operators is finitely generated by* $\mathbf{R}$, $\mathbf{L}$ *and* $\bar{\mathbf{G}}$.

The proof rests on the verification of two equations which permit an application of Theorem 3.1:

(3.16) $\mathbf{L} \circ \bar{u} \circ \mathbf{R} = \lambda x.u(x)$

(3.17) $\bar{u} \circ \bar{v} \circ \mathbf{R} = u(v).$

Certainly the word problem for $\mathbf{RE} \cap \mathbf{FUN}$ is unsolvable, indeed, not even recursively enumerable. Can the semigroup be generated by *two* generators by the way?

4. Retracts and data types. Data can be structured in many ways: ordered tuples, lists, arrays, trees, streams, and even operations and functions. The last point becomes clear if one thinks of *parameters*. We would normally hardly consider the pairing function $\lambda x\lambda y.\langle x, y\rangle$ as being in itself a piece of data. But if we treat the first variable as a parameter, then it can be specialized to a *fixed value*, say the element a, producing the function $\lambda y.\langle a, y\rangle$. This function is more likely to be the output of some process and *in itself* can be considered as a datum. It is rather like one *whole row* of a matrix. If we were to regard a two-argument function f as being a matrix, then its ath row would be exactly $\lambda y.f(a)(y)$. If s were a selection function, then, for example, $\lambda y.f(s(y))(y)$ would represent the selection of one element out of each column of the matrix. This selection could be taken as a specialization of parameters in the operator $\lambda u\lambda v\lambda y.u(v(y))(y)$. We have not been very definite here about the exact nature of the fixed a, f, or s, or the range of the variable y or the range of values of the function f. The point is only to recall a few elements of structure and to suggest an abstract view of data going beyond the usual iterated arrays and trees.

What then is a *data type*? Answer: a type of data. That is to say, a collection of data that have been grouped together for reasons of similarity of structure or perhaps mere convenience. Thus the collection may very well be a mixed bag, but more often than not canons of taste or demands of simplicity dictate an adherence to regularity. The grouping may be formed to eliminate irrelevant objects and focus the attention in other ways. It is frequently a matter of good organization that aids the understanding of complex definitions. In programming languages, one of the major reasons for making an explicit declaration of a data type (that is, the restriction of certain variables to certain "modes") is that the computed objects of that type can enjoy a special *representation* in the machine that allows the manipulation of these objects via the chosen representation to be reasonably efficient. This is a very critical matter for good language design and good compiler writing. In this report, however, we cannot discuss the problems of representation,

important as they may be. Our objective here is conceptual organization, and we wish to show how such ideas, in the language for computable functions used here, can find the proper expression.

Which are the data types that can be defined in LAMBDA? No final answer can be given since the number is infinite and inexhaustible. From one point of view, however, there is only one: $\mathbf{P}\omega$ itself. It is the universal type and all other types are subtypes of it; so $\mathbf{P}\omega$ plays a primary role in this exposition. But in a way it is too big, or at least too complex, since each of its elements can be used in so many different ways. When we specify a subtype the intention is to restrict attention to a special use. But even the various subtypes overlap, and so the same elements still get different uses. Style in writing definitions will usually make the differentiation clear though. The main innovation to be described in this section is the use of LAMBDA expressions to define types *as well as* elements. Certain expressions define *retracts* (or better: retraction mappings), and it is the *ranges* (or as we shall see: *sets of finite points*) of such retracts that form the groupings into types. Thus LAMBDA provides a calculus of type definitions including *recursive* type definitions. Examples will be explained both here and in the following sections. Note that types as retracts turn out to be types as *lattices*, that is, types of partial and many-valued objects. The problem of cutting these lattice types down to the perfect or complete objects is discussed in § 6. Another view of types and functionality of mappings is presented in § 7.

The notion of a retract comes from (analytic) topology, but it seems almost an accident that the idea can be applied in the present context. The word is employed not because there is some deep tie-up with topology but because it is short and rather descriptive. Three easy examples will motivate the general plan:

$$\textbf{fun} = \lambda u \lambda x.u(x); \tag{4.1}$$

$$\textbf{pair} = \lambda u.\langle u_0, u_1\rangle; \tag{4.2}$$

$$\textbf{bool} = \lambda u.u \trianglerighteq 0, 1. \tag{4.3}$$

Here $\trianglerighteq$ is the *doubly strict* conditional defined by

$$z \trianglerighteq x, y = z \supset (z \supset x, \top), (z \supset \top, y), \tag{4.4}$$

which has the property that if z is both zero and positive, then it takes the value $\top$ instead of the value $x \cup y$.

DEFINITION. An element $a \in \mathbf{P}\omega$ is called a *retract* iff it satisfies the equation $a = a \circ a$.

Of course the $\circ$-notation is used for functional composition in the standard way:

$$u \circ v = \lambda x.u(v(x)). \tag{4.5}$$

And it is quite simple to prove that each of the three combinators in (4.1)–(4.3) is a retract according to the definition. But what is the point?

Consider **fun**. No matter what $u \in \mathbf{P}\omega$ we take, $\textbf{fun}(u)$ is (the graph of) a *function*. And if u already is (the graph of) a function, then $u = \textbf{fun}(u)$. That is to say, the *range* of **fun** is the same as the set of *fixed points* of **fun** is the same as the set of all (graphs of) functions. Any mapping a whose range and fixed-point set

coincide satisfies $a = a \circ a$, and conversely. A retract is a mapping which "retracts" the whole space onto its range *and* which is the identity mapping on its range. That is the import of the equation $a = a \circ a$. Strictly speaking, the range is the retract and the mapping is the *retraction*, but for us the mapping is more important. (Note, however, that distinct retracts can have the same range.) We let the mapping stand in for the range.

Thus the combinator **fun** represents in itself the concept of a *function* (continuous function on $\mathbf{P}\omega$ into $\mathbf{P}\omega$). Similarly **pair**, represents the idea of a *pair* and **bool** the idea of being a boolean value as an element of $\{\bot, 0, 1, \top\}$, since we must think in the multiple-valued mode. What is curious (and, as we shall see, useful) is that all these retracts which are defining *subspaces* are at the same time *elements* of $\mathbf{P}\omega$.

DEFINITION. If a is a retract, we write $u : a$ for $u = a(u)$ and $\lambda u : a.\tau$ for $\lambda u.\tau[a(u)/u]$.

Since retracts are *sets* in $\mathbf{P}\omega$, we cannot use the ordinary membership symbol to signify that u *belongs* to the range of a; so we write $u : a$. The other notation with the λ-operator *restricts* a function to the range of a. For f to be so restricted simply means $f = f \circ a$. For the *range* of f to be *contained in* that of the retract a means $f = a \circ f$. These algebraic equations will be found to be quite handy. We are going to have a calculus of retracts *and* mappings between them involving many operators on retracts yet to be discovered. Before we turn to this calculus, we recall the well-known connection between lattices and fixed points.

THEOREM 4.1 (The lattice theorem). *The fixed points of any continuous function form a complete lattice (under* $\subseteq$*); while those of a retract form a continuous lattice.*

We note further that by the embedding theorem (Theorem 1.6), it follows that any separable (by which we mean countably-based) continuous lattice is a retract of $\mathbf{P}\omega$; hence, our universal space is indeed rich in retracts. A very odd point is that $a = a \circ a$ is a fixed-point equation itself ($\lambda u.u \circ u$ is obviously continuous). Thus the retraction mappings form a complete lattice. Is this a continuous lattice? (Ershov has proved it is not; see the Appendix for a sketch.) A related question is solved positively in the next section. Actually the ordering of retracts under $\subseteq$ does not seem to be all that interesting; a more algebraic ordering is given by:

DEFINITION. For retracts a and b we write $a \Subset b$ for $a = a \circ b = b \circ a$.

The idea here should be clear: $a \Subset b$ means that a is a retract *of* b. It is easy to prove the:

THEOREM 4.2 (The partial ordering theorem). *The retracts are partially ordered by* $\Subset$.

There do not seem to be any lattice properties of $\Subset$ of a general nature. Note, however, that if retracts *commute*, $a \circ b = b \circ a$, then $a \circ b$ is the greatest lower bound under $\Subset$ of a and b. Also if we have a sequence where both $a_n \Subset a_{n+1}$ and $a_n \subseteq a_{n+1}$ for all $n \in \omega$, then $\bigcup\{a_n | n \in \omega\}$ is the upper bound for the a_n under $\Subset$, as can easily be argued from the definition by continuity of $\circ$.

Certainly there is no "least" retract under $\Subset$. One has $\bot = \bot \circ a$ (recall: $\bot = \lambda x.\bot$), but not $a \circ \bot = \bot$. This last equation means more simply that $a(\bot) = \bot$; that is, a is *strict*. For retracts strictness is thus equivalent to $\bot \Subset a$, so we can say that there is a least strict retract. The combinator $\mathbf{I} = \lambda u.u$ clearly represents the

largest retract (the *whole* space), and it is strict also. In a certain sense strictness can be assumed without loss of generality. For if a is not strict, let

$$b = \lambda x.\{n | a(x) \neq a(\bot)\}.$$

This function takes values in $\{\bot, \top\}$ and is continuous because $\{x \in \mathbf{P}\omega | a(x) \neq a(\bot)\}$ is *open*. Next define:

$$a^* = \lambda u.a(u) \cap b(u).$$

This is a strict retract whose range is homeomorphic (and lattice isomorphic) to that of a. Note, however, that the mapping from a to a^* is not continuous (or even monotonic).

To have a more uniform notation for retracts we shall often write **nil**, **id**, and **seq** for the combinators $\bot$, **I**, \$. Two further retracts of interest are

$$\textbf{open} = \lambda u.\{m | \exists e_n \subseteq e_m . n \in u\}; \tag{4.6}$$

$$\textbf{int} = \lambda u.u \sqsupseteq 0, \textbf{int}(u-1) \sqsupseteq u, u. \tag{4.7}$$

The range of **open** is lattice isomorphic to the lattice of open subsets of $\mathbf{P}\omega$; definition (4.6) is not a LAMBDA-definition of the retract, but such can be given.

In (4.7) we intend **int** to be the least fixed point of the equation. By induction on the least element of u (if any) one proves that:

$$\textbf{int}(u) = \begin{cases} \bot & \text{if } u = \bot; \\ u & \text{if } u \in \omega; \\ \top & \text{otherwise.} \end{cases}$$

This retract wipes out the distinctions between multiple values, moving all above the singletons up to $\top$; its range thus has a very simple structure. The retract **int** clearly generalizes **bool**. The range of **fun** is homeomorphic to the space of all continuous functions from $\mathbf{P}\omega$ into $\mathbf{P}\omega$; the range of **pair**, to the space of all ordered pairs; the range of **seq**, to the space of all infinite sequences. A combination like $\lambda u.\textbf{int} \circ \textbf{seq}(u)$ is a retract whose range is homeomorphic to the space of infinite sequences of elements from the range of **int**.

We now wish to introduce some operators that provide systematic ways of forming new combinations of retracts. There are three principal ones:

$$a \multimap b = \lambda u.b \circ u \circ a; \tag{4.8}$$

$$a \otimes b = \lambda u.\langle a(u_0), b(u_1)\rangle; \tag{4.9}$$

$$a \oplus b = \lambda u.u_0 \sqsupseteq \langle 0, a(u_1)\rangle, \langle 1, b(u_1)\rangle. \tag{4.10}$$

These equations clearly generalize (4.1)–(4.3). Before we explain our operators, note these three equations which hold for *arbitrary* $a, b, a', b' \in \mathbf{P}\omega$:

$$(a \multimap b) \circ (a' \multimap b') = (a' \circ a) \multimap (b \circ b'); \tag{4.11}$$

$$(a \otimes b) \circ (a' \otimes b') = (a \circ a') \otimes (b \circ b'); \tag{4.12}$$

$$(a \oplus b) \circ (a' \oplus b') = (a \circ a') \oplus (b \circ b'). \tag{4.13}$$

The reversal of order $(a' \circ a)$ on the right-hand side of (4.11) should be remarked.

These equations will be used not only for properties of types (ranges of retracts) but also for the mappings between the types.

THEOREM 4.3 (The function space theorem). *Suppose a, b, a', b', c are retracts. Then we have*:

(i) $a \multimap b$ *is a retract, and it is strict if* b *is*;
(ii) $v : a \multimap b$ *iff* $u = \lambda x : a.u(x)$ *and* $\forall x : a.u(x) : b$;
(iii) *if* $a \trianglelefteq a'$ *and* $b \trianglelefteq b'$, *then* $a \multimap b \trianglelefteq a' \multimap b'$;
(iv) *if* $f : a \multimap b$ *and* $f' : a' \multimap b'$, *then* $f \multimap f' : (b \multimap a') \multimap (a \multimap b')$;
(v) *if* $f : a \multimap b$ *and* $f' : b \multimap c$, *then* $f' \circ f : a \multimap c$.

Parts (i), (iii), (iv), and (v) can be proved using (4.8) and (4.11) in an algebraic (formal) fashion. It is (ii) that tells us what it all means: the range of $a \multimap b$ consists exactly of those functions which are restricted to (the range of) a and which have values in b. So we can read $u : a \multimap b$ in the normal way: u is a (continuous) mapping from a into b. In technical jargon, we can say that the (strict) retracts and continuous functions form a *category*. In fact, it is equivalent to the category of separable continuous lattices and continuous maps. In this context, (iv) shows that $\multimap$ operates not only on spaces (retracts) but also on maps: it is a *functor* contravariant in the first argument and covariant in the second. Further categorical properties will emerge.

THEOREM 4.4 (The product theorem). *Suppose a, b, a', b' are retracts. Then we have*:

(i) $a \otimes b$ *is a retract, and it is strict if* a *and* b *are*;
(ii) $u : a \otimes b$ *iff* $u = \langle u_0, u_1 \rangle$ *and* $u_0 : a$ *and* $u_1 : b$;
(iii) *if* $a \trianglelefteq a'$ *and* $b \trianglelefteq b'$, *then* $a \otimes b \trianglelefteq a' \otimes b'$;
(iv) *if* $f : a \multimap b$ *and* $f' : a' \multimap b'$, *then* $f \otimes f' : a \otimes a' \multimap b \otimes b'$.

Again the operator proves to be a functor, but what is stated in Theorem 4.4 is not quite enough for the standard identification of $\otimes$ as the categorical product. For this we need some additional combinators:

$$\mathbf{fst} = \lambda u.u_0; \tag{4.14}$$

$$\mathbf{snd} = \lambda u.u_1; \tag{4.15}$$

$$\mathbf{diag} = \lambda u.\langle u, u \rangle. \tag{4.16}$$

Then we have these properties:

$$\mathbf{fst} \circ (a \otimes b) : (a \otimes b) \multimap a; \tag{4.17}$$

$$\mathbf{snd} \circ (a \otimes b) : (a \otimes b) \multimap b; \tag{4.18}$$

$$\mathbf{diag} \circ a : a \multimap a \otimes a; \tag{4.19}$$

$$\mathbf{fst} \circ (f \otimes f') = f \circ \mathbf{fst}; \tag{4.20}$$

$$\mathbf{snd} \circ (f \otimes f') = f' \circ \mathbf{snd}. \tag{4.21}$$

Here a and b are retracts and f and f' are functions. Now suppose a, b, and c are retracts and $f : c \multimap a$ and $g : c \to b$. Let

$$h = (f \otimes g) \circ \mathbf{diag} \circ c.$$

We can readily prove that:

$$h : c \multimap (a \otimes b),$$

and

$$\mathbf{fst} \circ h = f \quad \text{and} \quad \mathbf{snd} \circ h = g.$$

Furthermore, h is the *unique* such function. It is this uniqueness and existence property of functions into $a \otimes b$ that identifies the construct as a product.

There are important connections between $\multimap$ and $\otimes$. To state these we require some additional combinators:

$$\mathbf{eval} = \lambda u.u_0(u_1); \tag{4.22}$$

$$\mathbf{curry} = \lambda u \lambda x \lambda y.u(\langle x, y\rangle). \tag{4.23}$$

If a, b, and c are retracts, the mapping properties are:

$$\mathbf{eval} \circ ((b \multimap c) \otimes b) : ((b \multimap c) \otimes b) \multimap c \tag{4.24}$$

$$\mathbf{curry} \circ ((a \otimes b) \multimap c) : ((a \otimes b) \multimap c) \multimap (a \multimap (b \multimap c)). \tag{4.25}$$

Suppose next that $f : (a \otimes b) \multimap c$ and $g : a \multimap (b \multimap c)$. We find that

$$\mathbf{eval} \circ (\mathbf{curry}(f) \otimes b) = f$$

and

$$\mathbf{curry}(\mathbf{eval} \circ (g \otimes b)) = g.$$

This shows that our category of retracts is a *Cartesian closed category*, which means roughly that product spaces and function spaces within the category interact harmoniously.

THEOREM 4.5 (The sum theorem). *Suppose a, b, a', b' are retracts. Then we have*

(i) *$a \oplus b$ is a retract, and it is always strict*;

(ii) *$u : a \oplus b$ iff $u = \bot$ or $u = \top$ or*

$$u = \langle 0, u_1\rangle \text{ and } u_1 : a \text{ or}$$

$$u = \langle 1, u_1\rangle \text{ and } u_1 : b;$$

(iii) *if $a \Subset a'$ and $b \Subset b'$, then $a \oplus b \Subset a' \oplus b'$*;

(iv) *if $f : a \multimap b$ and $f' : a' \multimap b'$, then $f \oplus f' : a \oplus a' \multimap b \oplus b'$.*

There are several combinators associated with $\oplus$:

$$\mathbf{inleft} = \lambda x.\langle 0, x\rangle; \tag{4.26}$$

$$\mathbf{inright} = \lambda x.\langle 1, x\rangle; \tag{4.27}$$

$$\mathbf{outleft} = \lambda u.u_0 \supset u_1, \bot; \tag{4.28}$$

$$\mathbf{outright} = \lambda u.u_0 \supset \bot, u_1; \tag{4.29}$$

$$\mathbf{which} = \lambda u.u_0; \tag{4.30}$$

$$\mathbf{out} = \lambda u.u_1. \tag{4.31}$$

(The last two are the same as **fst** and **snd**, but they will be used differently.) We find:

$$(a \oplus b) \circ \mathbf{inleft} \circ a : a \rightarrowtail (a \oplus b); \tag{4.32}$$

$$(a \oplus b) \circ \mathbf{inright} \circ b : b \rightarrowtail (a \oplus b); \tag{4.33}$$

$$a \circ \mathbf{outleft} \circ (a \oplus b) : (a \oplus b) \rightarrowtail a; \tag{4.34}$$

$$b \circ \mathbf{outright} \circ (a \oplus b) : (a \oplus b) \rightarrowtail b; \tag{4.35}$$

$$\mathbf{which} \circ (a \oplus b) : (a \oplus b) \rightarrowtail \mathbf{bool}; \tag{4.36}$$

$$a \circ \mathbf{out} \circ (a \oplus a) : (a \oplus a) \rightarrowtail a; \tag{4.37}$$

where a and b are retracts. Most of these facts as they stand are trivial until one sets down the relations between all these maps; but there are too many to put them down here. Note, however, if a, b, and c are retracts and $f : a \rightarrowtail c$ and $g : b \rightarrowtail c$, then if we let

$$h = c \circ \mathbf{out} \circ (f \oplus g),$$

we have:

$$h : (a \oplus b) \rightarrowtail c,$$

and

$$h \circ \mathbf{inleft} = f \quad \text{and} \quad h \circ \mathbf{inright} = g.$$

But, though h exists, it is *not* unique. So $a \oplus b$ is not the categorical sum (coproduct). The author does not know a neat categorical characterization of this operator.

There would be no difficulty in extending $\otimes$ and $\oplus$ to more factors by expanding the range of indices from 0, 1 to $0, 1, \cdots, n-1$. The explicit formulae need not be given; but if we write $a_0 \otimes a_1 \otimes \cdots \otimes a_{n-1}$, we intend this expanded meaning rather than the iterated binary product.

To understand sums and other facts about retracts, consider the least fixed point of this equation:

$$\mathbf{tree} = \mathbf{nil} \oplus (\mathbf{tree} \otimes \mathbf{tree}). \tag{4.38}$$

To be certain that **tree** *is* a retract, we need a general theorem:

THEOREM 4.6 (The limit theorem). *Suppose F is a continuous function that maps retracts to retracts and let $c = \mathbf{Y}(F)$. Then c is also a retract. If in addition F maps strict retracts to strict retracts and is monotone in the sense that $a \Subset b$ implies $F(a) \Subset F(b)$ for all (strict) retracts a and b, then the range of c is homeomorphic to the inverse limit of the ranges of the strict retracts $F^n(\perp)$ for $n \in \omega$.*

This can be applied in the case of (4.38) where $F = \lambda z.\mathbf{nil} \oplus (z \otimes z)$. Thus we can analyze **tree** as an inverse limit. This approach has the great advantage over the earlier method of the author where limits were required in showing that **tree** *exists*. Here we use **Y** to give existence at once, and then apply Theorems 4.3–4.5 to figure out the nature of the retract.

In Theorem 4.6, the fact that c is a retract can be reasoned as follows: $\perp$ is a

retract. Thus each $F^n(\bot)$ is a retract. We compute:

$$\begin{aligned} c \circ c &= \bigcup\{F^n(\bot) \mid n \in \omega\} \circ \bigcup\{F^n(\bot) \mid n \in \omega\} \\ &= \bigcup\{F^n(\bot) \circ F^n(\bot) \mid n \in \omega\} \qquad (\textit{Note}\text{: same } n.) \\ &= \bigcup\{F^n(\bot) \mid n \in \omega\} = c. \end{aligned}$$

In case F is monotone and preserves strictness, then we can argue that each $F^n(\bot) \ll c$. The retracts $F^n(\bot)$ are the *projections* of c onto the terms of the limit. Of course $F^n(\bot) \ll F^m(\bot)$ if $n \leqq m$. The $u : c$ can be put into a one-to-one correspondence (homeomorphism, lattice isomorphism) with the infinite sequences $\langle v_0, v_1, \cdots, v_n, \cdots \rangle$, where $v_n : F^n(\bot)$ and $v_n = F^n(\bot)(v_{n+1})$. Indeed $v_n = F^n(\bot)(u)$ and $u = \bigcup\{v_n \mid n \in \omega\}$. This is exactly the inverse limit construction.

Retreating from generalities back to the example of **tree**, we can grant that it exists and is provably a retract. Two things in its range are $\bot$ and $\top$ by Theorem 4.5(ii), but they are not so interesting. Now $\bot$: **nil**, so by Theorem 4.5(ii) we have $\langle 0 \rangle = \langle 0, \bot \rangle$: **tree**. Let us think of this as *the* atom. What else can we have? If x, y : **tree**, then $\langle x, y \rangle$: **tree** $\otimes$ **tree** and so $\langle 1, \langle x, y \rangle \rangle$: **tree**. Thus (the range of) **tree** contains an atom and is closed under a binary operation. Note that the atomic and nonatomic trees are distinguished by **which** and that suitable constructor and destructor functions are definable on **tree**. But the space also contains infinite trees since we can solve for the least fixed point of:

$$t = \langle 1, \langle\langle 0 \rangle, t \rangle\rangle$$

and t : **tree**. (Why?) And there are many other examples of infinite elements in **tree**.

A point to stress in this construction is that **tree** being LAMBDA-definable is *computable*, and there are many computable functions definable on or to (the range of) **tree**. All the "structural" functions, for example, are computable. These are functions which in other languages would be called **isatom** or **construct** or **node**, and they are all easily LAMBDA-definable. Just as with $\oplus$, $\otimes$, $\multimap$, they are not explicit in the *notation*, but they are definable nevertheless. In the case of **node**, we could use finite sequences of Boolean values to pick out or name nodes. Thus solve for **name** = **nil** $\oplus$ **bool** $\otimes$ **name**, and then give a recursive definition of:

$$\textbf{node} : \textbf{name} \multimap (\textbf{tree} \multimap \textbf{tree}).$$

Any combination of retract preserving functors can be used in this game. For example:

$$\textbf{lamb} = \textbf{int} \oplus (\textbf{lamb} \multimap \textbf{lamb}). \tag{4.39}$$

This looks innocent, but the range of **lamb** would give a quite different and not unattractive model for the λ-calculus (plus arithmetic). What we do to investigate this model is to modify LAMBDA slightly by replacing the ternary conditional $z \supset x, y$ by a *quarternary* one $w \supset x, y, z$; otherwise the syntax of the language remains the same. The semantics, however, is a little more complex.

Let us use τ, σ, ρ, θ as syntactical variables for expressions in the modified language. The semantics is provided by a function $\mathscr{H}$ that maps the expressions of the language to their values in (the range of) **lamb**. To be completely rigorous we

also have to confront the question of free and bound variables. For simplicity let us index the variables of the language by integers, and let us take the variables to be $v_0, v_1, v_2, \cdots, v_n, \cdots$. We cannot simply evaluate out an expression τ to its value $\mathscr{H}[\![\tau]\!]$ until we know the values of the free variables in τ. The values of these variables will be given by an "environment" t which can be construed as a *sequence* of values in **lamb**. We can restrict these environments to the retract:

$$\textbf{env} = \lambda t.\textbf{lamb} \circ \textbf{seq}(t). \tag{4.40}$$

When $t : \textbf{env}$, then $t_n : \textbf{lamb}$ is the value that the environment gives to the variable v_n. We also need to employ a transformation on environments as follows:

$$t[x/n] = \lambda m \in \omega.\textbf{eq}(n)(m) \supset x, t_m. \tag{4.41}$$

Here **eq** is the primitive recursive function that is 0, if n, m are equal, and is 1, otherwise, for $n, m \in \omega$. The effect of $t[x/n]$ is to replace the nth term of the sequence t by the value of x, otherwise to leave the rest of the sequence unchanged. To correspond with our use of very simple variables we have selected a simple notion of environment: in the semantics of more general languages it is customary to regard an environment as a function from the set of variables into the domain of denotable values.

The correct way to evaluate a term τ given an environment t is to find $\mathscr{H}[\![\tau]\!](t)$. We use the brackets $[\![$ and $]\!]$ here simply as an aid to the eye in keeping the syntactical part separated from the rest. The environment enters as a function-argument in the usual way; thus we shall have:

$$\mathscr{H}[\![\tau]\!] : \textbf{env} \multimap \textbf{lamb}. \tag{4.42}$$

$$\begin{aligned}
&\mathscr{H}[\![v_n]\!](t) = t_n \\
&\mathscr{H}[\![0]\!](t) = \textbf{inleft}(0) \\
&\mathscr{H}[\![\tau + 1]\!](t) = \textbf{which}(\mathscr{H}[\![\tau]\!](t)) \supset \textbf{inleft}(\textbf{out}(\mathscr{H}[\![\tau]\!](t)) + 1), \perp \\
&\mathscr{H}[\![\tau - 1]\!](t) = \textbf{which}(\mathscr{H}[\![\tau]\!](t)) \supset \textbf{inleft}(\textbf{out}(\mathscr{H}[\![\tau]\!](t)) - 1), \perp \\
&\mathscr{H}[\![\theta \supset \tau, \sigma, \rho]\!](t) = \textbf{lamb}(\textbf{which}(\mathscr{H}[\![\theta]\!](t)) \supset \\
&\qquad\qquad (\textbf{out}(\mathscr{H}[\![\theta]\!](t)) \supset \mathscr{H}[\![\tau]\!](t), \mathscr{H}[\![\sigma]\!](t)), \mathscr{H}[\![\rho]\!](t)) \\
&\mathscr{H}[\![\tau(\sigma)]\!](t) = \textbf{which}(\mathscr{H}[\![\tau]\!](t)) \supset \perp, \textbf{out}(\mathscr{H}[\![\tau]\!](t))(\mathscr{H}[\![\sigma]\!](t)) \\
&\mathscr{H}[\![\lambda v_n.\tau]\!](t) = \textbf{inright}(\lambda x : \textbf{lamb}.\mathscr{H}[\![\tau]\!](t[x/n])).
\end{aligned} \tag{4.43}$$

A good question is: why does $\mathscr{H}$ exist? The answer is: because of the fixed-point theorem.

If we rewrite the semantic equations $\mathscr{H}[\![\tau]\!](t) = (\cdots)$ in (43) by the equation $\mathscr{H}[\![\tau]\!] = \lambda t : \textbf{env}(\cdots)$, then $\mathscr{H}$ is seen to be a function from expressions to values in **lamb**. As the range of **lamb** is contained in $\mathbf{P}\omega$, we can say more broadly that $\mathscr{H} \in \mathbf{P}\omega^{\mathbf{Exp}}$, where **Exp** is the syntactical set of expressions and the exponential notation designates the set of *all* functions from **Exp** into $\mathbf{P}\omega$. This function set is a complete lattice because $\mathbf{P}\omega$ is. Therefore if we read (4.43) as a definition by cases on **Exp**, then we can find $\mathscr{H}$ as a suitable fixed point in the complete lattice $\mathbf{P}\omega^{\mathbf{Exp}}$. Indeed it is the fixed point of a continuous operator.

Actually we can regard **Exp** as being a *subset* of $\mathbf{P}\omega$ to avoid dragging in other lattices. What we need is another recursive definition of a data type:

$$\begin{aligned}\mathbf{exp} = \mathbf{int} \oplus \mathbf{nil} \oplus \mathbf{exp} \oplus \mathbf{exp} \oplus (\mathbf{exp} \otimes \mathbf{exp} \otimes \mathbf{exp} \otimes \mathbf{exp}) \\ \oplus (\mathbf{exp} \otimes \mathbf{exp}) \oplus (\mathbf{int} \otimes \mathbf{exp})\end{aligned} \tag{4.44}$$

Note that there are as many summands in (4.44) as there are clauses in (4.43). We can think of **exp** as giving the "abstract" syntax of the language. We use the integers to index the variables and the nil element to stand for the individual constant. Read (4.44) as saying that every expression is *either* a variable *or* a constant *or* the successor of an expression *or* the predecessor of an expression *or* the conditional formed from a tuple of expressions *or* the abstraction formed from a pair of a variable and an expression. We do not need in (4.44) to introduce special "symbols" for the successor, application, etc., because the separation by cases given by the $\oplus$ operation is sufficient to make the distinctions. (That is why the syntax is "abstract".) The point is that for recursive definitions it does not matter how we make the distinctions as long as they can be made. From this new point of view, we could rewrite (4.43) so as to show:

$$\mathscr{H} : \mathbf{exp} \multimap (\mathbf{env} \multimap \mathbf{lamb}), \tag{4.45}$$

which is clearly more satisfactory—especially as it is now clear that $\mathscr{H}$ is *computable.* And this is a method that can be generalized to many other languages. The method also shows why it is useful to allow *function spaces* as particular data types.

Another example of this method can be illustrated, if the reader will recall the Gödel numbering of § 3. It will be seen that there are similarities with the **tree** construction: instead of 0 and $\mathbf{apply}(n)(m)$, **tree** uses $\langle 0 \rangle$ and $\langle 1, \langle x, y \rangle \rangle$. Note, however, that Gödel numbers are *finite* while **tree** has *infinite* objects. But the infinite objects are always *limits* of finite objects, so there *are* connections. (We discuss this again in § 6.) In particular, recursive definitions on Gödel numbers, like that of **val**, have analogues on **tree**. Here is the companion of (3.7).

$$\mathbf{vaal} = \lambda x : \mathbf{tree}.\mathbf{which}(x) \supseteq \mathbf{G}, \mathbf{vaal}(\mathbf{fst}(\mathbf{out}(x)))(\mathbf{vaal}(\mathbf{snd}(\mathbf{out}(x)))). \tag{4.46}$$

We have $\mathbf{vaal} : \mathbf{tree} \multimap \mathbf{id}$, where of course (4.46) is taken as defining **vaal** as the least fixed point. This is an example of a computable function between effectively given retracts. The LAMBDA-definable elements of $\mathbf{P}\omega$ are the computable elements in the range of **vaal**.

We have discussed the category of retracts and continuous maps, but if they are all LAMBDA-definable, then they fall within the countable model **RE**. Thus there is *another* category of effectively given retracts and effectively given continuous maps. (Examples: **tree**, **id**, **vaal**, and all those retracts and maps generated by $\oplus$, $\otimes$, and $\multimap$.) This category seems to deserve the status of a generalized recursion theory; though this is not to say that as yet very much is known about it. In fact, the proper formulation may require an enriched category rather than a restricted one. Thus instead of confining attention to the computable retracts and computable maps, it might be better to use the full category with all maps and to single out the computable ones (also maybe the finite ones) by special predicates. In effect we have avoided any methodological decisions by working in

the universal space $\mathbf{P}\omega$ and by *defining* a notion when required—if possible with the aid of LAMBDA. This makes it possible to give all the necessary definitions and to prove the theorems without at first having to worry about axiomatic problems.

5. Closure operations and algebraic lattices. Given any family of (finitary) operations on a set (say, ω) there is a closure operation defined on the subsets of that set obtained by forming the *least subset* including the given elements and *closed* under the operations. Examples are very familiar from algebra: the subgroup generated by a set of elements, the subspace spanned by a set of vectors, the convex hull of a set of geometric points. We simplify matters here by restricting attention to closures operating on sets in $\mathbf{P}\omega$, but the idea is quite general. The main point about these "algebraic" closure operations—as distinguished from topological closure operations—is that they are *continuous*. Thus, in the case of subgroups, if an element belongs to the subgroup generated by some elements, then it also belongs to the subgroup generated by *finitely* many of them. In the context of $\mathbf{P}\omega$ we can state the characteristic condition very simply.

DEFINITION. An element $a \in \mathbf{P}\omega$ is called a *closure operation* iff it satisfies: $\mathbf{I} \subseteq a = a \circ a$.

We see by definition that a closure operation is not only continuous, but it is also a retract. This is reasonable since the closure of the closure of a subset must be equal to the closure. To say of a function that $\mathbf{I} \subseteq a$, means that $x \subseteq a(x)$ for all $x \in \mathbf{P}\omega$. In other words, every set is contained in its closure. (Note that closures are opposite to the "projections", those retracts where $a \subseteq \mathbf{I}$.) Among examples of closure operations we find $\mathbf{I}$ and $\top$; the first has the most closed sets (fixed points), the second has the least. (Note that $\top = \omega$ always is a fixed point of a closure operation; $\top = \lambda x.\top$ is thus the most trivial closure operation.) The examples **fun**, **open**, **int** of § 4 are all closure operations (cf. (4.1), (4.6), (4.7)). We remarked that **fun** is a retract, but the reader should prove in addition:

$$u \subseteq \lambda x.u(x), \tag{5.1}$$

for all $u \in \mathbf{P}\omega$ (cf. Theorem 1.2). We note that this fact can be rewritten in the language of retracts as:

$$\mathbf{I} \subseteq \mathbf{I} \multimap \mathbf{I}, \tag{5.2}$$

the significance of which will emerge after we develop a bit of the theory of closure operations.

Unfortunately the natural definition of the retract **bool** does not yield a closure operation. In this section we adopt this modification:

$$\mathbf{boool} = \lambda u.u \supseteq 0, \top + 1. \tag{5.3}$$

The closed sets of **boool** are $\bot$, 0, $\top + 1$, and $\top$. Note that with any closure operation a, the function value $a(x)$ is the least closed set (fixed point of a) including as a subset the given set x. Thus given any family $\mathscr{C} \subseteq \mathbf{P}\omega$ of "closed" sets which is closed under the intersection of subfamilies, if we define

$$a(x) = \bigcap\{y \in \mathscr{C} | x \subseteq y\}, \tag{5.4}$$

then this will be a closure operation *provided* it is continuous. This remark makes it easy to check that certain functions are closure operations if we can spot easily the family $\mathscr{C}$ of fixed points.

Alas, the "natural" definition of ordered pairs (cf. (2.21)) leads to projections rather than closures. Here we must choose another:

$$[x, y] = \{2n | n \in x\} \cup \{2m+1 | m \in y\}, \tag{5.5}$$

with these inverse functions:

$$[u]_0 = \{n | 2n \in u\}, \tag{5.6}$$

$$[u]_1 = \{m | 2m+1 \in u\}. \tag{5.7}$$

We shall find that the main advantage of these equations lies in the obvious equation:

$$u = [[u]_0, [u]_1], \tag{5.8}$$

which is not true for the other pairing functions. Of course we have:

$$[[x, y]]_0 = x, \tag{5.9}$$

$$[[x, y]]_1 = y. \tag{5.10}$$

We shall not extend the idea of these new functions to triples and sequences, though it is clear what to do.

Abstractly, an *algebraic lattice* is a complete lattice in which the isolated points are dense. An isolated (sometimes called: *compact*) point in a lattice is one that is not the limit (sup or l.u.b.) of any directed family of its proper subelements. This definition works in continuous lattices, but more generally it is better to say that if the isolated point is *contained in* a sup, then it is also contained in a finite subsup (a sup of a finite selection of elements out of the given sup). In the case of the lattice of subgroups of a group, the isolated ones are the finitely generated subgroups. The isolated points of $\mathbf{P}\omega$ are the finite sets e_n. To say that isolated points are *dense* means that every element in the lattice is the sup of the isolated points it contains. The sequel to Theorem 4.1 for closure operations relates them to algebraic lattices.

THEOREM 5.1 (The algebraic lattice theorem). *The fixed points of any closure operation form an algebraic lattice.*

The proof is very easy if one notes that the isolated points of $\{x | x = a(x)\}$, where a is a closure operation, are exactly the images $a(e_n)$ of the finite sets in $\mathbf{P}\omega$. What makes Theorem 5.1 more interesting is the converse.

THEOREM 5.2 (The representation theorem for algebraic lattices). *Every algebraic lattice with a countable number of isolated points is isomorphic to the range of some closure operation.*

By Theorem 1.6 we know that the algebraic lattice is a retract, but a more direct argument makes the closure property clear. Thus, let D be the algebraic lattice with $\{d_n | n \in \omega\}$ as the set of all isolated points with the indicated enumeration. We shall use the square notation with symbols $\sqsubseteq$ and $\bigsqcup$ for the lattice ordering and sup. The desired closure operation is defined by:

$$a(x) = \{m | d_m \sqsubseteq \bigsqcup \{d_n | n \in x\}\}.$$

It is an easy exercise to show that from the definition of "isolated" it follows that a is continuous; and from density, it follows that D is in a one-to-one order preserving correspondence with the fixed points of a.

In the last section we introduced an algebra of retracts, much of which carries over to closure operations given the proper definitions. Without any change we can use Theorem 4.3 on function spaces, provided we check that the required retracts are closures.

THEOREM 5.3 (The function space theorem for algebraic lattices). *Suppose that a and b are closure operations; then so is* $a \multimap b$.

The proof comes down to showing that:

$$u(x) \subseteq b(u(a(x))), \tag{5.11}$$

whenever a and b are closure operations. But this is easy by monotonicity. Note that (5.1) is needed.

For those interested in topology, one can give a construction of the isolated points of the function space which is much more direct than just taking the functions $b \circ e_n \circ a$, which on the face of it do not tell us too much. But we shall not need this explicit construction here.

The reason for changing the pairing functions is to be able to form products and sums of closure operations. In the case of products, the analogue of $\otimes$ is straightforward:

$$a \boxtimes b = \lambda u.[a([u]_0), b([u]_1)]; \tag{5.12}$$

while for sums using $a' = \lambda x.0 \cup a(x-1)+1$ and similarly for b' we write:

$$a \boxplus b = \lambda u.([u_0]_0 \supset 0, 0) \cup ([u]_1 \supset 1, 1) \ni [a'([u]_0), \bot], [\bot, b'([u]_1)]. \tag{5.13}$$

We can then establish with the aid of (5.8)–(5.10):

THEOREM 5.4 (The product and sum theorem for algebraic lattices). *Suppose that a and b are closure operations; then so are* $a \boxtimes b$ *and* $a \boxplus b$. *Analogues of the results in Theorems* 4.4 *and* 4.5 *carry over.*

Following the discussion in § 4, we can also show that the closure operations form a Cartesian closed category, which in some ways is better than the category of *all* retracts. What makes it better is the existence of a "universe".

Every continuous operation generates a closure operation by just closing up the sets under the continuous function (as a set operation). We can institutionalize this thought by means of this definition:

$$\mathbf{V} = \lambda a \lambda x.\, \mathbf{Y}(\lambda y.x \cup a(y)). \tag{5.14}$$

Clearly $\mathbf{V}$ is LAMBDA-definable, continuous, etc. A more understandable characterization would define $\mathbf{V}(a)(x)$ by this equation:

$$\mathbf{V}(a)(x) = \bigcap\{y \mid x \subseteq y \text{ and } a(y) \subseteq y\}. \tag{5.15}$$

These two definitions are easily seen to be equivalent. What is unexpected is the discovery (due in a different form to Peter Hancock and Per Martin-Löf) that $\mathbf{V}$ itself is a closure operation.

THEOREM 5.5 (The universe theorem for algebraic lattices). *The function* $\mathbf{V}$ *is a closure operation and its fixed points comprise the set of all closure operations.*

Thus to say a is a closure operation, write $a : \mathbf{V}$. To have a mapping on closure operations, write $f : \mathbf{V} \to \mathbf{V}$. Remark that 5.5 allows us to write $\mathbf{V} : \mathbf{V}$. It all seems rather circular, but it is quite consistent. The category of separate algebraic lattices "contains itself"—if we are careful to work through retracts of $\mathbf{P}\omega$.

The proof of Theorem 5.5 requires a few steps. We note first that for all $x, a \in \mathbf{P}\omega$:

$$x \subseteq \mathbf{V}(a)(x). \tag{5.16}$$

Let $y = \mathbf{V}(a)(x)$. This is the least y with $x \cup a(y) \subseteq y$. What is the least z with $y \cup a(z) \subseteq z$? The answer is of course y, which proves:

$$\mathbf{V}(a)(\mathbf{V}(a)(x)) = \mathbf{V}(a)(x). \tag{5.17}$$

Thus $\mathbf{V}(a)$ is always a closure operation. If a is already a closure operation, then clearly $\mathbf{V}(a)(x) = a(x)$. Therefore we have shown:

$$a = \mathbf{V}(a) \quad \text{iff} \quad a \text{ is a closure operation.} \tag{5.18}$$

But then by (5.16) and (5.17) we have by (5.18):

$$\mathbf{V}(a) = \mathbf{V}(\mathbf{V}(a)). \tag{5.19}$$

From (5.16) by monotonicity we see:

$$a(x) \subseteq a(\mathbf{V}(a)(x)) \subseteq \mathbf{V}(a)(x). \tag{5.20}$$

Hence by (5.1) we can derive:

$$a \subseteq \lambda x.a(x) \subseteq \lambda x.\mathbf{V}(a)(x) = \mathbf{V}(a). \tag{5.21}$$

From (5.19) and (5.21) it follows that $\mathbf{V}$ itself is a closure operation.

The operation $\mathbf{V}$ forms the least closure operation containing a given element, and it shows that the lattice of closure operations is not only a retract of $\mathbf{P}\omega$ but also an algebraic lattice. Since we can now use $\mathbf{V}$ as a retract, the earlier results become formulas:

$$(\lambda a : \mathbf{V}.\lambda b : \mathbf{V}.a \boxtimes b) : \mathbf{V} \to (\mathbf{V} \to \mathbf{V}); \tag{5.22}$$

$$(\lambda a : \mathbf{V}.\lambda b : \mathbf{V}.a \boxplus b) : \mathbf{V} \to (\mathbf{V} \to \mathbf{V}); \tag{5.23}$$

we can also state such functorial properties as:

$$(\lambda a : \mathbf{V}.\lambda b : \mathbf{V}.a \to b) : \mathbf{V} \to (\mathbf{V} \to \mathbf{V}). \tag{5.24}$$

Using this style of notation we have:

THEOREM 5.6 (The limit theorem for algebraic lattices).

$$(\lambda f : \mathbf{V} \to \mathbf{V}.\mathbf{Y}(f)) : (\mathbf{V} \to \mathbf{V}) \to \mathbf{V}.$$

In words: if f is a mapping on closure operations, then its least fixed point is also a closure operation. The proof of course holds with *any* retract in place of $\mathbf{V}$, but we are more interested in applications to $\mathbf{V}$. For example, note that $\mathbf{V}(\bot) = \mathbf{I}$. Now let $f = \lambda a : \mathbf{V}.a \to a$. The least fixed point of this f is the limit of the sequence:

$$\bot,\ \mathbf{I},\ \mathbf{I} \to \mathbf{I},\ (\mathbf{I} \to \mathbf{I}) \to (\mathbf{I} \to \mathbf{I}),\ ((\mathbf{I} \to \mathbf{I}) \to (\mathbf{I} \to \mathbf{I})) \to ((\mathbf{I} \to \mathbf{I}) \to (\mathbf{I} \to \mathbf{I})), \cdots,$$

and we see that all these retracts are *strict.* This means $\mathbf{Y}(f)$ is nontrivial in that it has at least *two* fixed points (viz., $\bot$ and $\top$). But $d = \mathbf{Y}(f)$ must be the least closure operation satisfying

$$d = d \multimap d, \tag{5.25}$$

and we have thus proved that there are *nontrivial* algebraic lattices isomorphic to their own function spaces. This construction (which rests on hardly more than (5.2), since we could take $d = \mathbf{Y}(\lambda a.\mathbf{I} \cup (a \multimap a))$) is much quicker than the inverse limit construction originally found by the author to give λ-calculus models satisfying (η). There are many other fixed points of (5.25) besides this least closure operation, but their connection with inverse limits is not fully investigated.

We note in conclusion that most constructions by fixed points give algebraic lattices (like **lamb** in § 4), and so we could just as well do them in **V** if we remember to use $\boxtimes$ and $\boxplus$. The one-point space is $\top$ (*not* **nil**), and so the connection with inverse limits via Theorem 4.6 is not as clear when nonstrict functions are used. For many purposes, this may not make any difference.

6. Subsets and their classification. Retracts produce very special subsets of $\mathbf{P}\omega$: a retract always has a nonempty range which forms a lattice under $\subseteq$. For example, the range of **int** is $\{\bot, \top\} \cup \omega$. We often wish to eliminate $\bot$ and $\top$; and with a retract like **tree** the situation is more complex, since combinations like $\langle 1, \langle\langle 1, \langle \bot, \langle 0\rangle\rangle\rangle, \top\rangle\rangle$ might require elimination. In these two cases the method is simple.

Consider these two functions:

$$\mathbf{mid} = \lambda x : \mathbf{int}.\ x \supseteq 0, 0 \tag{6.1}$$

$$\mathbf{perf} = \lambda u : \mathbf{tree}.\mathbf{which}(u) \supseteq 0, \Delta(\mathbf{perf}(\mathbf{fst}(\mathbf{out}(u))))(\mathbf{perf}(\mathbf{snd}(\mathbf{out}(u)))) \tag{6.2}$$

where Δ is a special combinator:

$$\Delta = \lambda x \lambda y.(x \supset (y \supset 0, \top), \top) \cup (y \supset (x \supset 0, \top), \top). \tag{6.3}$$

We find that $\omega = \{x : \mathbf{int} | \mathbf{mid}(x) = 0\}$. In the case of trees, note first this behavior of Δ:

Δ	$\bot$	0	$\top$
$\bot$	$\bot$	$\bot$	$\top$
0	$\bot$	0	$\top$
$\top$	$\top$	$\top$	$\top$

The question is: what subset is $\{u : \mathbf{tree} | \mathbf{perf}(u) = 0\}$?

Now **perf** is defined recursively. We can see that

$$\mathbf{perf}(\bot) = \bot, \quad \mathbf{perf}(\top) = \top, \quad \mathbf{perf}(\langle 0\rangle) = 0,$$

and

$$\mathbf{perf}(\langle 1, \langle x, y\rangle\rangle) = \Delta(\mathbf{perf}(x))(\mathbf{perf}(y))$$

when $x, y : \mathbf{tree}$. Every tree, aside from $\top$ or $\bot$, is either atomic or a pair of trees. The atomic tree is "perfect" (that is, $\mathbf{perf}(\langle 0\rangle) = 0$). A *finite* tree which does not

contain $\bot$ or $\top$ is perfect—as we can see inductively using the table above for Δ. An infinite tree is never perfect: either some branch ends in $\top$ and **perf** maps it to $\top$, or $\top$ is never reached and **perf** maps it to $\bot$. Thus the subset in question is then seen to be the set of *finite* trees generated from the atom by pairing. This is clearly a desirable subset, and it is sorted out by a function with a simple recursive definition. The general question is: what subsets can be characterized by equations? The answer can be given by reference to the *topology* of $\mathbf{P}\omega$.

DEFINITION. Let $\mathfrak{G}$ be the class of *open* subsets of $\mathbf{P}\omega$, and $\mathfrak{F}$ be the class of *closed* subsets. Further let $\mathfrak{B}$ be the class of all (finite) *Boolean combinations* of open sets.

We recall from § 1 that $U \in \mathfrak{G}$ just in case for all $x \in \mathbf{P}\omega$, we have $x \in U$ if and only if some finite subset of x is in U. The class of open sets contains $\varnothing$ and $\mathbf{P}\omega$ and is closed under finite intersection and arbitrary union; in fact, it can be generated by these two closure conditions from subsets of the special form $\{x \in \mathbf{P}\omega \mid n \in x\}$ for the various $n \in \omega$ An open set is always *monotonic* (whenever $x \in U$ and $x \subseteq y$, then $y \in U$), so that every nonempty $U \in \mathfrak{G}$ has $\top \in U$.

Another characterization of openness can be given by continuous functions. Suppose $U \in \mathfrak{G}$. Define $f : \mathbf{P}\omega \to \{\bot, \top\}$ so that

$$U = \{x \mid f(x) = \top\};$$

then f is continuous. Conversely, if such an f is continuous, then U is open. But if we do not assume the range of f is included in $\{\bot, \top\}$, this is not true. For the case of general functions we know that f is continuous if and only if $\{x \mid f(x) \in V\}$ is open for all open V. This defines continuity in terms of openness, but we can turn it the other way around:

THEOREM 6.1 (The $\mathfrak{G}$ theorem). *The open subsets of* $\mathbf{P}\omega$ *are exactly the sets of the form*:

$$\{x \mid f(x) \supseteq 0\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

We could have written $0 \in f(x)$ or the equation $f(x) \cap 0 = 0$ instead of $f(x) \supseteq 0$. Note that in case $f : \mathbf{P}\omega \to \{\bot, \top\}$, then $f(x) \supseteq 0$ is equivalent to $f(x) = \top$. Also any other integer could have been used in place of 0.

We can say that $\{x \mid 0 \in x\}$ is the *typical* open set, and that every other open set can be obtained as an inverse image of the typical set by a continuous function. We shall extend this pattern to other classes, especially looking for equations. In the case of openness an inequality could also be used, giving as the typical set $\{x \mid x \neq \bot\}$. But since closed sets are just the complements of open sets, this remark gives us:

THEOREM 6.2 (The $\mathfrak{F}$ theorem). *The closed subsets of* $\mathbf{P}\omega$ *are exactly the sets of the form*:

$$\{x \mid f(x) = \bot\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

Aside from $\{x \mid x = \bot\}$, we could have used $\{x \mid x \subseteq a\}$ as the typical closed set where $a \in \mathbf{P}\omega$ is any element whatsoever aside from $\top$. This $\top$ has, by the way, a

special character. We note:

$$\{\top\} = \bigcap\{\{x | n \in x\} | n \in \omega\}.$$

Thus $\{\top\}$ is a *countable intersection* of open sets, otherwise called a $\mathfrak{G}_\delta$-set. There are of course many other $\mathfrak{G}_\delta$-sets, but $\{\top\}$ is the typical one:

THEOREM 6.3 (The $\mathfrak{G}_\delta$ theorem). *The countable intersections of open subsets of* $\mathbf{P}\omega$ *are exactly the sets of the form*:

$$\{x | f(x) = \top\},$$

where $f: \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

It may not be obvious that every $\mathfrak{G}_\delta$-set has this form. Certainly, as we have remarked, every $\mathfrak{G}$-set has this form. Thus if W is a $\mathfrak{G}_\delta$, we have:

$$W = \bigcap\{U_n | n \in \omega\}$$

and further,

$$U_n = \{x | f_n(x) = \top\},$$

where the f_n are suitably chosen continuous functions. Define the function g by the equation:

$$g(x) = \{(n, m) | m \in f_n(x)\}.$$

Clearly g is continuous, and we have:

$$W = \{x | g(x) = \top\},$$

as desired.

We let $\mathfrak{F} \dot{\cap} \mathfrak{G}$ denote the class of all sets of the form $C \cap U$, where $C \in \mathfrak{F}$ and $U \in \mathfrak{G}$. Similarly for $\mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$. Now $\{x | x \subseteq 0\}$ is closed and $\{x | x \supseteq 0\}$ is open. Thus $\{0\} \in \mathfrak{F} \dot{\cap} \mathfrak{G}$. This set is typical.

THEOREM 6.4 (The $\mathfrak{F} \dot{\cap} \mathfrak{G}$ theorem). *The sets that are intersections of closed sets with open sets are exactly the sets of the form*:

$$\{x | f(x) = 0\},$$

where $f: \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous.*

Again it may not be obvious that every $\mathfrak{F} \dot{\cap} \mathfrak{G}$ set has this form. We can write:

$$C = \{x | f(x) = \bot\},$$

and

$$U = \{x | g(x) = 0\},$$

where $C \in \mathfrak{F}$ and $U \in \mathfrak{G}$ and the continuous f and g are suitably chosen. Define

$$h(x) = \{2n + 1 | n \in f(x)\} \cup \{2n | n \in g(x)\},$$

and remark that h is continuous. We have:

$$C \cap U = \{x | h(x) = 0\},$$

as desired.

It is easy to see that $\{e\} \in \mathfrak{F} \dot{\cap} \mathfrak{G}$ if e is finite, but in general $\{a\} \in \mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$. In

case a is infinite but not equal to $\top$ (say, $a = \{n | n > 0\} = \top + 1$), then $\{a\}$ is typical in its class.

THEOREM 6.5 (The $\mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$ theorem). *The sets that are intersections of closed sets with countable intersections of open sets are exactly the sets of the form*:

$$\{x | f(x) = a\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous and* a *is a fixed infinite set not equal to* $\top$.

Note that $(\mathfrak{F} \dot{\cap} \mathfrak{G})_\delta$ is the same class as $\mathfrak{F} \dot{\cap} \mathfrak{G}_\delta$, so we see by Theorem 6.4 that a good choice of a is $\lambda n \in \omega.0$.

There is no single subset of $\mathbf{P}\omega$ typical for $\mathfrak{B}$, which can be viewed as the *finite unions* of sets from the class $\mathfrak{F} \dot{\cap} \mathfrak{G}$.

THEOREM 6.6 (The $\mathfrak{B}$ theorem). *The sets that are Boolean combinations of open sets are exactly the sets of the form*:

$$\{x | f(x) \in \mathscr{E}\},$$

where $f : \mathbf{P}\omega \to \mathbf{P}\omega$ *is continuous and* $\mathscr{E}$ *is a finite set of finite elements of* $\mathbf{P}\omega$.

To see that every $\mathfrak{B}$ set has this form, suppose that

$$V = W_0 \cup W_1 \cup \cdots \cup W_{n-1},$$

where each $W_i \in \mathfrak{F} \dot{\cap} \mathfrak{G}$. We can write:

$$W_i = \{x | f_i(x) = 0\},$$

where $f_i : \mathbf{P}\omega \to \{\bot, 0, \top\}$ is continuous. Then define:

$$g(x) = \{2i + j | j \in f_i(x) \cap \{0, 1\}, i < n\},$$

and note that g is continuous. Let

$$\mathscr{E} = \{y \subseteq \{m | m < 2n\} | \exists i < n. 2i \in y, 2i + 1 \notin y\};$$

we have:

$$V = \{x | g(x) \in \mathscr{E}\}$$

as desired.

THEOREM 6.7 (The $\mathfrak{B}_\delta$ theorem). *The sets that are countable intersections of Boolean combinations of open sets are exactly the sets of the form*:

$$\{x | f(x) = g(x)\},$$

where $f, g : \mathbf{P}\omega \to \mathbf{P}\omega$ *are continuous.*

This is clearly the most interesting of these characterization theorems, because equations like $f(x) = g(x)$ turn up all the time and the collection is a very rich totality of subsets of $\mathbf{P}\omega$. It includes all the retracts, since they are of the form $\{x | x = a(x)\}$. And much more. That every such set in Theorem 6.7 is $\mathfrak{B}_\delta$ follows from these logical transformations:

$$\{x | f(x) = g(x)\} = \{x | \forall n \in \omega [n \in f(x) \leftrightarrow n \in g(x)]\}$$

$$= \bigcap_{n=0}^{\infty} (\{x | n \in f(x), n \in g(x)\} \cup \{x | n \notin f(x), n \notin g(x)\})$$

That puts the set in the class $(\mathfrak{F} \dot{\cup} \mathfrak{G})_\delta \subseteq \mathfrak{B}_\delta$.

On the other hand, we can see that $(\mathfrak{F} \dot{\cup} \mathfrak{G})_\sigma$ is exactly $\mathfrak{B}_\delta$. Because, in view of Theorem 6.6, $\mathfrak{B}_\sigma$, the class of *countable* unions of $\mathfrak{B}$-sets, is exactly $(\mathfrak{F} \dot{\cap} \mathfrak{G})_\sigma$. The remark we want to make then follows by taking complements.

Now let S be an arbitrary $\mathfrak{B}_\delta$-set. We can write:

$$S = \bigcap_{n=0}^{\infty} (\{x | f_n(x) = 0\} \cup \{x | g_n(x) = 0\}),$$

where $f_n, g_n : \mathbf{P}\omega \to \{\bot, 0, \top\}$ are continuous. Now let u, v be continuous functions which on $\{\bot, 0, \top\}$ realize these two tables:

u	$\bot$	0	$\top$
$\bot$	$\bot$	0	0
0	0	0	$0'$
$\top$	0	$0'$	$0'$

v	$\bot$	0	$\top$
$\bot$	0	0	$0'$
0	0	0	$0'$
$\top$	$0'$	$0'$	$\top$

where $0' = 0 \cup 1$. This is an exercise in many-valued logic, and we find for $x, y \in \{\bot, 0, \top\}$:

$$u(x)(y) = v(x)(y) \quad \text{iff} \quad x = 0 \text{ or } y = 0.$$

Thus define continuous functions f' and g' such

$$f' = \lambda x \lambda n \in \omega . u(f_n(x))(g_n(x)),$$

$$g' = \lambda x \lambda n \in \omega . v(f_n(x))(g_n(x)),$$

and we find:

$$S = \{x | f'(x) = g'(x)\}$$

as desired.

This is as far as we can go with *equations*. More complicated sets can be defined using *quantifiers*, for example the Σ^1_1 or *analytic* sets can be put in the form:

$$\{x | \exists y . f(x)(y) = g(x)(y)\},$$

and their complements, the Π^1_1 sets, in the form

$$\{x | \forall y \; \exists z . h(x)(y)(z) = 0\},$$

with continuous f, g, h. For the three classes we then have as "typical" sets those shown in Table 3.

It should be remarked that $\mathfrak{B}_\delta$ contains all the closed sets in the *Cantor space* topology on $\mathbf{P}\omega$ (that is, the topology obtained when it is regarded as the infinite product of *discrete* two-point spaces). Therefore the Σ^1_1 sets for the two topologies on $\mathbf{P}\omega$ are the *same*. Hence, since we know for Cantor space that $\Delta^1_1 = \Sigma^1_1 \cap \Pi^1_1$ is the class of *Borel sets*, we can conclude that the two topologies on $\mathbf{P}\omega$ have the *same* Borel sets. (That is, in both cases Δ^1_1 is the Boolean σ-algebra generated from the open sets.)

TABLE 3
Classes and typical sets

Classes	Typical sets
$\mathfrak{G}$	$\{x \mid 0 \in x\}$
$\mathfrak{F}$	$\{\bot\}$
$\mathfrak{G}_\delta$	$\{\top\}$
$\mathfrak{F} \dot\cap \mathfrak{G}$	$\{0\}$
$\mathfrak{F} \dot\cap \mathfrak{G}_\delta$	$\{\top + 1\}$
$\mathfrak{B}_\delta$	$\{u \mid u_0 = u_1\}$
Σ^1_1	$\{u \mid \exists y.u_0(y) = u_1(y)\}$
Π^1_1	$\{u \mid \forall y\, \exists z.u(y)(z) = 0\}$

Returning now to the example involving trees mentioned at the beginning of this section, we see that the set of perfect (finite) trees can be written in the form:

$$\{x \mid x = \mathbf{tree}(x), \mathbf{perf}(x) = 0\} = \{x \mid \langle x, \mathbf{perf}(x)\rangle = \langle \mathbf{tree}(x), 0\rangle\};$$

thus it is a $\mathfrak{B}_\delta$-set. (Note that $\mathfrak{B}_\delta$ are obviously closed under finite intersection by the ordered pair method just illustrated; that they are closed under finite union is a little messier to make explicit, but the essential idea is contained in the proof of Theorem 6.7.)

As another example, we might wish to allow infinite trees but not the strange tree $\top$. Consider the following function:

$$\mathbf{top} = \lambda u : \mathbf{tree}.\mathbf{which}(u) \subseteq \bot, \mathbf{top}(\mathbf{fst}(\mathbf{out}(u))) \cup \mathbf{top}(\mathbf{snd}(\mathbf{out}(u))). \tag{6.4}$$

We can show that $\mathbf{top} : \mathbf{P}\omega \to \{\bot, \top\}$. For a tree u the equation $\mathbf{top}(u) = \bot$ means that it *does not* contain $\top$, or as we might say: it is *topless*. The topless trees form a closed subset of the subspace of trees. (An interesting retract is the function $\lambda u.\mathbf{tree}(u) \cup \mathbf{top}(u)$ whose range consists exactly of the topless trees plus *one* exceptional tree $\top$.) Such a closed subset of (the range of) a retract is a kind of *semilattice*. (We shall not introduce a precise definition here.) Every directed subset has a limit (least upper bound) and every pair with an upper bound has a least upper bound. But generally least upper bounds do not have to exist within the semilattice. The type of domains that interest us *become* continuous lattices with the *addition* of a top element $\top$ larger than all the other elements. The elimination of $\top$ is done with a function like **top** of our example. This is convincing evidence to the author that an independent theory of semilattices is quite unnecessary: they can all be *derived* from lattices. The problem is simply to define the top-cutting operation, then restriction to the "topless" elements is indicated by an equation (like $\mathbf{top}(u) = \bot$). In this way all the constructions are kept within the control of a smooth-running theory based on LAMBDA. This point seems to be important if one wants to keep track of which functions are computable.

An aspect of the problem of classification treated in this section which has not been given close enough attention is the explicitly constructive way of verifying the closure properties of the classes. Consider the class $\mathfrak{B}_\delta$, for example. Let B be the typical set as shown in Table 3. Then whatever $f \in \mathbf{P}\omega$ we choose, the set

$$\{x \mid f(x) \in B\}$$

is a $\mathfrak{B}_\delta$-set and every such set has this form. Thus the f's *index* the elements of the class. Suppose $f, g \in \mathbf{P}\omega$. What we should look for are two LAMBDA-definable combinators such that **union**$(f)(g)$ and **inter**$(f)(g)$ give the functions that index the union and intersection of the sets determined by f and g. That is, we want:

$$\{x|\mathbf{union}(f)(g)(x) \in B\} = \{x|f(x) \in B\} \cup \{x|g(x) \in B\}.$$

It should be possible to extract the precise definition from the outline of the proofs given above, but in general this matter needs more investigation. There may very well be certain classes where such operations are not constructive, even though the classes are simply defined.

7. Total functions and functionality. There is an inevitable conflict between the concepts of *total* and *partial* functions: we desire the former, but it is the latter we usually get. Total functions are better because they are "well-defined" at all their arguments, but the rub is that there is no general way of deciding when a *definition* is going to be well-defined in all its uses. In analysis we have singularities, and in recursion theory we have endless, nonfinishing computations. In the present theory we have in effect evaded the question in two ways. First we have embraced the partial function as the norm. But secondly, and possibly confusingly, the multiple-valued functions are normal, total functions from $\mathbf{P}\omega$ into $\mathbf{P}\omega$. The point, of course, is that we are making a *model* of the partial functions in terms of ordinary mathematical functions. But note that the success of the model lies in *not* using arbitrary functions: it is only the continuous functions that correspond to the kind of partial functions we wanted to study. It would be a mistake to think of the variables in λ-calculus as ranging over arbitrary functions—and this mistake was made by both Church and Curry. The fixed-point operator **Y** shows that we must restrict attention to functions which do have fixed points. It is certainly the case that $\mathbf{P}\omega$ is not the only model for the λ-calculus, but it is a very satisfactory model and is rich enough to illustrate what can and what cannot be done with partial functions.

Whatever the pleasures of partial functions (and the multiple-valued ones, too), the desire for total functions remains. Take the integers. We are more interested in ω than $\omega \cup \{\bot, \top\}$. Since the multiple values $\bot$ and $\top$ are but two in number, it is easy to avoid them. The problem becomes tiresome in considering functions, however. The lattice represented by the retract **int** $\multimap$ **int** is much too large, in that there are as many nontotal functions in this domain as total ones. The aim of the present section is to introduce an interpretation of a theory of functionality in the model $\mathbf{P}\omega$ that provides a convenient way of restricting attention to the functions (or other objects) that are total in the desired sense. The theory of functionality is rather like proposals of Curry, but not quite the same for important reasons as we will see.

In the theory of retracts of §§ 5 and 6, the plan of "restricting attention" was the very simple one of restricting to a *subset*. It was made notationally simple as the subsets in question could be parameterized by continuous functions. The retraction mappings stand in for their ranges. Even better, certain continuous functions act on these retractions as space-forming functors (such as $\oplus$ and $\multimap$), which gives greater notational simplicity because one language is able to serve for several

tasks. When we pass to the theory of total functions, this same kind of simplicity is no longer possible owing to an increase in quantifier complexity in the necessary definitions. (This remark is made definite below.) Another point where there is some loss of simplicity concerns the representation of entities in $\mathbf{P}\omega$: subsets will no longer be enough, since we will need *quotients* of subsets. This is not a very startling point. Many constructions are affected in a natural way via equivalence classes. An equivalence relation makes you blind to certain distinctions. It may be easier also to remain a bit blind than to search for the most beautiful representative of an equivalence class: there may be nothing to choose between several candidates, and it can cost too much effort to attempt a choice. Thus our first agreement is that for many purposes a *kind* of object can be taken as a set of equivalence classes for an equivalence relation on a subset of $\mathbf{P}\omega$.

Because $\mathbf{P}\omega$ is closed under the pairing function $\lambda x\lambda y.\langle x, y\rangle$, we shall construe relations on subsets of $\mathbf{P}\omega$ *as* subsets of $\mathbf{P}\omega$ all of whose elements are ordered pairs. That is, a relation A satisfies this inclusion:

$$A \subseteq \{\langle x, y\rangle | x, y \in \mathbf{P}\omega\}. \tag{7.1}$$

DEFINITION. A (restricted) *equivalence relation* on $\mathbf{P}\omega$ is a symmetric and transitive relation on $\mathbf{P}\omega$.

Such relations are restricted because they are only reflexive on their domains—which are the same as their ranges—and these are the subsets with which the relations are concerned. We shall write $x\,A\,y$ for $\langle x, y\rangle \in A$ and $x : A$ for $x\,A\,x$. What we assume about these relations is the following:

$$x\,A\,y \quad \text{implies} \quad y\,A\,x. \tag{7.2}$$

$$x\,A\,y \quad \text{and} \quad y\,A\,z \quad \text{imply} \quad x\,A\,z. \tag{7.3}$$

In case a is a retract, we introduce an equivalence relation to correspond:

$$E_a = \{\langle x, x\rangle | x : a\}. \tag{7.4}$$

This is the identity relation restricted to the range of a. Such relations (for obvious reasons) and many others satisfy an additional *intersection property*:

$$x\,A\,y \quad \text{and} \quad x\,A\,z \quad \text{imply} \quad x\,A\,(y \cap z). \tag{7.5}$$

We shall not generally assume (7.5) in this short discussion, but it is often convenient.

Each equivalence relation represents a *space*: the space of all its equivalence classes. Such spaces form a category more extensive than the category of retracts studied above. The familiar functors can be extended to this larger category by these definitions:

$$A \to B = \{\langle \lambda x.u(x), \lambda x.v(x)\rangle | u(x)Bv(y) \text{ whenever } x\,A\,y\}, \tag{7.6}$$

$$A \times B = \{\langle\langle x, x'\rangle, \langle y, y'\rangle\rangle | x\,A\,y \text{ and } x'\,B\,y'\}, \tag{7.7}$$

$$A + B = \{\langle\langle 0, x\rangle, \langle 0, y\rangle\rangle | x\,A\,y\} \cup \{\langle\langle 1, x'\rangle, \langle 1, y'\rangle\rangle | x'\,B\,y'\}. \tag{7.8}$$

THEOREM 7.1 (The closure theorem). *If A and B are restricted equivalence relations, then so are $A \to B$, $A \times B$ and $A + B$. We find*:

(i) $f : A \to B$ *iff* $f = \lambda x.f(x)$ *and whenever* $x\, A\, y$, *then* $f(x)\, B f(y)$, *in particular*:

(ii) *if* $f : A \to B$ *and* $x : A$, *then* $f(x) : B$; *furthermore*,

(iii) $u : A \times B$ *iff* $u = \langle u_0, u_1 \rangle$ *and* $u_0 : A$ *and* $u_1 : B$;

(iv) $u : A + B$ *iff either* $u = \langle 0, u_1 \rangle$ *and* $u_1 : A$ *or* $u = \langle 1, u_1 \rangle$ *and* $u_1 : B$.

It follows easily from 7.1 that the restricted equivalence relations form a Cartesian closed category which—in distinction to the category of retracts—has disjoint sums (or *coproducts* as they are usually called in category theory). This result is probably a special case of a more general theorem. The point is that $\mathbf{P}\omega$ itself is a space in a Cartesian closed category (that of continuous lattices and continuous maps) and it contains as subspaces the Boolean space and especially its own function space and Cartesian square. In this circumstance any such rich space must be such that its restricted equivalence relations again form a good category. Our construction is not strictly categorical in nature, as we have used the *elements* of $\mathbf{P}\omega$ and have relied on being able to form arbitrary subsets (arbitrary relations). But a more abstract formulation must be possible. The connection with the category of retracts is indicated in the next theorem.

THEOREM 7.2 (The isomorphism theorem). *If a and b are retracts, we have the following isomorphisms and identities relating the spaces*:

(i) $E_a \cong \{\langle x, y \rangle | a(x) = a(y)\}$;

(ii) $E_{a \multimap b} \cong E_a \to E_b$;

(iii) $E_{a \otimes b} = E_a \times E_b$;

(iv) $E_{a \oplus b} = E_a + E_b \cup \{\langle \bot, \bot \rangle, \langle \top, \top \rangle\}$.

Part (iv) is not categorical in nature as it stands, but (ii) amd (iii) indicate that E is a functor from the category of retracts into the category of equivalence relations that shows that the former is a full sub-Cartesian-closed category of the latter. We cannot pursue the categorical questions here, but note that there are many subcategories that might be of interest; for example, the equivalence relations with the intersection property are closed under $\to$, $\times$, and $+$.

Returning to the question of total functions we introduce this notation:

$$N = \{\langle n, n \rangle | n \in \omega\}. \tag{7.9}$$

This is the type of the integers *without* and $\bot$ and $\top$, i.e., the total integers. We note that:

$$N = \{u | u = \langle u_0, u_0 \rangle,\ u_0 = \mathbf{int}(u_0),\ \mathbf{mid}(u_0) = 0\}. \tag{7.10}$$

Thus N is a $\mathfrak{B}_\delta$-set. What is $N \to N$? We see:

$$f : N \to N \quad \text{iff} \quad f : \mathbf{fun} \text{ and } f(n) \in \omega \text{ whenever } n \in \omega. \tag{7.11}$$

This $N \to N$ is indeed the type of *all* total functions from ω into ω. It can be shown that $N \to N$ is also a $\mathfrak{B}_\delta$-set: good. But what is $(N \to N) \to N$? This is no longer a $\mathfrak{B}_\delta$-set, the best we can say is Π^1_1. By Theorem 7.1 it corresponds to the type of all

(extensional) *continuous* total functions from $N \to N$ into N. (The condition on A and B on the right side of (7.6) makes the concept of function embodied in $A \to B$ extensional, since the functions are meant to preserve the equivalence relations.)

A more precise discussion identifies $N \to N$ as a topological space, usually called the *Baire space.* If we introduce the finite discrete spaces by:

$$N_k = \{\langle n, n\rangle | n < k\}, \tag{7.12}$$

then $N \to N_2$ can also be identified with a topological space, usually called the Cantor space. In this identification we find at the next type, say either $(N \to N) \to N$ or $(N \to N_2) \to N_2$, that elements correspond to the usual notion of continuous function defined in topological terms. However, these higher type spaces are not at all conveniently taken as *topological* spaces. Certain of them can be identified as *limit spaces* according to the work of Hyland, and for these $\to$, $\times$, and $+$ have the natural interpretation. We cannot enter into these details here, but we can remark that the higher type spaces become ever more complicated. Thus $((N \to N) \to N) \to N$ is a Π^1_2-set and each $\to$ will add another quantifier to the definition. This is reasonable, because to say that a function is total is to say that *all* its values are well-behaved. But if its domain is a complex space, this statement of totality is even more complex. Despite this complexity, however, it is possible to sort out what kind of mapping properties many functions have. We shall mention a few of the combinators.

THEOREM 7.3 (The functionality theorem). *The combinators* **I**, **K**, *and* **S** *enjoy the following functionality properties which hold for all equivalence relations* A, B, C:

(i) $\mathbf{I} : A \to A$;

(ii) $\mathbf{K} : A \to (B \to A)$;

(iii) $\mathbf{S} : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$.

Furthermore, these combinators are uniquely determined by these properties.

Let us check that S satisfies (iii). Suppose that:

$$f(A \to (B \to C))f'.$$

We must show that:

$$\mathbf{S}(f)((A \to B) \to (A \to C))\mathbf{S}(f').$$

To this end suppose that:

$$g(A \to B)g'.$$

We must show that:

$$\mathbf{S}(f)(g)\,(A \to C)\,\mathbf{S}(f')(g').$$

To this end suppose that:

$$x\,A\,x'.$$

We must show that:

$$\mathbf{S}(f)(g)(x)C\mathbf{S}(f')(g')(x').$$

Now by definition of the combinator **S** we have:

$$\mathbf{S}(f)(g)(x) = f(x)(g(x)),$$

$$\mathbf{S}(f')(g')(x') = f'(x')(g'(x')).$$

By assumptions on g, g' and on x, x', we know:

$$g(x)\ B\ g'(x').$$

By assumptions on f, f' and on x, x', we know:

$$f(x)(B \to C)f'(x').$$

The desired conclusion now follows when we note such combinations as $\mathbf{S}(f)$ and $\mathbf{S}(f)(g)$ are indeed functions. (We are using Theorem 7.1(i) several times in this case.)

In the case of the converse, let us suppose by way of example that $k \in \mathbf{P}\omega$ is such that

$$k : A \to (B \to A)$$

holds for all equivalence relations A and B. By specializing to, say, the identity relation we see that whatever $a \in \mathbf{P}\omega$ we take, both k and $k(a)$ are functions. To establish that $k = \mathbf{K}$ we need to show that the equation:

$$k(a)(b) = a$$

holds for all $a, b \in \mathbf{P}\omega$. This is easy to prove, for we have only to set:

$$A = \{\langle a, a\rangle\} \quad \text{and} \quad B = \{\langle b, b\rangle\},$$

and the equation follows at once. Not all proofs are quite so easy, however.

In the case of the combinator **S** it is not strictly true to say that Theorem 7.3(iii) determines it outright. The exact formulation is this: *if* $s \in \mathbf{P}\omega$ is such that:

$$s(f) = s(\lambda x \lambda y.f(x)(y)) \quad \text{and} \quad s(f)(g) = s(f)(\lambda x.g(x))$$

for all $f, g \in \mathbf{P}\omega$; and *if*

$$s : (A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

for all A, B, C, *then* $s = \mathbf{S}$. In other words, we need to know that s converts its first two arguments into functions with the right number of places before we can say that its explicit functionality identifies as being the combinator **S**.

In Hindley, Lercher and Seldin (1972) they show that the functionality property:

$$\lambda f \lambda g.f \circ g : (B \to C) \to ((A \to B) \to (A \to C)), \tag{7.13}$$

follows from Theorem 7.1(ii) and Theorem 7.3(ii) and (iii) in view of the identity:

$$\lambda f \lambda g.f \circ g = \mathbf{S}(\mathbf{K}(\mathbf{S}))(\mathbf{K}) \tag{7.14}$$

(see Appendix A).

A more interesting result concerns the *iterators* defined as follows:

$$\mathbf{Z}_0 = \lambda f \lambda x.x, \tag{7.15}$$

$$\mathbf{Z}_{n+1} = \lambda f \lambda x.f(\mathbf{Z}_n(f)(x)). \tag{7.16}$$

In other words, $\mathbf{Z}_n(f)(x) = f^n(x)$. These natural combinators can be typed very easily, but Gordon Plotkin has shown that the obvious typing actually characterizes them.

THEOREM 7.4 (The iterator theorem). *The combinators* $\mathbf{Z}_n$ *enjoy the following functionality property which holds for all equivalence relations* A:

(i) $$\mathbf{Z}_n : (A \to A) \to (A \to A).$$

Further, if any element $z \in \mathbf{P}\omega$ *satisfies* (i) *for all* A, *then it must be one of the iterators, provided that* $z(f) = z(\lambda x.f(x))$ *holds for all* $f \in \mathbf{P}\omega$.

That each of the $\mathbf{Z}_n$ satisfies Theorem 7.4(i) is obvious. Suppose z were another such element. Then clearly:

$$z = \lambda f \lambda x.z(f)(x).$$

Suppose f and x are fixed for the moment. Let:

$$A = \{\langle f^n(x), f^n(x)\rangle | n \in \omega\},$$

where we can suppose in addition that:

$$f = \lambda x.f(x).$$

Then $f : A \to A$ is clear, and so $z(f) : A \to A$ also. But $x : A$, therefore $z(f)(x) = f^n(x)$, for some $n \in \omega$, because $z(f)(x) : A$. The trouble with this easy part of the argument is that the integer n *depends* on f and x. What we must show is that it is independent of f and x, then $z = \mathbf{Z}_n$ will follow.

Plotkin's method for this case is to introduce some independent successor functions:

$$\sigma_j = \lambda x.\{(j, k+1) | (j, k) \in x\}. \tag{7.17}$$

Note that:

$$\sigma_j^m((j', 0)) = \begin{cases} (j, m) & \text{if } j = j'; \\ \bot & \text{if } j \neq j'. \end{cases} \tag{7.18}$$

It then follows that:

$$(\sigma_j \cup \sigma_{j'})^m((j, 0) \cup (j', 0)) = (j, m) \cup (j', m). \tag{7.19}$$

Having these identities, we return to the argument.

From what we saw before, given $j \in \omega$, there is an n_j such that:

$$z(\sigma_j)((j, 0)) = \sigma_j^{n_j}(j, 0) = (j, n_j).$$

Take any two $j, j' \in \omega$. We also know there is an $n \in \omega$ where:

$$z(\sigma_j \cup \sigma_{j'})((j, 0) \cup (j', 0)) = (j, n) \cup (j', n),$$

in view of (7.19). But since $\sigma_j \subseteq \sigma_j \cup \sigma_{j'}$ and $\sigma_{j'} \subseteq \sigma_j \cup \sigma_{j'}$, we have:

$$(j, n_j) \cup (j', n_{j'}) \subseteq (j, n) \cup (j', n).$$

It follows that

$$n_j = n = n_{j'}$$

and so they are all equal. This determines the fixed $n \in \omega$ we want.

Suppose that both f and x are finite sets in $\mathbf{P}\omega$. Choose $j > \max(f \cup x)$. Let A this time be the least equivalence relation such that:

$$f^m(x)\, A\, f^m(x) \cup (j, m)$$

holds for all $m \in \omega$. We then check that:

$$\lambda x.f(x)(A \to A)(\lambda x.f(x)) \cup \sigma_j.$$

Therefore, we have:

$$z(\lambda x.f(x))(A \to A) z((\lambda x.f(x)) \cup \sigma_j),$$

and since $x\, A\, x \cup (j, 0)$, we get:

$$z(\lambda x.f(x))(x)\, A\, z((\lambda x.f(x)) \cup \sigma_j)(x \cup (j, 0)).$$

Now there is an integer $m \in \omega$ such that:

$$\begin{aligned} z((\lambda x.f(x)) \cup \sigma_j)(x \cup (j, 0)) &= f^m(x) \cup \sigma_j^m((j, 0)) \\ &= f^m(x) \cup (j, m), \end{aligned}$$

where we have been able to separate f and σ because j is so large. But the right-hand side must contain $z(\sigma_j)(j, 0) = (j, n)$. Thus m and our fixed n are the same. The other element

$$z(\lambda x.f(x))(x) = f^q(x)$$

for some $q \in \omega$. Thus we have:

$$f^q(x)\, A\, f^n(x) \cup (j, n).$$

Again since j is so large, $(j, n) \notin f^q(x)$. Thus by our choice of A we must have $f^q(x) = f^n(x)$. This means then, since n is fixed, that for *all finite* f, x:

$$z(\lambda x.f(x))(x) = f^n(x).$$

But then by continuity this equation holds for *all* f, x. It follows now that $z = \mathbf{Z}_n$, by the proviso of the theorem.

These results bring up many questions which we leave unanswered here. For example, which combinators (i.e., pure λ-terms) have functionality as in the examples above, and can we decide when a term is one such? In particular, can the diagonal combinator $\lambda x.x(x)$ be typed? (The argument of Hindley, et al. (1972, p. 81) is purely formal and does not apparently apply to the model.) What about terms in LAMBDA beyond the pure λ-calculus?

Appendix A. Proofs and technical remarks.

For Section 1. If we give the two-point space $\{\bot, \top\}$ the weak T_0-topology with just three open sets: $\varnothing$, $\{\top\}$, $\{\bot, \top\}$, we have what is called the Sierpinski space and its infinite product $\{\bot, \top\}^\omega$ with the product topology is the same as $\mathbf{P}\omega$. The finite sets $e_n \in \mathbf{P}\omega$ correspond exactly to the usual basic open sets for the product. For those familiar with such notions, this well-known observation makes many of the facts mentioned in this section fairly obvious. From any point of view, Theorem 1.1 and the remarks in the following paragraph are simple exercises.

Proof of Theorem 1.2. Equation (i) as a functional equation comes down to

$$\{m | \exists e_n \subseteq x. m \in f(e_n)\} = f(x),$$

which is just another way of writing the definition of continuity. Thus it is indeed true for all x. Next, inclusion (ii) means that if $(n, m) \in u$, then $\exists e_k \subseteq e_n.(k, m) \in u$. Clearly all we need to do is take $k = n$. If we also want the converse inclusion to hold, then what we need is condition (iii).

Proof of Theorem 1.3. Substitution is generalized composition of functions of many variables with all possible identifications and permutations of the variables; however, as we are able to define continuity by separating the variables, the argument reduces to a few special cases. The first trick is to take advantage of monotonicity. Thus, suppose $f(x, y)$ is continuous in each of its variables. What can we say of $f(x, x)$, a very special case of substitution? We calculate

$$\begin{aligned} f(x, x) &= \bigcup\{f(e_n, x) | e_n \subseteq x\} \\ &= \bigcup\{f(e_n, e_m) | e_n \subseteq x, e_m \subseteq x\}. \end{aligned}$$

Then if we think of $e_k = e_n \cup e_m$ and realize that $f(e_n, e_m) \subseteq f(e_k, e_k)$, we see that

$$f(x, x) = \bigcup\{f(e_k, e_k) | e_k \subseteq x\}.$$

This means that $f(x, x)$ is continuous in x. This same argument works if other variables are present, as in the passage from $f(x, y, z, w)$ to $f(x, x, z, w)$. When an identification of more than two variables is required, as from $f(x, y, z, w)$ to $f(x, x, x, x)$, the principle is just applied several times.

Finally to show that $f(g(x, y), h(y, x, y))$ is continuous, it is sufficient to show that $f(g(x, y), h(z, u, v))$ is continuous in each of its variables *separately*. By simply overlooking the remaining variables, this comes down to showing that $f(g(x))$ is continuous if f and g are. But the proof for ordinary composition is very easy with the aid of the characterization theorem (Theorem 1.1).

Proof of Theorem 1.4. This well-known fact holds for continuous functions on many kinds of chain-complete partial orderings; but $\mathbf{P}\omega$ illustrates the idea well enough. Suppose f had a fixed point $x = f(x)$. Then since $\varnothing \subseteq x$ and f is monotonic, we see that $f(\varnothing) \subseteq f(x) = x$. But then again, $f(f(\varnothing)) \subseteq f(x) = x$; and so by induction, $f^n(\varnothing) \subseteq x$. This proves that $\mathbf{fix}(f) \subseteq x$; and thus if $\mathbf{fix}(f)$ is a fixed point, it must be the least one. To prove that it *is* a fixed point, we need a fact that will often be useful:

LEMMA. *If $x_n \subseteq x_{n+1}$ for all n, and if f is continuous, then*

$$f(\bigcup\{x_n | n \in \omega\}) = \bigcup\{f(x_n) | n \in \omega\}.$$

Proof. By monotonicity, the inclusion holds in one direction. Suppose $e_m \subseteq f(\bigcup\{x_n | n \in \omega\})$. Then by Theorem 1.1 we have $e_m \subseteq f(e_k)$ for some $e_k \subseteq \bigcup\{x_n | n \in \omega\}$. Because e_k is finite and the sequence is increasing, we can argue that $e_k \subseteq x_n$ for some n. But then $f(e_k) \subseteq f(x_n)$. This shows that $e_m \subseteq \bigcup\{f(x_n) | n \in \omega\}$ and proves the inclusion in the other direction. (*Exercise*: Does this property characterize continuous functions?)

Proof of Theorem 1.4 *concluded.* Noting that $f^n(\varnothing) \subseteq f^{n+1}(\varnothing)$ holds for all n, we can calculate:

$$f(\mathbf{fix}(f)) = \bigcup\{f(f^n(\varnothing)) | n \in \omega\} = \bigcup\{f^{n+1}(\varnothing) | n \in \omega\}.$$

But this is just $\mathbf{fix}(n)$, since the only term left out is $f^0(\varnothing) = \varnothing$.

Proof of Theorem 1.5. The function $\bar{f}$ is clearly well-defined even when $y \in Y$ has a neighborhood U where $X \cap U = \varnothing$: in that case $\bar{f}(y) = \omega$ by convention on the meaning of $\bigcap$ in $\mathbf{P}\omega$. In case $x \in X$, it is obvious that $\bar{f}(x) \subseteq f(x)$. For the opposite inclusion, suppose that $m \in f(x)$. Because f is continuous and $\{z | m \in z\}$ is open in $\mathbf{P}\omega$, there is an open subset V of X such that $x' \in V$ always implies that $m \in f(x')$. But X is a subspace of Y, so $V = X \cap U$ for some open subset U of Y. Thus we can see why $m \in \bar{f}(x)$. It remains to show that $\bar{f}$ is itself continuous.

We must show that the inverse image under $\bar{f}$ of every open subset of $\mathbf{P}\omega$ is open in Y. But the open subsets of $\mathbf{P}\omega$ are unions of finite intersections of sets of the form $\{z | m \in z\}$. Thus it is enough to show that $\{y | m \in \bar{f}(y)\}$ is always open in Y. But this set equals $\bigcup\{U | m \in \bigcap\{f(x) | x \in X \cap U\}\}$, which being a union of open sets is open. Note that what we have proved is that $\bar{f}$ is continuous no matter what function f is given; however, if f is not continuous, then $\bar{f}$ cannot be an extension of f.

For readers not as familiar with general topology we note that the idea of Theorem 1.5 can be turned into a *definition.* Suppose $X \subseteq \mathbf{P}\omega$ is a sub*set* of $\mathbf{P}\omega$. It becomes a sub*space* with the relative topology. What are the continuous functions $f: X \to \mathbf{P}\omega$? From Theorem 1.5 we see that a necessary and sufficient condition is that the $\bar{f}: \mathbf{P}\omega \to \mathbf{P}\omega$ *be* an extension of f. Thus for $x \in X$ we can write the equation $f(x) = \bar{f}(x)$ as a biconditional:

$$m \in f(x) \quad \text{iff} \quad \exists e_n \subseteq x \; \forall x' \in X[e_n \subseteq x' \text{ implies } m \in f(x')],$$

which is to hold for all $m \in \omega$. This form of the definition of continuity on a subspace is more complicated than the original definition, because in general $e_n \notin X$ and we cannot write $f(e_n)$.

Proof of Theorem 1.6. What T_0 means is that every point of X is uniquely determined by its neighborhoods. Now $\varepsilon(x)$ just tells you the set of indices of the (basic) neighborhoods of x. Thus it is clear that ε is one-to-one. To prove that it is continuous, we need only note:

$$\{x | n \in \varepsilon(x)\} = U_n,$$

which is always open. To show that ε is an embedding, we must finally check that the *images* of the open sets U_n are open in $\varepsilon(X)$. This comes down to showing:

$$\varepsilon(U_n) = \varepsilon(X) \cap \{z | n \in z\},$$

which is clear.

For Section 2. Equation (2.1) defines a continuous function because it is a special case of Theorem 1.5, where we have been able to simplify the definition into cases because ω is a very elementary subset of $\mathbf{P}\omega$. Equation (2.2) gives a continuous function since the definition makes $\hat{p}$ distributive, as remarked in the text for finite unions, but it is just as easy to show that $\hat{p}$ distributes over arbitrary unions. The difference between a continuous f and a distributive $\hat{p}$ is this: to find $m \in f(x)$ we need a finite subset $e_n \subseteq x$ with $m \in f(e_n)$; however, to find $m \in \hat{p}(x)$ we need only *one* element $n \in x$ with $m \in \hat{p}(\{n\}) = \hat{p}(n) = p(n)$. Continuous functions are generalizations of distributive functions. The generality is necessary. For example, in (2.3) we see another function x; y distributive in each of its variables; but take care: the function x; x is not distributive in x—there is no closure under substitution. This is just one reason why continuous functions are better. Another good example comes from (2.4) if you compare the functions x, $x+x$, $x+x+x$, etc.

Equations (2.5)–(2.7) are very elementary. Note that $z \supset x, y$ is distributive in each of its variables. We could write: $z \supset x, y = \hat{p}(z)$, where $p(0) = x$ and $p(n+1) = y$, to show that it is distributive in z.

Proof of Theorem 2.1. If we did not use the λ-notation, then all LAMBDA-definable functions would be obtained by substitution from the first five (cf. Table 2). Since they are all seen to be continuous, the result would then follow by the substitution theorem (Theorem 1.3). Bringing in λ-abstraction means that we have to combine Theorem 1.3 with this fact:

LEMMA. *If $f(x, y, z, \cdots)$ is a continuous function of all its variables, then $\lambda x.f(x, y, z, \cdots)$ is a continuous function of the remaining variables.*

Proof. It is enough to consider one extra variable. We compute from the definition of λ in Table 2 as follows:

$$\begin{aligned}\lambda x.f(x, y) &= \{(n, m) | m \in f(e_n, y)\} \\ &= \{(n, m) | \exists e_k \subseteq y.m \in f(e_n, e_k)\} \\ &= \bigcup\{\{(n, m) | m \in f(e_n, e_k)\} | e_k \subseteq y\} \\ &= \bigcup\{\lambda x.f(x, e_k) | e_k \subseteq y\}.\end{aligned}$$

Thus $\lambda x.f(x, y)$ is continuous in y.

Proof of Theorem 2.2. The reason behind this result is the restriction to continuous functions. Theorem 2.1 shows that we cannot violate the restriction by giving definitions in LAMBDA, and the graph theorem (Theorem 1.2) shows that continuous functions correspond perfectly with their graphs.

The verification of (α) of Table 1 is obvious as the 'x' in '$\lambda x.\tau$' is a bound variable. (Care should be taken in making the proviso that 'y' is not otherwise free in τ.) The same would of course hold for any other pair of variables. We do not bother very much about alphabetic questions.

The verification of (β) is just a restatement of Theorem 1.2(i). Let τ define a function f (of x). Then by definition $f(x) = \tau$ and $\lambda x.\tau = \mathbf{graph}(f)$. Also $\mathbf{fun}(u)(x)$ in the notation of Theorem 1.2 is the same as the binary operation $u(x)$ in the notation of LAMBDA. Thus in Theorem 1.2(i) if we apply both sides to y we get nothing else than (β).

Half of property (ξ) is already implied by (β): the implication from left to right. (Just apply both sides to x.) In the other direction, $\forall x.\tau = \sigma$ means that τ and σ define the *same* function of x; thus, the two graphs must be equal.

Remarks on other laws. The failure of (η) simply means that not every set in $\mathbf{P}\omega$ *is* the graph of a function. Condition (iii) of Theorem 1.2 is equivalent to saying that $u = \lambda x.u(x)$, in other words, u is the graph of some function if and only if it is the graph of the function determined by u.

Law (μ) is the monotone property of application (in both variables); therefore, (μ) and (ξ) together imply (ξ^*) from left to right. Suppose that $\forall x.\tau \subseteq \sigma$; then clearly:

$$\{(n, m) | m \in \tau[e_n/x]\} \subseteq \{(n, m) | m \in \sigma[e_n/x]\},$$

which gives (ξ^*) from right to left.

There are, by the way, other laws valid in the model, as explained in the later results.

Proof of Theorem 2.3. This is a standard result combinatory logic. We have only to put:

$$u = \lambda x_0 \lambda x_1 \cdots \lambda x_{n-1}.f(x_0, x_1, \cdots, x_{n-1}).$$

That is, we use the iteration of the process of forming the graph of a continuous function. As each step (from the inside out) keeps everything continuous, we are sure that the equation of Theorem 2.3 will hold for iterated application.

Proof of Theorem 2.4. This can be found in almost any reference on combinatory logic or λ-conversion. The main idea is to eliminate the λ in favor of the combinators. The fact that we have a few other kinds of terms causes no problem if we introduce the corresponding combinators. The method of proof is to show, for any LAMBDA-term τ with free variables among $x_0, x_1, \cdots, x_{n-1}$, that there is a combination γ of combinators such that:

$$\tau = \gamma(x_0)(x_1) \cdots (x_{k-1}).$$

This can be done by induction on the complexity of τ.

Proof of Theorem 2.5. The well-known calculation shows that we have from (2.8):

$$\mathbf{Y}(u) = (\lambda x.u(x(x)))(\lambda x.u(x(x))) = u(\mathbf{Y}(u)).$$

Thus $\mathbf{Y}(u)$ *is* a fixed point of the function $u(x)$. What is needed is the proof to show that it is the least one.

Let $d = \lambda x.u(x(x))$ and let a be any other fixed point of $u(x)$. To show, as we must, that $d(d) \subseteq a$, it is enough to show that $e_l \subseteq d$ always implies $e_l(e_l) \subseteq a$; because by continuity we have:

$$d(d) = \bigcup \{e_l(e_l) | e_l \subseteq d\}.$$

By way of induction, suppose that this implication holds for all $n < l$. Assume that $e_l \subseteq d$ and that $m \in e_l(e_l)$. We will want to use the induction hypothesis to show that $m \in a$. By the definition of application, there exists an integer n such that $(n, m) \in e_l$ and $e_n \subseteq e_l$. But $n \leqq (n, m) < l$, and $e_n \subseteq d$. By the hypothesis, we have $e_n(e_n) \subseteq a$. Note that $(n, m) \in d$ also, and that d is defined by λ-abstraction; thus,

$m \in d(e_n)$ by definition. By monotonicity $u(e_n(e_n)) \subseteq u(a) = a$; therefore $m \in a$. This shows that $e_l(e_l) \subseteq a$, and the inductive proof is complete.

Remark. Note that we did not actually use the fixed-point theorem in the proof, but we did use rather special properties of the pairing (n, m) and the finite sets e_l.

Equation (2.9) is proved easily from the definition of application; indeed $u(x)$ is distributive in u. Equation (2.10) is proved even more easily from the definition of λ-abstraction. For (2.11), we see that the inclusion from left to right would hold in general by monotonicity. In the other direction, suppose $m \in f(x) \cap g(x)$. Then for suitable k and l we have $(k, m) \in f$ and $e_k \subseteq x$, also $(l, m) \in g$ and $e_l \subseteq x$. Let $e_n = e_k \cup e_l \subseteq x$. Because f and g are *graphs*, we can say $(n, m) \in f \cap g$; and thus $m \in (f \cap g)(x)$. This is the only point where we require the assumption on graphs. Equation (2.12) follows directly from the definition of abstraction. For (2.13), which generalizes (2.9), we can also argue directly from the definition of application. In the case of intersection it is easy to find u_n such that $0 \in u_n(\top)$ for all n, but $\bigcap\{u_n | n \in \omega\} = \bot$.

Equation (2.14) is obvious because the least fixed point of the identity function must be $\bot$. A less mysterious definition would be $\bot = 0 - 1$, but the chosen one is more "logical".

For (2.15) we note that by definition:

$$\lambda z.0 = \{(n, m) | m \in 0\} = \{(n, 0) | n \in \omega\}.$$

Because $0 = (0, 0)$ and $1 = (1, 0)$, we get the hint. Equation (2.16) makes use of $\cup$ for iteration. If $x = 0 \cup (x + 1)$, then x must contain all integers; hence $x = \top$. The iteration for $\cap$ in (2.17) is more complex. The fundamental equation we need is:

$$x \cap y = x \supset (y \supset 0, \bot), ((x - 1) \cap (y - 1)) + 1.$$

This says to compute the intersection of two sets x and y, we first test whether $0 \in x$. If so, then test whether $0 \in y$. If so, then we know $0 \in x \cap y$. In the meantime we begin testing x for positive elements. If we could compute (by the same program) the intersection $(x - 1) \cap (y - 1)$, then we would get the positive elements of the intersection $x \cap y$ by adding one. This is a very slow program, but we can argue by induction that it gives us all the desired elements. Of course, $\cap$ is the least function satisfying this equation.

In the case of (2.18) it is clear that we have:

$$\lambda x.\bot = \{(n, m) | m \in \bot\} = \bot;$$

$$\lambda x.\top = \{(n, m) | m \in \top\} = \top;$$

because in the last every integer is a (number of a) pair. Suppose now that $a = \lambda x.a$ and $a \neq \top$. Let k be the least integer where $k \notin a$. Now $k = (n, m)$ for some n and m. If $m \in a$, then $(n, m) \in a = \lambda x.a$; hence $m \notin a$. But $m \leqq k$ and k is minimal; therefore, $m = k$. But this is only possible if $k = m = n = 0$. Suppose further $a \neq \bot$ and that l is the least integer where $l \in a$. Now $l = (i, j)$ with $j \in a$ and $j \leqq l$. So $j = l$ and $l = j = i = 0$. This contradiction proves that $a = \bot$ or $a = \top$.

Equations (2.19)–(2.22) are definitions, and (2.23) is proved easily by induction on i. Equation (2.24) is also a definition. To prove (2.25) we note that

$\{u_i | i \in x\}$ is continuous (even: distributive) in u and x. Thus, there is a continuous function $\mathbf{seq}(u)(x)$ giving this value. What is required is to prove that it is LAMBDA-definable. We see:

$$\begin{aligned}\mathbf{seq}(u)(x) &= \{n \in u_0 | 0 \in x\} \cup \{m \in \bigcup \{u_{i+1} | i+1 \in x\} | \exists k.k+1 \in x\} \\ &= x \supset u_0, \mathbf{seq}(\lambda t.u_{t+1})(x-1);\end{aligned}$$

that is, **seq** satisfies the fixed-point equation for \$. Thus $\$ \subseteq \mathbf{seq}$. To establish the other inclusion we argue by induction on i for:

$$\forall x,\ u[i \in x \Rightarrow u_i \subseteq \$(u)(x)]$$

This is easy by cases using what we are given about \$ in (2.24); it implies that $\mathbf{seq} \subseteq \$$. Note that:

$$\lambda n \in \omega.\tau = \lambda n \in \omega.\sigma \quad \text{iff} \quad \forall n \in \omega.\tau = \sigma.$$

For primitive recursive functions, even of several variables, there is no trouble in transcribing into LAMBDA-notation any standard definition—especially as we can use the abstraction operator $\lambda n \in \omega$. If we recall that every r.e. set a has the form:

$$a = \{m | \exists n.p(n) = m+1\},$$

where p is primitive recursive, we then see that $a = \hat{p}(\top) - 1$. This means that every r.e. set is LAMBDA-definable.

Proof of Theorem 2.6. In case of a function of several variables, we remark:

$$\begin{aligned}&\lambda x_0 \lambda x_1 \cdots \lambda x_{k-1}.f(x_0)(x_1) \cdots (x_{k-1}) \\ &\quad = \{(n_0, (n_1, (\cdots, (n_{k-1}, m) \cdots))) | m \in f(e_{n_0})(e_{n_1}) \cdots (e_{n_{k-1}})\}.\end{aligned}$$

This makes the implication from (i) to (ii) obvious. Conversely, if a *graph* u is r.e., then from the definition of application we have:

$$m \in u(e_{n_0})(e_{n_1}) \cdots (e_{n_k}) \quad \text{iff} \quad (n_0, (n_1, (\cdots, (n_{k-1}, m) \cdots))) \in u,$$

which is r.e. in $m, n_0, n_1, \cdots, n_{k-1}$. Therefore (i) and (ii) are equivalent.

We have already proved that (ii) implies (iii). For the converse we have only to show that all LAMBDA-definable sets are r.e. For this argument we could take advantage of the combinator theorem, (Theorem 2.4). Each of the six combinators are r.e., and there is no problem of showing that if u and x are r.e., then so is $u(x)$; because it is defined in such an elementary way with existential and bounded universal number quantifiers and with membership in u and in x occurring positively. Explicitly we have:

$$\begin{aligned}m \in u(x) \quad &\text{iff} \quad \exists e_n \subseteq x.(n, m) \in u \\ &\text{iff} \quad \exists n\ \forall m < n[[m \in e_n \text{ implies } m \in x] \text{ and } (n, m) \in u].\end{aligned}$$

For Section 3. For the proof of (3.1) we distinguish cases. In case $x = y = \bot$, we note that $\mathbf{cond}(\bot)(\bot) = \bot$ and $\bot(\bot) = \bot$, so the equation checks in this case. Recall:

$$\mathbf{cond}(x)(y) = \lambda z.z \supset x, y = \{(n, m) | m \in (e_n \supset x, y)\}.$$

We can show $0 \not\in \mathbf{cond}(x)(y)$. Note first $0 = (n, m)$ iff $n = 0 = m$; furthermore, $e_0 = \bot$ and $\bot \supset x$, $y = \bot$; but $0 \not\in \bot$. Also we have:

$$\mathbf{cond}(x)(y)(0) = x \quad \text{and} \quad \mathbf{cond}(x)(y)(1) = y;$$

so if either $x \neq \bot$ or $y \neq \bot$, then $\mathbf{cond}(x)(y) \neq \bot$. In this case, $\mathbf{cond}(x)(y)$ must contain *positive* elements. The result now follows.

Theorem 3.1 is obvious from the construction of **G**, because $\mathbf{G}(\mathbf{G}) = 0$ and $\mathbf{G}(0)(0) = \mathbf{suc}$, and so the $\mathbf{G}(0)(i)$ give us all the other combinators.

The primitive recursive functions needed for (3.4)–(3.6) are standard. Equation (3.7) is a definition—if we rewrote it using the **Y**-operator—and the proof of Theorem 3.2 is easy by induction. There is also no difficulty with (3.8)–(3.12). The idea of the proof of Theorem 3.3 is contained in the statement of the theorem itself. The proof of Theorem 3.4 is already outlined in the text.

Proof of Theorem 3.5. The argument is essentially the original one of Myhill–Shepherdson. Suppose p is computable, total and extensional. Define:

$$q = \{(j, m) | m \in \mathbf{val}(p(\mathbf{fin}(j)))\},$$

where **fin** is primitive recursive, and for all $j \in \omega$:

$$\mathbf{val}(\mathbf{fin}(j)) = e_j.$$

Certainly $q \in \mathbf{RE}$, and we will establish the theorem if we can prove "continuity":

$$\mathbf{val}(p(n)) = \bigcup \{\mathbf{val}(p(\mathbf{fin}(j))) | e_j \subseteq \mathbf{val}(n)\}.$$

We proceed by contradiction. Suppose first we have a $k \in \mathbf{val}(p(n))$, where $k \not\in \mathbf{val}(p(\mathbf{fin}(j)))$ whenever $e_j \subseteq \mathbf{val}(n)$. Pick r to be a primitive recursive function whose range is *not* recursive. Define s, primitive recursive, so that for all $m \in \omega$:

$$\mathbf{val}(s(m)) = \{j \in \mathbf{val}(n) | m \not\in \{r(i) | i \leqq j\}\}.$$

The set $\mathbf{val}(n)$ must be *infinite*, because p is extensional, and if $\mathbf{val}(n) = e_j = \mathbf{val}(\mathbf{fin}(j))$, then $k \not\in \mathbf{val}(p(\mathbf{fin}(j))) = \mathbf{val}(p(n))$. Note that $\mathbf{val}(s(m))$, as a subset of the infinite set, is *finite* if m is in the range of r; otherwise it is equal to $\mathbf{val}(n)$. Again by the extensionality of p we see that $k \in \mathbf{val}(p(s(m)))$ if and only if m is *not* in the range of r. But this puts an r.e. condition on m equivalent to a non-r.e. condition, which shows there is no such k.

For the second case suppose we have a $k \not\in \mathbf{val}(p(n))$, where for a suitable $e_j \subseteq \mathbf{val}(n)$ it is the case that $k \in \mathbf{val}(p(\mathbf{fin}(j)))$. Define:

$$t = \lambda m \in \omega . e_j \cup (\mathbf{val}(m) \supset \mathbf{val}(n), \mathbf{val}(n)).$$

We have:

$$t(m) = \begin{cases} e_j & \text{if } \mathbf{val}(m) = \bot; \\ \mathbf{val}(n) & \text{if not.} \end{cases}$$

We choose u primitive recursive, where:

$$\mathbf{val}(u(m)) = t(m).$$

By the choice of k, and by the extensionality of p, and by the fact that $\mathbf{val}(n) \neq e_j$,

we have:

$$k \in \mathbf{val}(p(u(m))) \quad \text{iff} \quad t(m) = e_j$$
$$\text{iff} \quad \mathbf{val}(m) = \perp.$$

But this is impossible, since one side is r.e. in m and the other is not by Theorem 3.4. As both cases lead to contradiction, continuity is established and the proof is complete.

Proof of Theorem 3.6. Consider a degree $\mathbf{Deg}(a)$. This set is closed under application, because:

$$u(a)(v(a)) = \mathbf{S}(u)(v)(a),$$

and $\mathbf{S}(u)(v)$ is r.e. if both u and v are. Note that it also contains the element $\mathbf{G}$; hence, as a subalgebra, it is generated by a and $\mathbf{G}$.

Let A be any finitely generated subalgebra with generators $a'_0, a'_1, \cdots, a'_{n-1}$. Consider the element $a = \mathbf{cond}(\langle a'_0, a'_1, \cdots, a'_{n-1}\rangle)(\mathbf{G})$. As in the proof of Theorem 3.1, a generates A under application. It is then easy to see why $A = \mathbf{Deg}(a)$.

Proof of Theorem 3.7. We first establish (3.16) and (3.17):

$$\begin{aligned} L \circ \bar{u} \circ R &= \lambda x.L(\bar{u}(\langle 0, x\rangle)) \\ &= \lambda x.L(\langle 1, u, x\rangle) \\ &= \lambda x.u(x). \\ \bar{u} \circ \bar{v} \circ R &= \lambda x.\bar{u}(\bar{v}(\langle 0, x\rangle)) \\ &= \lambda x.\bar{u}(\langle 1, v, x\rangle) \\ &= \lambda x.\overline{u(v)}(x) \\ &= \overline{u(v)}. \end{aligned}$$

Now starting with any $u \in \mathbf{RE}$, we write $u = \tau$, where τ is formed from $\mathbf{G}$ by application alone. By (3.17), we can write $\bar{u}$ in terms of $\bar{\mathbf{G}}$ and $\mathbf{R}$ using only $\circ$. That is, $\bar{u}$ belongs to a special subsemigroup. In view of (3.16), we find that $\lambda x.u(x)$ belongs to that generated by $\mathbf{R}$, $\mathbf{L}$ and $\bar{\mathbf{G}}$. But

$$\mathbf{RE} \cap \mathbf{FUN} = \{\lambda x.u(x) | u \in \mathbf{RE}\},$$

and so the theorem is proved.

For Section 4. The notion of a *continuous lattice* is due to Scott (1970/71) and we shall not review all the facts here. One special feature of these lattices is that the lattice operations of meet and join ($\sqcap$ and $\bigsqcup$) are continuous (that is, commute with directed sups). As topological spaces, they can be characterized as those T_0-spaces satisfying the extension theorem (which we proved for $\mathbf{P}\omega$ in Theorem 1.5).

Proof of Theorem 4.1. Consider a continuous function a, and let $A = \{x | x = a(x)\}$. By the fixed-point theorem (Theorem 1.4) we know that A is nonempty and that it has a least element under $\subseteq$. Certainly A is partially ordered by $\subseteq$; further, A is closed under *directed* unions but not under arbitrary unions.

That is, A is not a complete sublattice of $\mathbf{P}\omega$ with regard to the lattice operations of $\mathbf{P}\omega$, but it could be a complete lattice on its own—if we can show sups exist. Thus, let $S \subseteq A$ be an arbitrary subset of A. By the fixed-point theorem, find the least solution to the equation:

$$y = \bigcup\{x | x \in S\} \cup a(y).$$

Clearly $x \subseteq y$ for all $x \in S$; and so $x = a(x) \subseteq a(y)$, for all $x \in S$. This means that $y = a(y)$, and thus $y \in A$. By construction, then, y is an upper bound to the elements of S. Suppose $z \in A$ is another upper bound for S. It will also satisfy the above equation; thus $y \subseteq z$, and so y is the least upper bound. A partially ordered set with sups also has infs, as is well known, and is a complete lattice.

Suppose that a is a retract. We can easily show that the fixed-point set A (with the relative topology from $\mathbf{P}\omega$) satisfies the extension theorem. For assume $f: X \to A$ is continuous, and $X \subseteq Y$ as a subspace. Now we can also regard $f: X \to \mathbf{P}\omega$ as continuous because A is a subspace of $\mathbf{P}\omega$. By Theorem 1.5 there is an extension to a continuous $\bar{f}: Y \to \mathbf{P}\omega$. But then $a \circ \bar{f}: Y \to A$ is the continuous function we want for A, and the proof is complete.

The space of retracts. Let us define:

$$\mathbf{RET} = \{a | a = a \circ a\},$$

the set of all retracts, which is a complete lattice in view of Theorem 4.1. It will be proved to be not a retract itself by showing it is not a continuous lattice; in fact, the meet operation on **RET** is not continuous on **RET**.

The proof was kindly communicated by Y. L. Ershov and rests on distinguishing some extreme cases of retracts. Call a retract *a nonextensive* if for all nonempty finite sets x we have $x \not\subseteq a(x)$. Call a retract *b finite* if all its values are finite (i.e., $b(\top)$ is finite). If a is nonextensive and b is finite, then Ershov notes that they are "orthogonal" in **RET** in the sense that $c = a \sqcap b = \bot$. The reason is that, since $c \subseteq b$, it is finite; but $c \subseteq a$, too, so $c(x) \subseteq a(x)$ for all x. Because c is a retract, we have $c(x) = c(c(x)) \subseteq a(c(x))$. As $c(x)$ is finite and a is nonextensive, it follows that $c(x) = \bot$ for all x.

This orthogonality is unfortunate, because consider the finite retracts $b_n = \lambda x.e_n$. We have here a directed set of retracts where $\bigcup\{b_n | n \in \omega\} = \lambda x.\top = \top$. If $\sqcap$ were continuous, it would follow that for nonextensive a:

$$a = a \sqcap \top = a \sqcap \bigcup\{b_n | n \in \omega\} = \bigcup\{a \sqcap b_n | n \in \omega\} = \bot,$$

showing that there are no nontrivial such a. But this is not so.

Let $\ll$ be a strict linear ordering of ω in the order type of the rational numbers. Define:

$$a(x) = \{m | \exists n \in x. m \ll n\}.$$

We see at once that a is continuous; and, because $\ll$ is transitive and dense, a is a retract. Since $\ll$ is irreflexive, it is the case for finite nonempty sets x that $\max_{\ll}(x) \notin a(x)$; hence, a is nonextensive. As $a(\top) = \top$, we find $a \neq \bot$. The proof is complete.

Note that there are many transitive, dense, irreflexive relations on ω, so there are many nonextensive retracts. These retracts, like a above, are *distributive.* A

nondistributive example is:

$$a' = \{m | \exists n, n' \in x. n \ll m \ll n'\}.$$

Many other examples are possible.

Proof of Theorem 4.2. The relation $\Subset$ is by the definition of retract reflexive on **RET**; it is also obviously antisymmetric. To prove transitivity, suppose $a \Subset b \Subset c$, then

$$a = a \circ b = a \circ b \circ c = a \circ c.$$

Similarly, $a = c \circ a$. Note, by the way, that $a \Subset b$ implies that the range of a is included in that of b; but that the relationship $a \subseteq b$ does not imply this fact. The relationship $a \Subset b$, however, is stronger than inclusion of ranges.

Proofs of Theorems 4.3–4.5. We will not give full details as all the parts of these theorems are direct calculations. Consider by way of example Theorem 4.3(i). We find:

$$\begin{aligned}(a \multimap b) \circ (a \multimap b) &= \lambda u.b \circ (b \circ u \circ a) \circ a \\ &= \lambda u.b \circ u \circ a \\ &= a \multimap b,\end{aligned}$$

provided that a and b are retracts. A very similar computation would verify part (iv), if one writes out the composition:

$$(a \multimap b') \circ (f \multimap f') \circ (b \multimap a')$$

and uses the equations:

$$f = b \circ f \circ a \quad \text{and} \quad f' = b' \circ f' \circ a'.$$

The main point of the proof of Theorem 4.6 has already been given in the text.

For Sections 5–7. Sufficient hints for proofs have been given in the text.

Appendix B. Acknowledgments and references. My greatest overall debt is to the late Christopher Strachey, who provided not only the initial stimulus and continuing encouragement, but also what may be termed the necessary irritation. Not being a trained mathematician, he often assumed that various operations made sense without looking too closely or rigorously at the details. This was particularly the case with the λ-calculus, which he used as freely as everyday algebra. Having repeatedly and outspokenly condemned the λ-calculus as a formal system without clear mathematical foundations, it was up to me to provide some alternative. The first suggestion was a typed system put forward in Scott (1969) (unpublished, but later developed as LCF by Robin Milner and his collaborators). Experience with the type structure of function spaces, which had come to my attention from work in recursion theory by Nerode, Platek and others, soon convinced me that there were many more similar structures than might at first be imagined. In particular, a vague idea about a space with a "dense" basis led quickly to the more direct construction, by inverse limits, of function spaces of "infinite" type that were very reasonable models of the classical "type-free"

λ-calculus (as well as many other calculi for other "reflexive domains). The details can be found in Scott (1971) and Scott (1973b). Algebra was justified, but the work in doing so was tiring and the exact connections with computability were not all that easy to describe.

In the meantime Plotkin (1972) contained suggestions for a "set-theoretical" construction of models, but not much notice was taken of the plan at the time it was circulated—perhaps owing to a fairly sketchy presentation of the precise semantics of the λ-calculus. The present paper evolved particularly from the project of making the connections with ordinary recursion theory easier to comprehend, since a satisfactory theory of computability and programming language semantics had to face this problem. The idea of using sets of *integers* for a model was first put forward by the author at a meeting at Oberwolfach at Easter in 1973 and in a more definitive form at the Third Scandinavian Logic Symposium shortly thereafter (see Scott (1975a) which is a preliminary and shorter version of this paper). The author gave a report on the model at the Bristol Logic Colloquium in July 1973, but did not submit a paper for the proceedings. A series of several lectures was presented at the Kiel Logic Colloquium in July 1974, covering most of the present paper which was distributed as a preprint at the meeting. The text (but unfortunately neither acknowledgments nor references) was printed in the proceedings (Springer Lecture Notes in Mathematics, vol. 499). In 1973 after experimentation with definitions somehow forced him into the definition of the model, the author realized that it was essentially the same as Plotkin's idea and, even more surprising, it was already implicit in a very precise form in much earlier work by Myhill–Shepherdson (1955) and Friedberg–Rogers (1959) (see also Rogers (1967)) on *enumeration operators*. What had happened was that Plotkin had not made enough tie-up with recursion theory, and the recursive people had not seen the tie-up with λ-calculus, even though they knew that one could do a lot with such operators. Actually, if the author had taken his own advice in 1971 (see *Continuous lattices*, Scott (1972a, end of § 2)), he would have seen that many spaces have their own continuous-function spaces as computable retracts, a fact which is just exactly the basis for the present construction; but instead he said: "it [the representation as a retract] does not seem to be of too much help in proving theorems."

Over the years in work on λ-calculus and programming language semantics, personal contact and correspondence with a large number of people has been very stimulating and helpful. I must mention particularly de Bakker, Barendregt, Bekic, Blikle, Böhm, Curry, Egli, Engeler, Ershov, Goodman, Hyland, Kreisel, Landin, Milne, Milner, Mosses, Nivat, Park, Plotkin, Reynolds, de Roever, Smyth, Stoy, Tang, Tennent, Wadsworth. (I apologize to those I have inadvertently left out of this list.) In the reference list a very imperfect attempt has been made to collect references directly relevant to the topics of this paper as well as pointers to related areas that may be of inspiration for future work. The list of papers is undoubtedly incomplete, and inaccurate as well, but the author hopes it may be of some use for those seeking orientation. It is a very vexing problem to keep such references up to date. Some remarks toward references and acknowledgments on the specific results in the various sections follow. Felipe Bracho deserves special thanks for help in the preparation of the final manuscript and with the reference list.

Section 1. The relevance of the "positive" or "weak" topology first came to the author's attention through the work of Nerode (1959). Continuous functionals were studied by Kleene and Platek and many other researchers in recursion theory. Monotonicity was particularly stressed by Platek (1964). The graphs and the definition of application are used in the same way by Plotkin (1972) and Rogers (1967, see p. 147). The fixed-point theorem is very well-known. See, e.g., Tarski (1955). The extension theorem was formulated by the author, but it is very similar to many results in point-set topology; it was used in a prominent way in Scott (1972a) to characterize continuous lattices. The embedding theorem is well-known; see, e.g., Čech (1966).

Section 2. The language LAMBDA is due to the author. Note in particular that Plotkin and Rogers do not define λ-abstraction, even though they know of the existence of many combinators and *could have* defined abstraction if anyone had ever asked them. In particular, they understood about conversion in many instances. The reduction and combinator theorems are well known from combinatory logic and can be found in any reference. The first recursion theorem is basic to all of elementary recursion theory; what is new here is the adaptation of David Park's proof (Park (1970c), unpublished) to the present model to show that Curry's "paradoxical" combinator actually does the recursion. The definition of computability and the definability theorem tie up the present theory with the older theory of enumeration operators.

Section 3. The idea of reduction to a few combinators is an old one in combinatory logic; the author only needed to find a small trick (formula (3.1)) which would take care of the arithmetical combinators. The ideas for the Gödel numbering and the proof of the second recursion theorem are standard, as is the proof of the incompleteness theorem. It only looks a little different since we combine arithmetic with the "type-free" combinators. The proof of the completeness theorem for definability (Theorem 3.5) is taken directly from Myhill–Shepherdson (1955). The author is indebted to Hyland for pointing this out. The subalgebra theorem is an easy reformulation of talk about enumeration degrees; for more information on such degrees consult Rogers (1967), Sasso (1975), and also Gutteridge (1971). The area is underdeveloped as compared to Turing degrees. Semigroups of combinators have been discussed by Church and Böhm.

Section 4. The notion of a *retract* is common in topology, but the idea of using them to define data types and of having a calculus of computable retracts is original with the author. Of course the connection between lattices and fixed points was known; more about lattices is to be found in Scott (1972a). The various operations on retracts and the idea of using fixed-point equations to define retracts recursively are due to the author. Applications to semantics were given in Scott (1971) for flow diagrams, and this has been followed up by many people, in particular Goguen, et al. (1975) and Reynolds (1974b).

Section 5. Algebraic lattices have been known for a long time (see Gratzer (1968)) and also closure operations (see, e.g., Tarski (1930)). It was Per Martin-Löf and Peter Hancock who suggested that they might form a "universe"; in particular the construction of $\mathbf{V}$ is essentially due to them. The limit theorem is due to the author.

Section 6. More information on the classification by notions in descriptive set theory of various subsets can be found in the work of Tang who also makes

connections with the work of Wadge. The various normal forms for the classes of sets (e.g., Table 3) are due to the author.

Section 7. Functionality has been studied for some time in combinatory logic (see, e.g., Hindley, et al. (1972) for an introduction). The author had the idea to see what it all means in the models; there are, of course, connections going back to Curry and Kleene, with functional interpretations of intuitionistic logic (cf. Theorem 7.3, which is well-known). The proof of Theorem 7.4 is due to Plotkin.

Appendix A. After the main body of the paper was written, Y. L. Ershov solved the author's problem about the space of retracts. Ershov's proof is presented after the discussion of the proof of Theorem 4.1 in this Appendix. Quite independently Hosono and Sato (1975) found almost exactly the same proof. Before corresponding with Ershov, the author was totally unaware of the connections with and the importance of Ershov's extensive work in "numeration" theory (see citations in the reference list).

Appendix B. All defects are due to the author.

REFERENCES

L. AIELLO, M. AIELLO AND R. W. WEYHRAUCH (1974), *The semantics of PASCAL in LCF*, Artificial Intelligence Lab. Memo. AIM-221, Stanford Univ., Stanford, Calif.

S. ALAGIC (1974), *Categorical theory of tree processing*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 80–99.

M. A. ARBIB AND E. G. MANES (1974a), *Fuzzy morphisms in automata theory*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 98–105.

——— (1974b), *Categorists' view of automata and systems*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 62–79.

——— (1974c), *Basic concepts of category theory applicable to computation and control*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 2–41.

J. W. DE BAKKER (1971a), *Recursive procedures*, Mathematical Centre Tracts, vol. 24, Amsterdam.

——— (1971b), *Recursion, induction and symbol manipulation*, Proc. MC-25 Informatica Symposium, Mathematical Centre Tracts, vol. 38, Amsterdam.

J. W. DE BAKKER AND W. P. DE ROEVER (1972), *A Calculus for Recursive Program Schemes*, Proc. IRIA Colloq. on Automata, North-Holland, Amsterdam.

J. W. DE BAKKER AND D. SCOTT (1969), *A theory of programs*, Unpublished notes, IBM Seminar, Vienna, 1969.

J. W. DE BAKKER (1969), *Semantics of Programming Languages*. Advances in Information Systems Science, vol. 2, Plenum Press, New York.

J. W. DE BAKKER AND L. G. L. MEERTENS (1973), *On the completeness of the inductive assertion method*, Mathematical Centre Rep. IW 12/72, Amsterdam.

——— (1974), *Fixed points in programming theory*, Foundations of Computer Science, J. W. de Bakker, ed., Mathematical Centre Tract, vol. 63, Amsterdam.

H. P. BARENDREGT (1971), *Some extensional term models for combinatory logics and λ-calculi*. Ph.D. thesis, Univ. of Utrecht.

H. BEKIĆ (1971), *Towards a mathematical theory of processes*, Tech. Rep. TR 25.125, IBM Lab., Vienna.

——— (1969), *Definable operations in general algebra, and the theory of automata and flowcharts*, Rep., IBM Lab., Vienna.

H. BEKIĆ ET AL. (1974), *A formal definition of a PL/1 subset, Parts I and II*, Tech. Rep. TR 25.139, IBM Lab., Vienna.

R. S. BIRD (1974), *Unsolvability of the inclusion problem for DBS schemas*, Rep. RCS22, Dept. of Computer Sci., Univ. of Reading.

G. BIRKHOFF (1967), *Lattice Theory*, 3rd ed., Colloquium Publications, vol. 25, American Mathematical Society, Providence, R.I.

A. BLIKLE (1971), *Algorithmically definable functions: A contribution towards the semantics of programming languages*, Dissertationes Math. Rozprawy Mat., 85.

——— (1972), *Equational languages*, Information Control, pp. 134–147.

——— (1973), *An algebraic approach to programs and their computations*, Proc. Symp. and Summer School on the Mathematical Foundations of Computer Sci., High Tatras, Czechoslovakia.

A. BLIKLE AND A. MAZURKIEWICZ (1972), *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, Proc. Internat. Sympos. and Summer School on the Mathematical Foundations of Computer Sci., Warsaw–Jablonna.

A. BLIKLE (1974), *Proving programs by sets of computations*, Proc. 3rd Symp. on the Mathematical Foundation of Computer Sci., Jadwisin, Poland; Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1974.

C. BÖHM (1968), *Alcune proprieta delle forme β-η-normal Nel λ-κ-calcolo.* Consiglio Nazionale del e Richerche: Publicazione 696 dell' Instituto per le Applicazioni del Calcolo, Roma.

——— (1966), *The CUCH as a formal and descriptive language.* Formal Language Description Languages for Computer Programming, T. B. Steel, ed., North-Holland, Amsterdam, pp. 179–197.

——— (1975), *λ-Calculus and computer science theory*, Proc. Rome Symp., March 1975, Lecture Notes in Computer Science, vol. 37, Springer-Verlag, Berlin.

R. M. BURSTALL (1972a), *Some techniques for proving the correctness of programs which alter data structures*, Machine Intelligence vol. 7, B. Meltzer and D. Michie, eds., Edinburgh, pp. 23–50.

——— (1969), *Proving properties of programs by structural induction*, Comput. J., 12, pp. 41–48.

——— (1972b), *An algebraic description of programs with assertions, verification and simulation*, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, New Mexico, pp. 7–14.

R. BURSTALL AND J. W. THATCHER (1974), *The algebraic theory of recursive program schemes*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 154–160.

J. M. CADIOU (1973), *Recursive definitions of partial functions and their computations*, Ph.D. thesis, Computer Sci. Dept., Stanford Univ., Stanford, Calif.

J. M. CADIOU AND J. J. LEVY (1973), *Mechanizable proofs about parallel processes*, Presented at the 14th Ann. Symp. on Switching and Automata Theory.

J. W. CASE (1971), *Enumeration reducibility and the partial degrees*, Ann. Math. Logic, 2, pp. 419–440.

E. ČECH (1966), *Topological Spaces*, Prague.

A. K. CHANDRA (1974), *Degrees of translatability and canonical forms in program schemas part 1*, IBM Res. Rep. RC4733, Yorktown Heights, N.Y.

——— (1974a), *The Power of Parallelism and Nondeterminism in Programming*, IBM Res. Rep. RC4776, Yorktown Heights, N.Y.

——— (1974b), *Generalized program schemas*, IBM Res. Rep. RC4827, Yorktown Heights, N.Y.

A. K. CHANDRA AND Z. MANNA (1973), *On the power of programming features*, Artificial Intelligence Memo. AIM-185, Stanford Univ., Stanford, Calif.

A. CHURCH (1951), *The calculi of lambda-conversion*, Annals of Mathematical Studies, vol. 6, Princeton University Press, Princeton, N.J.

M. J. CLINT (1972), *Program proving: Co-routines*, Acta Informatica, 2, pp. 50–63.

R. C. CONSTABLE AND D. GRIES (1972), *On classes of program schemata*, this Journal, 1, pp. 66–118.

D. C. COOPER (1966), *The equivalence of certain computations*, Comput. J., 9, pp. 45–52.

B. COURCELLE, G. KAHN AND J. VUILLEMIN (1974), *Algorithmes d'equivalince pour des equations recursive simples*, Proc. 2nd Colloq. on Automata Languages and Programming, Saarbrucken; Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 213–220.

B. COURCELLE AND J. VUILLEMIN (1974), *Completude d'un systeme formel pour des equations recursives simples*, Compte-Rendu du Colloque de Paris.

——— (1974), *Semantics and axiomatics of a simple recursive language*, Rapport Laboria, vol. 60, IRIA: Also: 6th Ann. ACM Symp. on the Theory of Computing, Seattle, Washington.

H. B. CURRY AND R. FEYS (1958), *Combinatory Logic*, vol. 1, North-Holland, Amsterdam.

H. B. CURRY, J. R. HINDLEY AND J. SELDIN (1971), *Combinatory Logic*, vol. 2, North-Holland, Amsterdam.

M. DAVIS (1958), *Computability and Unsolvability*, McGraw-Hill, New York.

E. W. DIJKSTRA (1974), *A simple axiomatic basis for programming language constructs.* Indag. Math., 36, pp. 1–15.

——— (1976), *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, N.J., to appear.

——— (1972), *Notes on structured programming,* Structural Programming, C. A. R. Hoare, E. W. Dijkstra and O. J. Dahl, Academic Press, New York.

J. DONAHUE (1974), *Scottery Child's Guide No. 1,* Dept. of Comput. Sci., Univ. of Toronto.

B. N. DUONG (1974), *A high level, variable free calculus for recursive programming,* Comput. Sci. Dept. Rep. CS 74 03, Univ. of Waterloo.

S. EILENBERG AND C. C. ELGOT (1970), *Recursiveness,* Academic Press, New York.

——— (1974), *Automata, Languages and Machines,* vol. A, Academic Press, New York (vol. B in press).

C. C. ELGOT (1971), *Algebraic theories and program schemes,* Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York.

——— (1967), *Abstract algorithms and diagram closure,* IBM Res. Rep. RC 1750, Yorktown Heights, N.Y.

C. C. ELGOT, A. ROBINSON AND J. D. RUTLEDGE (1966), *Multiple control computer models,* IBM Res. Rep. RC 1622, Yorktown Heights, N.Y.

M. H. VAN EMDEN AND R. A. KOWALSKI (1974), *The semantics of predicate logic as a programming language,* Memo. no. 83 (MIP-R-103), Dept. of Machine Intelligence, Univ. of Edinburgh.

E. ENGELER (1968), *Formal languages: Automata and structures,* Lectures in Advanced Mathematics, Markham.

——— (1974), *Algorithmic Logic, Foundations of Computer Science,* J. W. de Bakker, ed., Mathematical Centre Tract MCT 63, Amsterdam.

——— (1967), *Algorithmic properties of structures,* Mathematical Systems Theory, 1, pp. 183–195.

J. ENGELFREIT (1974), *Simple program schemes and formal languages,* Springer Lecture Notes in Computer Science, vol. 20.

J. ENGELFREIT AND E. M. SCHMIDT (1975), *IO and OI,* DAIMI PB-47, Matematisk Institut, Aarhus Universitet, Datalogisk Afdeling, Denmark.

JN. L. ERSHOV (1971), *Computable numerations of morphisms,* Algebra and Logic, 10, pp. 247–308.

——— (1972a), *Computable functions of finite types,* Ibid., 11, pp. 367–437.

——— (1972b), *Everywhere defined (total) continuous functionals,* Algebra and Logic, 11, pp. 656–665.

——— (1972d), *Continuous lattices and A-spaces,* Dockl. Acad. Nauk USSR, 207, pp. 523–526.

——— (1972e), *Theory of A-spaces,* Algebra and Logic, 12, pp. 369–916.

——— (1973), *Theorie der Numerierungen,* Veb. Deutscher Verlag der Wissenschaften, Berlin.

A. E. FISCHER AND M. J. FISCHER (1973), *Mode modules as representations of domains,* Proc. ACM Symp. on Principles of Programming Languages, Boston, pp. 139–143.

M. J. FISCHER (1972), *Lambda calculus schemata,* Proc. ACM Conf. Proving Assertions about Programs, Las Cruces, N.M., pp. 104–109.

R. W. FLOYD (1967), *Assigning meanings to programs,* Proc. Symp. in Appl. Math., vol. 19, Mathematical Aspects of Computer Science, J. T. Schwartz, ed., pp. 33–41.

M. M. FOKKINGS (1974), *Inductive Assertion Patterns for Recursive Procedures,* Compte-Rendu de Colloque de Paris.

R. M. FRIEDBERG AND H. R. ROGERS (1959), *Reducibility and completeness for sets of integers,* Z. Math. Logik Grunlagen Math., 5, pp. 117–125.

H. FRIEDMAN (1971), *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory,* Logic Colloquium 1969, North-Holland, Amsterdam, pp. 361–389.

S. J. GARLAND AND D. C. LUCKHAM (1972), *Translating recursion schemes into program schemes,* Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N.M., pp. 83–96.

H. GERICKE (1966), *Lattice Theory,* Harrap.

J. A. GOGUEN (1967), *L-fuzzy sets,* J. Math. Anal. Appl., 18, pp. 145–174.

——— (1969), *Categories of L-sets,* Bull. Amer. Math. Soc., 75, pp. 524–622.

——— (1972), *On homomorphisms, simulations, correctness, subroutines and termination for programs and program schemes,* Proc. 13th IEEE Conf. on Switching and Automata Theory, pp. 52–60.

——— (1974), *Semantics of computation*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 234–249.

J. A. GOGUEN ET AL. (1975), *Initial algebra semantics*, IBM Res. Rep. RC 5243, Yorktown Heights, N.Y.

M. J. C. GORDON (1973a), *Models of pure LISP*, Experimental Programming Rep. 31, School of Artificial Intelligence, Univ. of Edinburgh. (Also Ph.D. thesis, *Evaluation and denotation of pure LISP programs: A worked example in semantics.*)

——— (1973b), *An extended abstract of "Models of pure LISP"*, Memo. SAI-RM-7, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1975a), *Operational Reasoning and denotational semantics*, Artificial Intelligence Memo. 264, Stanford Univ., Stanford, Calif.

——— (1975b), *Towards a semantic theory of dynamic binding*, Artificial Intelligence Memo. 265, Stanford Univ., Stanford, Calif.

G. GRATZER (1968), *Universal Algebra*, Van Nostrand, New York.

I. GUESSARIAN (1974), *Sur une reduction des schemas de programmes polyadiques a des schemas monadiques et ses applications*, Compte-Rendu de Colloque de Paris.

L. GUTTERIDGE (1971), *Some Results on Enumeration Reducibility*, Ph.D. dissertation, Simon Fraser Univ., Burnaby, B.C., Canada.

J. R. HINDLEY, B. LERCHER AND J. P. SELDIN (1972), *Introduction to combinatory logic*, London Mathematical Society Lecture Note Series, vol. 7, Cambridge University Press, Cambridge, England.

J. R. HINDLEY AND G. MITSCHKE (1975), *Some remarks about the connections between combinatory logic and axiomatic recursion theory*, Preprint 203, Fachbereich Mathematik, Technische Hochschule Darmstadt.

P. HITCHCOCK (1974), *An approach to formal reasoning about programs*, Ph.D. thesis, Univ. of Warwick, England.

P. HITCHCOCK AND D. M. R. PARK (1972), *Induction rules and proofs of termination*, Proc. Colloques IRIA, Theorie des Automates des Languages et de la Programmation.

C. A. R. HOARE (1969), *An axiomatic basis of computer programming*, Comm. ACM, 12, pp. 576–580, 583.

——— (1971a), *Proof of a program: FIND*, Ibid., 14, pp. 39–45.

——— (1971b), *Procedures and parameters: An axiomatic approach*, Symposium on Semantics of Programming Languages, E. Engeler, ed., Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York, pp. 102–116.

——— (1972), *Notes on data structuring*, Structured Programming, C. A. R. Hoare, E. W. Dijkstra and O. J. Dahl, Academic Press, New York.

C. A. R. HOARE AND P. LAUER (1974), *Consistent and complementary formal theories of the semantics of programming languages*, Acta Informatica, 3, pp. 135–153.

C. A. R. HOARE AND N. WIRTH (1972), *An axiomatic definition of the programming language PASCAL*, Bericht der Fachgrüppe Computer-Wissenschaften 6, Eidenossische Technische Höchschule, Zürich.

C. HOSONO AND M. SATO (1975), *A solution to Scott's problem: "Do the retracts in $P\omega$ form a continuous lattice?"*, Research Inst. for Mathematical Sci., Kyoto (preprint).

G. HOTZ (1966), *Eindeutigkeit und Mehrdeutigkeit formaler Sprachen, Electron. Informationsverarbeit*, Kybernetik (Berlin), 2, pp. 235–246.

J. M. E. HYLAND (1975a), *Recursion theory on the countable functionals*, D. Phil. thesis, Oxford Univ., Oxford, England.

——— (1975b), *A survey of some useful partial order relations in terms of the lambda calculus*, Proc. Conf. on λ-Calculus and Computer Science Theory, Rome, pp. 83–95.

——— (to appear), *A syntactic characterization of the equality in some models for the lambda calculus*, J. London Math. Soc.

Y. I. IANOV (1970), *The logical scheme of algorithms*, Problems in Cybernetics, vol. 1, Pergamon Press, New York.

S. IGARASHI (1972), *Admissibility of fixed point induction in first order logic of typed theories*, Artificial Intelligence Memo. AIM-168, Computer Sci. Dept., Stanford Univ., Stanford, Calif.

K. INDERMARK (1974), *Gleichungsdefinierbarkeit in Relationalstukturen*, Habilitationsschrift, Mathematisch Naturwissen-schaftlichen Fakultat der Rheinischen Freidrich-Wilhelms-Universatat, Bonn, W. Germany.

C. B. JONES (1974), *Mathematical semantics of goto: Exit formulation and its relation to continuations*, preprint.

G. KAHN (1973), *A preliminary theory of parallel programs*. Rapport Laboria IRIA.

D. M. KAPLAN (1969), *Regular expressions and the equivalence of programs*, J. Comput. System Sci., 3, pp. 361–385.

R. M. KARP AND R. E. MILLER (1969), *Parallel program schemata*, Ibid., 3, pp. 147–195.

D. J. KFOURY (1972), *Comparing algebraic structures up to algorithmic equivalence*, Automata, Languages and Programming, M. Nivat, ed., North-Holland, Amsterdam, pp. 253–263.

——— (1974), *Translatability of schemas over restricted interpretations*, J. Comput. System Sci., 8, pp. 387–408.

S. C. KLEENE (1950), *Introduction to Metamathematics*, Van Nostrand, New York.

B. KNASTER (1928), *Un theorème sur les fonctions d'ensembles*, Ann. Soc. Polon. de Math., 8, pp. 133–134.

P. J. LANDIN (1964), *The mechanical evaluation of expressions*, Comput. J. 6, pp. 308–320.

——— (1965), *A correspondence between ALGOL 60 and Church's lambda notation*, Comm. ACM, 8, pp. 89–101.

——— (1966a), *A λ-Calculus approach*, Advances in Programming and Non-numerical Computation, L. Fox, ed., Pergamon Press, New York, pp. 97–141.

——— (1966b), *The next 700 programming languages*, Comm. ACM, 9, pp. 157–164.

——— (1966a), *A formal description of ALGOL 60*, Formal Language Description Languages for Computer Programming, T. B. Steel, ed., North-Holland, Amsterdam, pp. 266–294.

——— (1969), *A program/machine symmetric automata theory*, Machine Intelligence, vol. 5, B. Meltzer and D. Michie, eds., Edinburgh University Press, pp. 99–120.

P. J. LANDIN AND R. M. BURSTALL (1969), *Programs and their proofs: An algebraic approach*, Machine Intelligence, vol. 4, American Elsevier, New York, pp. 17–44.

J. LESZCZYKOWSKI (1971), *A theorem on resolving equations in the space of languages*, Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys., 19, pp. 967–970.

C. H. LEWIS AND B. K. ROSEN (1973–74), *Recursively defined data types, Parts I and II*. IBM Res. Reps. RC 4429 (1973), RC 4713 (1974), Yorktown Heights, N.Y.

B. LISKOV AND S. ZILLES (1973), *An approach to abstraction computation structures*, Group Memo. 88, Project MAC, Mass. Inst. of Tech., Cambridge, Mass.

D. LUCKHAM, D. M. R. PARK AND M. PATERSON (1970), *On formalized computer programs*, J. Comput. System Sci., 4, pp. 220–249.

S. MACLANE (1972), *Categories for the Working Mathematician*, Springer-Verlag, New York.

E. MANES (1974), *Category Theory Applied to Computation and Control*, Proc. 1st Internat. Symp. Math. Dept. and the Dept. of Computer and Information Sci., Univ. Mass., Amherst. Also: Lecture Notes in Computer Science, vol. 26, Springer-Verlag, New York.

Z. MANNA (1969), *The correctness of programs*, J. Comput. System Sci., 3, pp. 119–127.

——— (1974), *Mathematical Theory of Computation*, McGraw-Hill, New York.

Z. MANNA AND J. M. CADIOU (1972), *Recursive definitions of partial functions and their computations*, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N.M., pp. 58–65.

Z. MANNA AND J. MCCARTHY (1970), *Properties of programs and partial function logic*, Machine Intelligence 5, B. Meltzer and D. Michie, eds., Edinburgh University Press, pp. 27–38.

Z. MANNA, Z. NESS AND J. VUILLEMIN (1972), *Inductive methods for proving properties of programs*, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, N.M., pp. 27–50.

Z. MANNA AND A. PNUELI (1970), *Formalization of properties of functional programs*, J. Assoc. Comput. Mach., 17, pp. 555–569.

——— (1972), *Axiomatic approach to total correctness of programs*, Artificial Intelligence Lab. Memo. AIM-210, Stanford Univ., Stanford, Calif.

Z. MANNA AND J. VUILLEMIN (1972), *Fixpoint approach to the theory of computation*, Comm. ACM, 15, pp. 528–536.

G. MARKOWSKY (1974), *Categories of chain-complete posets*, RC 5100, Comput. Sci. Dept. IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

——— (1974), *Chain-complete posets and directed sets with applications*, RC 5024, IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

A. MAZURKIEWICZ (1973), *Proving properties of processes*, PRACE CO PAN-CC PAS Reports, vol. 134, Warsaw.

——— (1971), *Proving algorithms by tail functions*, Working Paper for IFIP WG2.2, Feb. 1970. Since published in: Information and Control, 18 (1971), pp. 220–226.

J. MCCARTHY (1963a), *A basis for a mathematical theory of computation*, Computer Programming and Formal Systems, D. Braffort and D. Hirshberg, eds., North-Holland, Amsterdam, pp. 37–70.

——— (1962), *The LISP 1.5 Programmers' Manual*, MIT Press, Cambridge, Mass.

——— (1963b), *Towards a mathematical science of computation*, Information Processing 1962, Proc. IFIP Congress 1962, C. M. Poppleworth, ed., North-Holland, Amsterdam, pp. 21–28.

——— (1966), *A formal description of a subset of ALGOL*, Formal Language Description Languages for Computer Programming, R. B. Steel, ed., North-Holland, Amsterdam, pp. 1–12.

J. MCCARTHY AND J. PAINTER (1967), *Correctness of a computer for arithmetic expressions*, Mathematical Aspects of Computer Science, J. T. Schwartz, ed., Proc. of a Symposium in Applied Mathematics, vol. 19, pp. 33–41.

R. E. MILNE (1974), *The formal semantics of computer languages and their implementations*, Ph.D. thesis, Cambridge Univ., Cambridge, England. [Also: Technical Monograph PRG-13, Oxford Univ. Computing Lab., Programming Research Group].

R. MILNER (1972), *Implementation and Application of Scott's Logic for Computable Functions*, Proc. ACM Conf., Las Cruces, N.M., pp. 1–6.

——— (1973), *Models of LCF*, Artificial Intelligence Memo. AIM-186, Computer Sci. Dept., Stanford Univ., Stanford, Calif.

——— (1969), *Program schemes and recursive function theory*, Machine Intelligence 5, B. Meltzer and D. Michie, eds., Edinburgh University Press, pp. 39–58.

——— (1970a), *Algebraic theory of computable polyadic functions*, Computer Sci. Memo., vol. 12, University College, Swansea.

——— (1970b), *Equivalences on program schemes*, J. Comput. System Sci., 4, pp. 205–219.

R. MILNER AND R. WEYHRAUCH (1972), *Proving compiler correctness in a mechanized logic*. Machine Intelligence, 7, pp. 51–72.

R. MILNER (1973a), *An approach to the semantics of parallel programs*, Proc. Convgno di Informatica Teorica, Instituto di Elaborazione delle Informazione, Pisa, Italy.

——— (1973b), *Processes: A mathematical model of computing agents*, Proc. Colloq. in Mathematical Logic, Bristol, England.

F. LOCKWOOD MORRIS (1973), *Advice on structuring compilers and proving them correct*, Proc. SIGACT/SIGPLAN Symp. on Principles of Programming Languages, Boston, pp. 144–152.

——— (1970), *The next 700 programming language descriptions*, Computer Center, Univ. of Essex (typescript).

——— (1972), *Correctness of translations of programming languages*, Computer Sci. Memo. CS 72-303, Stanford Univ., Stanford, Calif.

J. H. MORRIS (1968), *λ-calculus models of programming languages*, Ph.D. thesis, Sloan School of Management, MIT MAC Reprint TR-57, Mass. Inst. of Tech., Cambridge, Mass.

——— (1971), *Another recursion induction principle*, Comm. ACM, 14, pp. 351–354.

——— (1973), *Types are not sets*, Proc. ACM Symp. on Principle of Programming Languages, Boston, pp. 120–124.

——— (1972), *A correctness proof using recursively defined functions*, Formal Semantics of Programming Languages, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, N.J., pp. 107–224.

J. MORRIS AND R. NAKAJIMA (1973), *Mechanical characterisation of the partial order in lattice models of λ-calculus*, Tech. Rep. 18, Univ. of Calif., Berkeley.

P. D. MOSSES (1974), *The mathematical semantics of ALGOL 60*, Tech. Monograph PRG-12, Oxford Univ. Computing Lab., Programming Research Group.

——— (1975a), *The semantics of semantic equations*, Mathematical Foundations of Computer Science, Goos and Hartmanis, eds., Lecture Notes in Computer Science, vol. 28, Springer-Verlag, New York, pp. 409–422.

——— (1975b), *Mathematical semantics and compiler generation*, Thesis, Oxford Univ., Oxford, England.

J. MYHILL AND J. C. SHEPHERDSON (1955), *Effective operations on partial recursive functions*, Z. Math. Logik Grundlagen Math., 1, pp. 310–317.

R. NAKAJIMA (1975), *Infinite normal forms for λ-calculus*, λ-Calculus and Computer Science Theory, C. Böhm, ed., Springer-Verlag, Berlin, pp. 62–82.

A. NERODE (1959), *Some Stone spaces and recursion theory*, Duke Math. J., 26, pp. 397–406.

M. NIVAT (1972), *Languages Algebriques sur le Magma Libre et Semantique des Schemes de Programmes*, Proc. IRIA Symposium on Automata Formal Languages and Programming, North-Holland, Amsterdam.

D. C. OPPEN (1975), *On logic and program verification*, Tech. Rep. 82, Dept. Computer Sci., Univ. of Toronto.

D. M. R. PARK (1970a), *Fixpoint induction and proofs of program properties*, Machine Intelligence, 5, B. Meltzer and D. Michie, eds., American Elsevier, New York, pp. 59–78.

——— (1970b), *Notes on a formalism for reasoning about schemes*, Univ. of Warwick, unpublished notes.

——— (1970c), *The Y combinator Scott's lambda-calculus models*, Univ. of Warwick, unpublished notes.

——— (1970d), *Finiteness is Mu-ineffable*, Theory of Computation Report, No. 3, Dept. of Computer Sci., Univ. of Warwick (1974).

M. S. PATERSON (1963), *Program schemata*, Machine Intelligence, 3, D. Michie, ed., American Elsevier, New York, pp. 19–32.

M. S. PATERSON AND C. S. HEWITT (1970), *Comparative schematology*. Record of Project MAC Conf. on Concurrent Systems and Parallel Computation, Association for Computing Machinery, New York, pp. 119ff.

R. PLATEK (1964), *New foundations for recursion theory*, Thesis, Stanford Univ., Stanford, Calif., unpublished.

G. D. PLOTKIN (1972), *A set-theoretical definition of application*, Memo. MIP-R-95, School of Artificial Intelligence, University of Edinburgh.

——— (1973a), *The λ-calculus is ω-incomplete*, Res. Memo. SAI-RM-2, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1973b), *Lambda-definability and logical relations*. Memo. SAI-RM-4, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1973c), *Call by name, Call-by-value and the λ-calculus, I*, Res. Memo. SAI-RM-6, School of Artificial Intelligence, Univ. of Edinburgh.

——— (1975), *A powerdomain construction*, Dept. of Artificial Intelligence Res. Rep. 3, Univ. of Edinburgh.

H. RASIOWA (1973), *On ω^+-valued algorithmic logic and related problems*, Supplement to Proc. Symp. and Summer School on Mathematical Foundations of Computer Sci., High Tatras, Czechoslovakia.

P. RAULEFS (1975a), *The overtyped lambda calculus*, Interner Bericht Nr. 2, Institut für Informatik, Universität Karlsruhe.

——— (1975b), *Standard models of the overtyped lambda-calculus*, Interner Bericht Nr. 3, Institut für Informatik, Universitat Karlsruhe.

R. R. REDIEJOWSKI (1972), *The theory of general events and its application to parallel programming*, T.P. 18.220 IBM Nordic Laboratory, Sweden.

J. C. REYNOLDS (1969), *Gedanken: A Simple Typeless Language*, Reprint 7621, Argonne National Laboratory, Argonne, Ill.

——— (1972a), *Definitional interpreters for higher order programming languages*, Proc. ACM 25th Nat. Conf., Boston, vol. 2, pp. 717–740.

——— (1972b), *Notes on a lattice-theoretic approach to the theory of computation*, Systems and Information Sci. Dept. Syracuse Univ., Syracuse, N.Y.

——— (to appear), *Towards a theory of type structure*, Colloq. on Programming Paris, April 1974, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

——— (1974a), *On the relation between direct and continuation semantics*, Proc. 2nd Colloq. on Automata, Languages and Programming, Saarbrucken, Lecture Notes in Computer Science, vol. 14, Springer-Verlag, Berlin, pp. 141–156.

——— (1974b), *Semantics of the lattice of flow diagrams*, Systems and Information Science Dept., Syracuse Univ., Syracuse, N.Y. (preprint).

J. C. REYNOLDS (to appear), *On the interpretations of Scott's domains*, Symposia Mathematica.

W. P. DE ROEVER (1974a), *Recursion and parameter mechanisms: An axiomatic approach*, Proc. 2nd Colloq. on Automata Languages and Programming, Saarbrucken, Lecture Notes in Computer Science, vol. 14, Springer-Verlag, Berlin, pp. 34–65.

——— (1974b), *Operational, mathematical and axiomatized semantics for recursive procedures and data structures*, Mathematical Centre Report ID/1, Amsterdam.

H. R. ROGERS (1967), *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

B. K. ROSEN (1973), *Tree-manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20.

J. D. RUTLEDGE (1970a), *Program schemata as automata, part 1*, IBM Res. RC 3098; J. Comput and System Sci., 7 (1973), pp. 543–578.

——— (1970b), *Parallel processes · · · Schemata and transformation*, IBM Res. Rep. RC 2912, Yorktown Heights, N.Y.

A. SALWICKI (1970a), *Formalized algorithmic languages*, Bull. Acad. Polon. Sci. Sér. Math. Astronom. Phys., 18, pp. 227–232.

——— (1970b), *On the equivalence of FS-expressions and programs*, Ibid., 18, pp. 275–278.

——— (1970c), *On the predicate calculi with iteration quantifiers*, Ibid., 18, pp. 279–286.

J. G. SANDERSON (1973), *The lambda calculus, lattice theory and reflexive domains*, Mathematical Institute Lecture Notes, Oxford, England.

L. P. SASSO (1971), *Degrees of unsolvability of partial functions*, Ph.D. dissertation, Univ. of Calif., Berkeley.

——— (1975), *A survey of partial degrees*, J. Symbolic Logic, 40, pp. 130–140.

——— (1973), *A minimal partial degree*, Proc. Amer. Math. Soc., 38, pp. 388–392.

D. SCOTT (1969), *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Unpublished notes, Oxford, England.

——— (1970), *Outline of a mathematical theory of computation*, Proc. 4th Ann. Princeton Conf. on Information Sciences and Systems, pp. 169–176. [Also: Tech. Monograph PRG-2, Oxford Univ. Computing Lab., Programming Research Group (1970).]

——— (1971), *The lattice of flow diagrams*, Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Lecture Notes in Mathematics, vol. 188, Springer-Verlag, New York, pp. 311–366. [Also: Tech. Monograph PRG-3, Oxford Univ. Computing Lab., Programming Research Group (1970)].

——— (1972a), *Continuous lattices*, Proc. 1971 Dalhousie Conference, Lecture Notes in Mathematics, vol. 274, Springer-Verlag, New York, pp. 97–136. [Also: Tech. Monograph PRG-7, Oxford Univ. Computing Lab., Programming Research Group (1971).]

——— (1972b), *Mathematical concepts in programming language semantics*, AFIPS Conf. Proc., vol. 40, pp. 225–234.

——— (1973a), *Data types as lattices*, Unpublished lecture notes, Mathematical Centre, Amsterdam, 1973.

——— (1973b), *Models for various type-free calculi*, Proc. IVth Internat. Cong. for Logic, Methodology and the Philosophy of Science, Bucharest, P. Suppes et. al., eds., North-Holland, Amsterdam, pp. 157–187.

——— (1975a), *Lambda calculus and recursion theory*, Proc. 3rd Scandinavian Logic Symposium, Stig Kanger, ed., North-Holland, Amsterdam, pp. 154–193.

——— (1975b), *Combinators and classes, λ-Calculus and computer science theory*, Proc. Rome Symp., March 1975; Lecture Notes in Computer Science, vol. 37, Springer-Verlag, Berlin, pp. 1–26.

——— (1975c), *Some philosophical issues concerning theories of combinators, λ-Calculus and computer science theory*, Proc. Rome Symp., March 1975; Lecture Notes in Computer Science, vol. 37, Springer-Verlag, Berlin, pp. 346–366.

D. SCOTT AND C. STRACHEY (1971), *Toward a mathematical semantics for computer languages*, Proc. Symposium on Computers and Automata, Polytechnic Inst. of Brooklyn, vol. 21, pp. 19–46. [Also: Technical Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971).]

J. C. SHEPERDSON (1975), *Computation over abstract structures: Serial and parallel procedures and Friedman's effective definitional schemes*, Logic Colloquium 1973, H. E. Rose and J. C. Sheperdson, eds., North-Holland, Amsterdam, pp. 445–513.

C. STRACHEY (1966), *Towards a formal semantics*, Formal Language Description Languages for Computer Programming, T. B. Steel, ed., North-Holland, Amsterdam, pp. 198–220.

——— (1967), *Fundamental concepts in programming languages*, Unpublished lecture notes for the NATO Summer School, Copenhagen.

——— (1972), *Varieties of programming languages*, Proc. Internat. Computing Symp., Cini Foundation, Venice, pp. 222–233. [Also: Tech. Monograph PRG-10, Oxford Univ. Computing Lab., Programming Research Group (1972).]

C. STRACHEY AND C. WADSWORTH (1974), *Continuations: A mathematical semantics for handling full jumps*, Tech. Monograph PRG-11, Oxford Univ. Computing Lab., Programming Research Group.

A. TANG (1974), *Recursion theory and descriptive set theory in effectively given T_0 spaces*, Ph.D. thesis, Princeton Univ., Princeton, N.J.

——— (1975a), *Borel sets in Pω*, IRIA-Laboria, preprint.

——— (1975b), *Notes on subsets on Pω with extra-finitary property*, IRIA, preprint.

——— (1975c), *Sets of the form $\{x|R(x) = T\}$ in Pω*, IRIA, preprint.

——— (1975d), $\underset{e}{\leqq}$ *Degrees in P*, IRIA, preprint.

——— (1975e), *A hierarchy of B_δ sets in Pω*, IRIA, preprint.

A. TARSKI (1930), *On some Fundamental Concepts of Mathematics* (1930); translated by J. H. Woodger in Logic, Semantics, Metamathematics, Cambridge, 1956, pp. 30–37.

——— (1955), *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math., 5, pp. 285–309.

R. D. TENNENT (1973), *Mathematical semantics and design of programming languages*, Ph.D. thesis, Univ. of Toronto. [Also: Tech. Rep. 59, Univ. of Toronto (1973).]

——— (1974a), *The mathematical semantics of programming languages*, Dept. of Computing and Information Sci., Queen's Univ., Kingston, Ontario, Canada (preprint).

——— (1974b), *A contribution to the development of PASCAL-like languages*, Tech. Rep. 74-25, Dept. of Computing and Information Sci., Queen's Univ., Kingston, Ontario, Canada.

——— (1975), *PASQUAL: A proposed generalization of PASCAL*, Tech. Rep. 75-32, Dept. of Computing and Information Sci., Queen's Univ., Kingston, Ontario, Canada.

J. VUILLEMIN (1973a), *Proof techniques for recursive programs*, IRIA—Laboria Rep.

——— (1973b), *Correct and optimal implementations of recursion in a simple programming language*, IRIA—Laboria Rep. 24.

C. P. WADSWORTH (1971), *Semantics and pragmatices of the lambda-calculus*, D. Phil. thesis, Oxford Univ., Oxford, England.

——— (1975a), *The relation between lambda-expressions and their denotation*, Dept. of Systems and Information Sci., Syracuse Univ., Syracuse, N.Y. (preprint).

——— (1975b), *Approximate reduction and lambda-calculus models*, Dept. of Systems and Information Sci., Syracuse Univ., Syracuse, N.Y.

——— (1975c), Some unusual λ-calculus numeral systems, in preparation.

——— (to appear), *On the topological ordering in the D-model of the lambda-calculus*, in preparation.

E. G. WAGNER (1971a), *Languages for defining sets in arbitrary algebras*, Proc. of the 11th IEEE Conf. on Switching and Automata Theory, pp. 192–201.

——— (1971b), *An algebraic theory of recursive definitions and recursive languages*, Proc. of the 3rd ACM Symp. on Theory of Computing.

——— (1974), *Notes on categories, algebras and programming languages*, Unpublished lecture notes, London.

M. WAND (1973), *Mathematical foundations of formal language theory*, MAC-TR-108, Project MAC, Mass. Inst. of Tech., Cambridge, Mass.

——— (1974a), *An algebraic formulation of the Chomsky hierarchy*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 216–221.

——— (1974b), *On the recursive specification of data types*, Category Theory Applied to Computation and Control, E. Manes, ed., Univ. of Mass., pp. 222–225.

——— (1974c), *Realizing data structures as lattices*, Tech. Rep. 11, Computer Sci. Dept., Indiana Univ.

——— (1975), *Fixed-point constructions in order-enriched categories*, Tech. Rep. 23, Computer Sci. Dept., Indiana Univ.

P. WEGNER AND D. LEHMANN (1972), *Algebraic and topological prerequisites to Scott's theory of computation*, Tech. Rep. 72-2, Dept. of Computer Sci., Hebrew Univ. of Jerusalem.

R. W. WEYRAUCH AND R. MILNER (1972), *Program correctness in a mechanized logic.* Proc. of the 1st USA–JAPAN Computer Conf., pp. 384–390.

J. B. WRIGHT (1972), *Characterization of recursively enumerable sets*, J. Symbolic Logic, 37, pp. 507–511.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

THEORY OF OUTPUT SET ASSIGNMENTS AND DEGREE SWITCHING OPERATIONS*

BHARAT KINARIWALA AND KABEKODE V. S. BHAT†

Abstract. In this paper we study the basic aspects of the output set assignment problem using graph theory. The output sets of an $n \times n$ matrix A constitute sets of disjoint cycles of total length n in the network (i.e., a graph having weighted directed edges including self-loops) of matrix A. We introduce two new graphical transformations, viz., the degree switching operations (DSO) in a network and study their properties. The DSO yield output sets and lead to all graphs that can be associated with a given matrix A. Possible applications for the theory are indicated.

Key words. output set assignment, systems of equations, matrices, graphs, networks, degree switching operations, decomposition

1. Introduction. Large sparse systems of linear equations occur in a variety of frequently encountered problems such as linear programming, circuit analysis, structural analysis and numerical solution of partial differential equations [1]. The set of nonzero elements which could be placed on the diagonal by a reordering of the equations is called an output set, and there may, of course, be a large number of possible output sets. The identification of an output set is central to the efficient solution of matrix equations. Equally useful are the ways of determining other output sets from a given output set. Different output sets usually lead to different structural graphs corresponding to the same set of equations, and these structural graphs have a direct relation to the efficiency of the corresponding elimination routine.

Besides the use in the solution to systems of linear equations and matrix inversion problems [2], [3], the output set assignment problem has applications in large scale decomposition problems arising in several disciplines [4], [5], [6] and in graph theoretic methods for partitioning and tearing of systems [7], [8], [9].

Several authors [6], [10]–[17] have provided ad hoc algorithms for the determination of an output set. A well-known method for output set assignment is due to Steward [10]. An algorithm which is computationally more efficient than Steward's method is given by Patel and Kinariwala [5]. The algorithm of Hopcroft and Karp [18] which is the best known algorithm for bipartite matching in a computational complexity sense can also be used for finding an output set. However, these works are inadequate to explain important questions concerning the characterization and implications of different output sets for a given problem. In this paper we study the basic aspects of the output set assignment problem using graph theory [19]. We examine the combinatoric properties that govern the existence and identification of such sets for a given $n \times n$ matrix A and develop the necessary theory for obtaining different graphs that can be associated with the given matrix A.

* Received by the editors December 18, 1974, and in revised form August 26, 1975.

† Department of Electrical Engineering, University of Hawaii, Honolulu, Hawaii 96822. This research was supported by the National Science Foundation under Grant ENG74-19788.

A network of a square matrix A is a graph having directed, weighted edges that may include self-loops. It is shown that output sets for an $n \times n$ matrix A constitute sets of disjoint cycles of total length n in the network of the matrix. We introduce the important concept of degree switching operation (DSO) in a network with n vertices. The DSO is a graphical transformation and leads to a network with n self-loops. We show that the DSO yields output sets and leads to all graphs that can be associated with a given matrix A.

In § 2 we briefly review the pertinent concepts from network theory and indicate the correspondence between a matrix and a network. In § 3 we present certain properties of permutation matrices that are of interest to us in the later sections and provide a characterization for a given network using the set of disjoint cycles of total length n and other edges in the network. In § 4 we define two new operations, namely, the indegree switching and the outdegree switching operations in a network. These operations correspond to specific permutations on the rows or columns of a matrix associated with the given network. Some of the interesting and useful properties of the newly defined DSO are given in this section. An example is presented to illustrate the DSO in a network and some of its properties.

In § 5 we examine the output set assignment problem. We consider the combinatoric output set assignment problem and present a network theoretic interpretation for the same. Thus once the set of disjoint cycles of length n in the assignment network is identified with an output set, the problem of forcing an output set along the diagonal of a resulting matrix simply corresponds to a DSO in the assignment network. The complexity involved in forcing the elements of an output set along the diagonal of the resulting matrix is shown to be directly related to the number of disjoint cycles which are associated with the chosen output set. The DSO is shown to provide a systematic method for obtaining the network associated with any output set.

The theory developed in this paper providing an understanding of the output set assignment problem will be used for developing an efficient algorithm for the determination of an output set for a given square matrix. The notion of degree switching transformation can be used in tackling the resource allocation problems.

2. Preliminaries. A *network* N is a pair (V, E), where V is a finite nonempty set of $|V|$ vertices and $E \subseteq \{(v_i, v_j) | v_i, v_j \in V\}$ is the set of directed edges, with each edge (v_i, v_j) being associated with a nonzero element a_{ij} (which can be a constant or a pair) termed as the *weight* of (v_i, v_j). A network that is embedded in N is a *subnetwork* of N. Let $N_1 = (V_1, E_1)$ and $N_2 = (V_2, E_2)$ be two subnetworks of N. Then the network $(N_1 \cup N_2)$ is given by $(V_1 \cup V_2, E_1 \cup E_2)$. For a network $N = (V, E)$, an *ordering* α of V is a one-to-one mapping

$$\alpha: \{1, 2, \cdots, |V|\} \leftrightarrow V,$$

and $N_\alpha = (V, E, \alpha)$ is an *ordered network* associated with N. Two ordered networks N_α and N_β are said to be *isomorphic* if one can be obtained from the other by a reordering of the vertices of one of the networks. Given $v_i \in V$, the set adj $(v_i) = \{v_j | (v_i, v_j) \in E\}$ is the set of vertices adjacent to v_i, and the set of directed

edges $\{(v_i, v_j)|v_j \in \text{adj}(v_i)\}$ is called the *outbundle* of v_i. The set $\text{iadj}(v_i) = \{v_j|(v_j, v_i) \in E\}$ is the set of vertices to which v_i is adjacent, and the set of directed edges $\{(v_j, v_i)|v_j \in \text{iadj}(v_i)\}$ is called the *inbundle* of the vertex v_i. The cardinality of inbundle (outbundle) of vertex v_i is called the *indegree* (*outdegree*) of v_i.

For $v_s, v_t \in V$, a *directed path* $v_s \to v_t$ *of length* l is a sequence of distinct vertices $\{k_1, k_2, \cdots, k_{l+1}\}$ (with the possible exception of k_1 and k_{l+1}) such that $k_1 = v_s$, $k_{l+1} = v_t$ and $k_i = \text{iadj}(k_{i+1})$, $i = 1, 2, \cdots, l$. If there exists a directed path between each and every pair of vertices in a network, the network is said to be *strongly connected.*

For $v_i \in V$, a directed path $v_i \to v_i$ is a *cycle.* Two cycles in a network are said to be *disjoint* if they do not have any common vertices. A set of cycles in a network is said to be a set of disjoint cycles if each and every pair of cycles in the set are disjoint. The number of edges in a set of disjoint cycles gives the total length of the set of disjoint cycles. A cycle of length one is called a *loop.* There can be at most n loops in a network with n vertices. The weight of a set of cycles is given by the set of weights of the directed edges in the set of cycles.

An ordered network $N(A) = (V, E, \alpha)$, $\alpha = \{i\}_{j=1}^{n}$, can be associated with an $n \times n$ matrix A as follows: to the row i of matrix A, the vertex v_i of $N(A)$ is assigned. A directed edge (v_i, v_j) with weight a_{ij} exists in E if and only if $a_{ij} \neq 0$. Note that there is a one-to-one correspondence between the matrix A and $N(A)$. An alternative way of associating an ordered network $\tilde{N}(A) = (V, \tilde{E}, \alpha)$ is to assign column i of matrix A to the vertex $v_i \in \tilde{N}(A)$. A directed edge (v_j, v_i) with weight a_{ij} exists in $\tilde{N}(A)$ if and only if $a_{ij} \neq 0$. Note that $\tilde{N}(A)$ is identical to $N(A^t)$, where A^t is the transpose of matrix A.

An adjacency matrix of a network $N(A)$ is an $n \times n$ matrix whose (i, j)th element is 1 if $(v_i, v_j) \in E$ and is 0 if $(v_i, v_j) \notin E$. A network G is called a *digraph* if there are no loops in G and each edge of G has weight equal to one.

3. Permutation matrices, set of disjoint cycles, and network characterization. A *permutation matrix* P *of order* n is an $n \times n$ matrix, in each row and each column of which there is exactly one nonzero element, the value of this element being one. Permutation matrices are orthonormal matrices; i.e.,

$$p \cdot p^t = I, \tag{1}$$

where I is an $n \times n$ identity matrix. On a premultiplication of an $n \times n$ matrix A by P, the rows of A get permuted; i.e., if $p_{ij} = 1$ for some i, j, then the row i of the resulting matrix, $\hat{A} = P \cdot A$, is the same as the row j of the matrix A. A postmultiplication of the matrix A, by P, implies a permutation of the columns of A. The product of any two $n \times n$ permutation matrices is also a permutation matrix.

Let $N(P)$ be the network associated with an $n \times n$ permutation matrix P. Then the following can be stated.

PROPOSITION 1. *$N(P)$ constitutes a set of disjoint cycles of total length n.*

Proof. Since each row and each column of P has exactly one nonzero element, there is exactly one directed edge in the inbundle and in the outbundle of every vertex of $N(P)$. Each vertex is in a cycle. No vertex belongs to more than one cycle.

Thus $N(P)$ constitutes a set of disjoint cycles. $N(P)$ must have n directed edges since P has exactly n nonzero elements. The total length of the set of disjoint cycles is exactly n.

It must be noted that the adjacency matrix of a set of disjoint cycles of total length n in a network is a permutation matrix of the order n.

We define an $n \times n$ *elementary permutation matrix* (denoted by EPM) as an $n \times n$ permutation matrix, which has at most one cycle of length greater than one, in its associated network. Thus if M is an EPM, then all the off-diagonal entries of M correspond to a single cycle in $N(M)$. With the definition of an EPM, we have a convenient procedure for a canonic factorization of any permutation matrix as follows.

PROPOSITION 2. *Any $n \times n$ permutation matrix P having k cycles in the associated network $N(P)$ can be expressed as the product of k* EPM*'s.*

Proof. Assume that the cycles of $N(P)$ have been ordered in some fashion. Let l_i be the length of cycle i. Then M_i, the ith EPM is obtained by the following procedure:

1. Except for those 1's which correspond to the ith cycle in $N(P)$, replace all 1's of P by zeros and obtain a matrix M_i.
2. For $j = 1, 2, 3, \cdots, n$, if the row j of M_i does not contain a 1, then set $m_{jj} = 1$.

Once $(M_1, M_2, M_3, \cdots, M_k)$ are found by the above procedure, the following relationship, viz.

$$\prod_{i=1}^{k} M_i = P, \tag{2}$$

is immediate in view of the rules for direct product of matrices.

An $n \times n$ *transposition* is an EPM which has a longest cycle of length 2 in its associated set of cycles. On a repeated premultiplication or postmultiplications by an appropriate sequence of transpositions, any EPM can be transformed into an identity matrix. The minimum number of such transpositions needed is of some interest and is given in the following.

PROPOSITION 3. *The minimum number of transpositions needed to reduce an* EPM *having the longest cycle of length l in its associated network to an identity matrix is $(l-1)$.*

Next we present a decomposition for a given network which would be useful in the next section. We begin with a few definitions.

DEFINITION 1. Let C_1 and C_2 be two sets of disjoint cycles. C_1 and C_2 are *equivalent* if the set of directed edges of C_1 are the same as those in C_2. Otherwise, C_1 and C_2 are *distinct.*

DEFINITION 2. A cycle R_1 which does not belong to any set of disjoint cycles of total length n in a network having n vertices is a *residual cycle.*

DEFINITION 3. The set of edges of a network which does not appear in any cycles of the network is defined to be a set of *residual edges of the network.*

With these definitions, we have the following proposition, which serves to characterize any given network.

PROPOSITION 4. *Any network $N(A)$ having n vertices can be characterized by the following equality, viz.*

$$N(A) = \bigcup_{i=1}^{s} C_i \bigcup_{j=1}^{t} R_j \cup Q, \tag{3}$$

where $\{C_i\}_{i=1}^{s}$ corresponds to all possible distinct sets of disjoint cycles of total length n. $\{R_i\}_{i=1}^{t}$ corresponds to the set of residual cycles in $N(A)$ and Q corresponds to the set of residual edges of $N(A)$.

Proof. An edge in $N(A)$ can either be in a cycle of $N(A)$ or in the set of residual edges, not both. Thus $N(A)$ can be decomposed into a network of residual edges and a network of cycles. A cycle can either be in a set of cycles of total length n or in a residual cycle but not in both. Thus all cycles can be partitioned uniquely into the class of set of disjoint cycles of total length n and a set of residual cycles. Thus the decomposition of the network as given by (3) is valid.

4. The degree switching operations in a network and their properties. Let C be a set of disjoint cycles of total length n in a network $N(A) = (V, E, \alpha)$ where $|V| = n$. Let P denote the adjacency matrix associated with the subnetwork C. Note that P is an $n \times n$ permutation matrix. We define the following new graphical transformations in $N(A)$.

DEFINITION 4. A network $N(\hat{A})$ is said to have *resulted from an outdegree switching operation on $N(A)$ about C* if and only if

$$\hat{A} = P^t \cdot A, \tag{4}$$

where $\hat{A}$ is the matrix corresponding to $N(\hat{A})$.

DEFINITION 5. A network $N(\tilde{A})$ is said to have *resulted from an indegree switching operation on $N(A)$ about C* if and only if

$$\tilde{A} = A \cdot P^t, \tag{5}$$

where $\tilde{A}$ is the matrix corresponding to $N(\tilde{A})$.

The matrix transformations of the type 4 and 5 are respectively defined as the outdegree switching and indegree switching transformations.

The graphical transformation performed by the degree switching operations in a network $N(A)$ about C is explained next. For this purpose, we assume that the disjoint cycles of the set C are ordered in some fashion; i.e., $C = \{c_j\}_{j=1}^{k}$, where c_j is a cycle of length l_j defined by the sequence of vertices $\{v_{j_1}, v_{j_2}, v_{j_3}, \cdots, v_{j_{l_j}}, v_{j_1}\}$. Then the outdegree switching operation in $N(A)$ about C switches the outbundles of the vertices $\{v_{j_1}, v_{j_2}, v_{j_3}, \cdots, v_{j_{l_j}}\}$, respectively, over to the vertices $\{v_{j_2}, v_{j_3}, \cdots, v_{j_{l_j}}, v_{j_1}\}$ for $j = 1, 2, \cdots, k$. The indegree switching operation in $N(A)$ about C switches the inbundles of the vertices $\{v_{j_2}, v_{j_3}, \cdots, v_{j_{l_j}}, v_{j_1}\}$, respectively, over to the vertices $\{v_{j_1}, v_{j_2}, \cdots, v_{j_{l_j}}\}$, for $j = 1, 2, \cdots, k$. The networks resulting out of these transformations would be $N(\hat{A})$ and $N(\tilde{A})$, respectively.

One important consequence of a degree switching operation on $N(A)$ is that, in the resulting network, there will be exactly n loops. This is because the n edges of the set C about which the degree switching operation is performed would become loops in the resulting network.

Next we consider some of the important properties of the newly defined degree switching operations. Since some of these properties are independent of C, the set of disjoint cycles of total length n about which an out(in)degree switching is performed in $N(A)$, C will not be specified. Further, since the weights of the edges do not influence the properties to be discussed, they are assumed to be unity. With these remarks, we present the following results.

THEOREM 1. *On an out(in)degree switching operation, in a network $N(A)$, a set of disjoint cycles of total length n in $N(A)$ remains as a set of disjoint cycles of total length n in the resulting network $N(\hat{A})(N(\tilde{A}))$.*

Proof. We prove the theorem for an outdegree switching operation. The proof for an indegree switching operation is similar. Analogous to the network decomposition equation (3), we have the following Boolean matrix equality for the Boolean A matrix:

$$A = \sum_{i=1}^{s} P_i \oplus \sum_{j=1}^{t} M_j \oplus R \tag{6}$$

where $\{P_i\}_{i=1}^{s}$ is the set of distinct $n \times n$ permutation matrices associated with the set of disjoint cycles of total length n, $\{M_j\}_{j=1}^{t}$ is the set of Boolean matrices (corresponding to the residual cycles) and R corresponds to the residual edges in $N(A)$. The "$\oplus$" operation is the Boolean "inclusive or" operation and $\sum_{i=1}^{s}$ stands for $\oplus$ operation for the s matrix operands.

An outdegree switching operation on $N(A)$ is characterized by a premultiplication of A by an $n \times n$ permutation matrix, say P_k^t, where, $P_k \in \{P_i\}_{i=1}^{s}$, i.e., $\hat{A} = P_k^t \cdot A$. Thus

$$\sum_{i=1}^{s} P_k^t \cdot P_i \oplus \sum_{j=1}^{t} P_k^t \cdot M_j \oplus P_k^t \cdot R = \hat{A}. \tag{7}$$

Since the operation $P_k^t \cdot P_i$ necessarily leads to an $n \times n$ permutation matrix, it follows that all sets of disjoint cycles of total length n in $N(A)$ (corresponding to P_i's) remain as sets of disjoint cycles of total length n (corresponding to $P_k^t \cdot P_i$) in $N(\hat{A})$.

However, it must be noted that the exact cycle structure of each set of disjoint cycles of total length n in $N(\hat{A})$ may be different from the one in $N(A)$. The next property concerns invariancy of the number of sets of disjoint cycles of total length n.

THEOREM 2. *The number of sets of disjoint cycles of total length n in the network $N(A)$ is invariant under degree switching operations.*

Proof. Given a matrix A, each distinct set of disjoint cycles of total length n can be coordinated with a nonzero term in the permanent expansion for the matrix A. Thus the number of sets of disjoint cycles of total length n in $N(A)$ is given by the number of nonzero terms in the permanent expansion for A. From determinant theory, it is known that the determinant expansion is invariant with respect to

the permutation of rows or columns of the matrix, assuming that the sign of each term is disregarded. Since the number of nonzero terms in the permanent expansion is invariant with respect to permutations of rows or columns, the theorem is true.

THEOREM 3. *Edges of the residual cycles which do not belong to any set of disjoint cycles of total length n do not remain in any cycle after a degree switching operation.*

Proof. It is true that each and every cycle in the resulting network must belong to at least one set of disjoint cycles of total length n in the resulting network. The set of edges which constitute all sets of disjoint cycles of total length n in the network and the resulting network are the same. Thus if any of the edges of the residual cycles which do not appear in any of the sets of disjoint cycles of total length n in the network were to appear in a cycle in the resulting network, then this would contradict the invariance property of the number of sets of disjoint cycles of total length n. If this were not to happen, the theorem must be true.

A direct consequence of Theorem 3 is the following corollary.

COROLLARY 1. *The network resulting on a degree switching operation on a strongly connected network having residual cycles may not be strongly connected.*

Proof. If the network has a residual cycle having edges not belonging to any set of disjoint cycles of total length n, then in the resulting network on degree switching there are edges which do not belong to any cycles. Existence of such edges destroys the strong connectivity property of the resulting network.

The corollary explains the need for a degree switching transformation on an $n \times n$ matrix A in order to exploit the bireducibility of the matrix A, in the large scale matrix inversion problem [3].

If the resulting network $N(\hat{A})$ were strongly connected, then necessarily $N(A)$ must be strongly connected. This can be shown by contradiction. Assume $N(A)$ is not strong; then either $N(A)$ is disconnected or $N(A)$ has some residual edges. In both cases, the resulting network $N(\hat{A})$ cannot be strong, which is a contradiction.

The next property concerns the relation between the indegree switching and outdegree switching operations.

THEOREM 4. $N(\tilde{A})$ *is isomorphic to* $N(\hat{A})$.

Proof. The outdegree switching on $N(A)$ about C is characterized by the equation

$$\hat{A} = P^t \cdot A. \tag{8}$$

The indegree switching operation on $N(A)$ about C is characterized by the equation

$$\tilde{A} = A \cdot P^t. \tag{9}$$

From (8) and (9) we get

$$\tilde{A} \cdot P = P \cdot \hat{A}, \tag{10}$$

i.e.,

$$\tilde{A} = P \cdot \hat{A} \cdot P^t. \tag{11}$$

Hence $N(\tilde{A})$ is isomorphic to $N(\hat{A})$.

Let $N(A)$ be a given network and $N(A_1)$ a network that resulted from an out(in)degree switching operation. If $N(A_1)$ has cycles, it is possible to perform a further degree switching operation on $N(A_1)$ to obtain a network $N(A_2)$. A sequence of k degree switching operations performed on $N(A)$, to obtain a network $N(A_k)$, is termed as a sequence of a degree switching operation.

Next we present an example to illustrate some of the concepts introduced in this section.

Example 1. Consider the network $N(A)$ with 5 nodes in Fig. 1. The associated matrix A is given in Fig. 2. There are two distinct sets of disjoint cycles of total length 5. These are $C_1 = (1, 2, 3, 4, 5, 1)$ and $C_2 = ((1, 1), (2, 2), (3, 4, 5, 3))$. Cycle $R_1 = (1, 2, 3, 1)$ is the residual cycle since this cycle does not belong to C_1 or C_2.

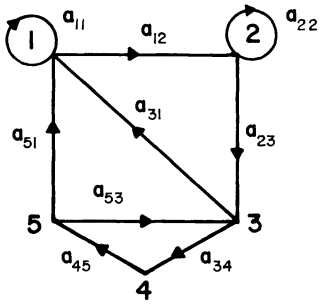


FIG. 1. *Network* $N(A)$

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 \\ a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & 0 & 0 & 0 & a_{45} \\ a_{51} & 0 & a_{53} & 0 & 0 \end{bmatrix}$$

FIG. 2 *Matrix* A

Permutation matrix P_1 is obtained from A by forcing all the elements of the Booleanized A matrix to zero except those 1's corresponding to the directed edges in C_1. Thus we have

$$P_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

An outdegree switching operation about C_1, characterized by the transformation $\hat{A} = P_1^t \cdot A$, leads to a new network $N(\hat{A})$ as shown in Fig. 3. Matrix $\hat{A}$ is shown in Fig. 4.

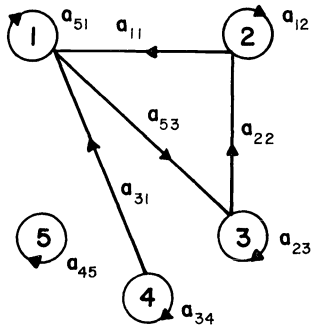


FIG. 3. *Network* $N(\hat{A})$ *obtained on an out-degree switching operation on* $N(A)$ *about* C_1

$$\hat{A} = \begin{bmatrix} a_{51} & 0 & a_{53} & 0 & 0 \\ a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 \\ a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & 0 & 0 & 0 & a_{45} \end{bmatrix}$$

FIG. 4. *Matrix* $\hat{A}$

It can be seen that the directed edges of the set of disjoint cycles of total length 5 remain as edges in a set of disjoint cycles of total length 5. However, the edge with weight a_{31} of $N(A)$ becomes a residual edge in $N(\hat{A})$. As can be seen from Figs. 3 and 1, on an outdegree switching operation on $N(A)$ about C_1, the outbundles of a vertex in C_1 are switched over to another vertex of the same cycle, which is nearest to it when traversed along the cycle. This switching of outbundles occurs for all vertices of C_1.

The indegree switching operation about C_1 is characterized by the transformation $\tilde{A} = A \cdot P_1^t$. The resulting network $N(\tilde{A})$ and the associated matrix $\tilde{A}$ are shown in Figs. 5 and 6, respectively.

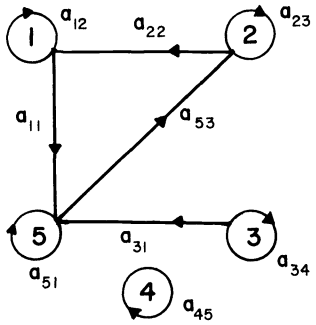


FIG. 5. *Network* $N(\tilde{A})$

$$\tilde{A} = \begin{bmatrix} a_{12} & 0 & 0 & 0 & a_{11} \\ a_{22} & a_{23} & 0 & 0 & 0 \\ 0 & 0 & a_{34} & 0 & a_{31} \\ 0 & 0 & 0 & a_{45} & 0 \\ 0 & a_{53} & 0 & 0 & a_{51} \end{bmatrix}$$

FIG. 6. *Matrix* $\tilde{A}$

It can be seen that in this case also, the edges in the set of disjoint cycles of total length 5 remain in a set of disjoint cycles of total length 5 in network $N(\tilde{A})$. The edge with weight a_{31} (which is an edge of the residual cycle R_1 not belonging to C_1 or C_2) does not remain in any cycle of $N(\tilde{A})$.

Note that the network $N(\hat{A})$ of Fig. 3 and the network $N(\tilde{A})$ of Fig. 5 are isomorphic since a node ordering (5, 1, 2, 3, 4) of network $N(\tilde{A})$ corresponds to the node ordering (1, 2, 3, 4, 5) of network $N(\hat{A})$.

5. The output set assignment problem. Given a nonsingular $n \times n$ matrix M, an output set assignment problem involves the permutation of the rows and columns of M, so that in the resulting matrix, the elements along the diagonal are all nonzeros. Since the problem involves only permutations, we need to be given only the (zero nonzero) structure for the matrix M. Let B be a matrix defined as follows:

$$b_{ij} = \begin{cases} 0 & \text{if } m_{ij} = 0, \\ (i, j) & \text{otherwise,} \quad i, j = 1, 2, \cdots, n. \end{cases}$$

The matrix B will be referred to as the assignment matrix associated with the matrix M. It provides the necessary structure for M in a convenient form. Then we define an output set assignment problem for a given matrix M as one of finding a pair of $n \times n$ permutation matrices, if one exists, such that

$$\hat{B} = P \cdot B \cdot Q, \tag{12}$$

where all the elements along the diagonal of the matrix $\hat{B}$ are nonzero. The set of these n nonzero diagonal elements of $\hat{B}$ is termed as an *output set for the matrix M.*

There may exist a set of (P, Q) pairs each of them leading to an output set. Output sets S_1 and S_2 are said to be *distinct* if and only if $S_1 \neq S_2$. If S_1 and S_2 are not distinct, then they are said to be *equivalent.*

It is well known (see, for example, [16]), that there exists at least one output set for a given $n \times n$ nonsingular matrix M. Concerning the number of distinct output sets for the given matrix M, we have the following.

PROPOSITION 5. *The number of distinct sets of disjoint cycles of total length n in $N(B)$ gives the number of distinct output sets for M.*

Proof. The number of distinct output sets for the matrix M is given by the number of distinct $n \times n$ permutation matrices that can be embedded in the Booleanized version of matrix M (i.e., the matrix obtained by replacing all nonzeros of M by 1's). The result follows in view of Proposition 1.

Thus in order to enumerate all possible output sets for the $n \times n$ matrix M we need to do the following:

1. Enumerate all directed cycles of the network $N(B)$, where B is the assignment matrix corresponding to M.
2. Find a set of disjoint cycles of total length n in $N(B)$. Include the weight of the set in the set of distinct output sets.
3. If all the output sets are found, stop; otherwise find a new output set, on branching to step 2.

A brief discussion on the complexity of the above method in the context of edge sparse $N(B)$ is pertinent. Step 1 can be effectively handled by the algorithm given by Johnson [20]. Step 2 can be implemented by enumerating all maximal cliques of a derived graph $G^* = (V^*, E^*)$, where each vertex $v_i^* \in V^*$ corresponds to exactly one of the elementary cycles c_i^* of $N(B)$ and (undirected) edge $[v_i^*, v_j^*] \in E^*$ whenever cycles c_i^* and c_j^* are disjoint. An effective algorithm given by Akkoyunlu [21] can be used for enumeration of all maximal cliques of the large graph G^*. For sparse matrix problems where major preprocessing is required before solving a class of problems with the same structure, the method given above can be meaningful.

Up to now, we have considered the problem of identifying the output set elements given an $n \times n$ matrix. The next important question pertains to forcing the elements of a chosen output set along the diagonal of a resulting matrix, and the number of row or column interchanges required to achieve this. We begin with the following.

THEOREM 5. *There exists row-only or column-only permutations to force an output set element along the diagonal position of the resulting matrix.*

Proof. Let B be the assignment matrix. Then

$$B = B_\varphi + B_R, \tag{13}$$

where the nonzero elements of B_φ are the elements of the output set φ. B_φ can be written as follows:

$$B_\varphi = P_\varphi \psi, \tag{14}$$

where P_φ is an $n \times n$ permutation matrix and ψ is an $n \times n$ diagonal matrix whose nonzero entries constitute the output set φ. On premultiplying (13) by P_φ^t and

simplifying, we get

$$P_\varphi^t B = \psi + P_\varphi^t B_R. \tag{15}$$

The above equation implies the existence of a row-only permutation which forces the elements of φ along the diagonals of the resulting matrix $P_\varphi^t B$. Similarly, by postmultiplication of (13) by P_φ^t, we get

$$B \cdot P_\varphi^t = P_\varphi \cdot \psi \cdot P_\varphi^t + B_R \cdot P_\varphi^t. \tag{16}$$

Note that $P_\varphi \cdot \psi \cdot P_\varphi^t$ is a diagonal matrix equivalent to ψ. This implies that a column-only permutation can force the output set elements along the diagonal of the resulting matrix.

It must be clear at this stage that the row alone permutation corresponds to an outdegree switching operation in the network $N(B)$ about the set of cycles associated with the permutation matrix P_φ. Similarly, the column-only permutation which forces the output set elements along the diagonal of the resulting matrix corresponds to an indegree switching transformation in $N(B)$ about a set of cycles associated with P_φ. Further, it is easily seen that the networks associated with $(B \cdot P_\varphi^t)$ and $(P_\varphi^t \cdot B)$ are isomorphic. In fact, with any permutations involving both rows and columns of B, which forces the elements of the output set along the diagonal of a resulting matrix, the network associated with the resulting matrix also is isomorphic to $N(B \cdot P_\varphi^t)$.

An interchange of rows (or columns) of a matrix is affected by a pre(post)multiplication of the matrix by a transposition. The minimum number of transpositions needed to force the output set along the diagonal of the resulting matrix is given in the following.

THEOREM 6. *The minimum number of transpositions needed to force an output set of M which constitutes k cycles in $N(M)$ along the diagonal of the resulting matrix is $n-k$.*

Proof. Let P^t be the permutation matrix associated with the degree switching operation. Then P^t can be expressed as follows:

$$P^t = \prod_{i=1}^{k} M_i,$$

where M_i is the EPM associated with ith cycle of $N(P^t)$. If the largest cycle associated with M_i is of length l_i, then from Proposition 3, we have that the minimum number of transpositions needed to force the elements along the diagonal of a resulting matrix is $(l_i - 1)$. Hence the required minimum number of transpositions is $\sum_{i=1}^{k} (l_i - 1)$, i.e., it is $n-k$.

Theorem 6 implies that the larger the number of cycles the output set constitutes in $N(M)$, the fewer transpositions are needed to force the elements of the output set along the diagonal of a resulting matrix. Thus if there is an output set along the diagonal, then no computational effort is needed. The theorem thus gives a criterion for choosing an optimal output set when more than one are available.

The next important problem is to provide a method to obtain the network associated with an alternate output set, given the network associated with an output set. This can be achieved in view of the following.

PROPOSITION 6. *Given a network $N(B)$ having n loops, the network corresponding to all other distinct output sets can be obtained by a sequence of degree switching operations on $N(B)$.*

The DSO in a network thus provides a systematic method for obtaining different networks which can be associated with an $n \times n$ matrix, depending upon the choice of output sets.

6. Conclusion. In this paper we have presented a theory needed to tackle important questions related to the output sets for a given $n \times n$ matrix. We have defined the degree switching operations in a network and studied their properties. The DSO are shown to yield output sets.

The theory developed in this paper, providing an understanding of the output set assignment problem, can be used for developing an algorithm for the determination of an output set for a square matrix. The concept of degree switching can be used for tackling more general resource allocation problems.

By extending the operations of degree switching to digraphs, a formal solution to the optimum elimination problem can be given. Other problems which can be tackled by our present understanding include optimum tearing procedures for the large scale systems and suboptimal algorithms for the elimination problem. These and other related topics will be the subject matter for our subsequent papers.

REFERENCES

[1] D. J. ROSE AND R. A. WILLOUGHBY, eds., *Sparse matrices and their applications*, Proc. IBM Conf. (Sept. 1970), Plenum Press, New York, 1972.

[2] R. K. BRAYTON, F. G. GUSTAVSON AND R. A. WILLOUGHBY, *Some results on sparse matrices*, Math. Comp., 24 (1969), pp. 937–954.

[3] A. L. DULMAGE AND N. S. MENDELSOHN, *On the inversion of sparse matrices*, Ibid., 16 (1962), pp. 494–496.

[4] D. M. HIMMELBLAU, *Morphology of decomposition*, Decomposition of Large Scale Systems, D. M. Himmelblau, ed., American Elsevier, New York, 1973, pp. 1–13.

[5] B. PATEL AND B. KINARIWALA, *On the decomposition of large scale systems*, Proc. 6th Hawaii Internat. Conf. on System Sciences, Western Periodicals, 1973, pp. 356–358.

[6] W. P. LEDET AND D. M. HIMMELBLAU, *Decomposition procedures for the solving of large scale systems*, Advances in Chem. Engrg., 8 (1970), pp. 186–254.

[7] D. V. STEWARD, *Partitioning and tearing of systems of equations*, SIAM J. Numer. Anal., 2 (1965), pp. 345–365.

[8] ———, *Tearing analysis of the structure of disorderly sparse matrices*, Sparse Matrices Proceedings, R. A. Willoughby, ed., Rep. RA1(11707), IBM, Yorktown Heights, N.Y., 1969, pp. 65–74.

[9] L. K. CHEUNG AND E. S. KUH, *A graph theoretic method for optimal partitioning of large sparse matrices*, Proc. 6th Hawaii Internat. Conf. on System Sciences, Western Periodicals, 1973, pp. 45–49.

[10] D. V. STEWARD, *On an approach to technique for the analysis of the structure of large systems of equations*, SIAM Rev., 4 (1962), pp. 321–342.

[11] R. L. WEIL AND D. V. STEWARD, *The question of determinancy in square systems of equations*, Z. Nationalökonome, 27 (1967), no. 3, pp. 261–266.

[12] A. L. DULMAGE AND N. S. MENDELSOHN, *Two algorithms for bipartite graphs*, SIAM J. Appl. Math., 11 (1963), pp. 183–194.

[13] L. R. FORD AND D. R. FULKERSEN, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1963, pp. 55–57.

[14] J. E. KALAN, *Aspects of large scale in core linear programming*, Proc. 1971 Ann. Conf. ACM (Aug. 1971), pp. 304–313.
[15] P. C. KETTLER AND R. C. WEIL, *An algorithm for providing structure for decomposition*, Sparse Matrix Proceedings, R. A. Willoughby, ed., Rep. RA1(11707), IBM, Yorktown Heights, N.Y., 1969, pp. 11–24.
[16] R. P. TEWARSON, *Sparse matrices*, Mathematics in Science and Engineering, vol. 99, Academic Press, New York, 1973, pp. 51–53.
[17] A. YASPAN, *On finding a maximal assignment*, Operations Res., 14 (1966), pp. 641–651.
[18] J. E. HOPCROFT AND R. M. KARP, *Maximum matchings in bipartite graphs*, this Journal, 2 (1973), pp. 225–231.
[19] F. HARARY, R. Z. NORMAN AND D. CARTWRIGHT, *Structural Models: An Introduction to the Theory of Directed Graphs*, John Wiley, New York, 1965.
[20] D. B. JOHNSON, *Finding all the elementary circuits of a directed graph*, this Journal, 4 (1975), pp. 77–84.
[21] E. A. AKKOYUNLU, *The enumeration of maximal cliques of large graphs*, this Journal, 2 (1973), pp. 1–6.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

RANKING ALGORITHMS FOR LISTS OF PARTITIONS*

S. G. WILLIAMSON†

Abstract. Given an algorithm for producing a list of objects, a "ranking" algorithm or "sequential numbering scheme" is a rule ρ which, given an object x, computes its position $m = \rho(x)$ in the list. The associated algorithm for computing ρ^{-1}, produces the object x given its position m. In this paper, we consider the objects to be various classes of partitions of a set S. We consider ranking algorithms for lists of all ordered partitions, ordered partitions corresponding to a given multinomial index, all unordered partitions, unordered partitions with a bounded number of blocks, unordered partitions with a fixed number of blocks, unordered partitions with blocks of equal size, unordered partitions corresponding to a fixed ordered partition, and unordered partitions of specified block type. These ranking algorithms are frequently useful for organizing computations associated with the above lists. The computation of ρ^{-1} at a randomly selected m provides a method for the random selection of elements from any of these lists.

Key words. ordered set partitions, unordered set partitions, ranking algorithms, lexicographic order.

1. Introduction. An *unordered partition* of a set S is a collection $\mathscr{C}$ of subsets of S satisfying the conditions

(i) the empty set $\varnothing \notin \mathscr{C}$,

(ii) $A, B \in \mathscr{C}$, $A \neq B$, then $A \cap B = \varnothing$,

(iii) the union of the subsets of $\mathscr{C}$ is S.

The subsets of S contained in the unordered partition $\mathscr{C}$ will be called the *blocks* of $\mathscr{C}$.

An *ordered partition* $\bar{\mathscr{C}}$ of S of *length* n is an n-tuple $(A_1, \cdots, A_n)$ of subsets of S satisfying

(i) if $i \neq j$ then $A_i \cap A_j = \varnothing$,

(ii) $\bigcup_{i=1}^{n} A_i = S$.

In an ordered partition some of the components A_i may be empty.

We are concerned with algorithms for listing in some specified order certain classes of ordered and unordered partitions of a set S. The basic feature that these algorithms must have is that they admit an easily computable "ranking algorithm." This means that given a number m we must have a simple algorithm for producing the mth element in the list being generated, and, conversely, given any element in the list we must be able to compute its corresponding m value. We do not attempt to be precise about "easily computable." This is left as a matter of practical judgment. For example, if the permutations of the set $\{0, 1, 2, 3, 4, 5\}$ are listed in lexicographic order, the 364th permutation to appear in the list will be $(3, 0, 1, 4, 5, 2)$. An easy method for computing this fact is described in [3]. A hard way to compute this information would be to list all of the permutations in lexicographic order and keep a count, stopping at the 364th.

Given a positive integer r we denote by $\mathbf{r}$ the set $\{0, 1, 2, \cdots, r-1\}$. The functions from $\mathbf{d}$ to $\mathbf{r}$ will be denoted by $\mathbf{r}^{\mathbf{d}}$. A function f in $\mathbf{r}^{\mathbf{d}}$ will be written as a

* Received by the editors December 30, 1974, and in revised form September 24, 1975.

† School of Mathematics, University of Minnesota, Minneapolis, Minnesota 55455. This research was supported by the National Science Foundation under Grant GJ43333.

d-tuple $(\alpha_{d-1}, \cdots, \alpha_0)$ where $f(i) = \alpha_{d-l-i}$. Given $f = (\alpha_{d-1}, \cdots, \alpha_0)$ and $g = (\beta_{d-1}, \cdots, \beta_0)$, we say f is *lexicographically less* than g if $\alpha_{d-1} < \beta_{d-1}$ or if $\alpha_{d-1} = \beta_{d-1}$ and $(\alpha_{d-2}, \cdots, \alpha_0)$ is *lexicographically less* than $(\beta_{d-2}, \cdots, \beta_0)$. This recursive definition defines a standard linear order which we shall call *lex order* on $\mathbf{r^d}$. We say that $(\alpha_{d-1}, \cdots, \alpha_0)$ is *colexicographically less* than $(\beta_{d-1}, \cdots, \beta_0)$ if $(\alpha_0, \cdots, \alpha_{d-1})$ is lexicographically less than $(\beta_0, \cdots, \beta_{d-1})$. We call the resulting linear order on $\mathbf{r^d}$ *colex order.*

Let P and Q be ordered sets and let $\eta : P \to Q$ be a bijection such that $x \leqq y$ if and only if $\eta(x) \leqq \eta(y)$. Such a bijection is called an *order isomorphism* between P and Q. The order isomorphism ρ between a linearly ordered set P with cardinality $|P| = p$ and the linearly ordered set $\{0, 1, \cdots, p-1\}$ is called the *ranking function* for P. If $x \in P$, then $\rho(x)$ is called the *rank* of x. The number $\rho(x)$ is expressed base 10 in this paper.

Given a function $f \in \mathbf{r^d}$, the unordered partition $\{f^{-1}(t) : t \in \text{image } f\}$ is called the *coimage* of f.

2. Ordered partitions. By associating with each f in $\mathbf{r^d}$ the ordered partition $(f^{-1}(0), \cdots, f^{-1}(r-1))$ we obtain a natural bijection η between $\mathbf{r^d}$ and $\mathscr{P}_r(d)$, the ordered partitions of $\mathbf{d}$ of length r. Let the order on $\mathbf{r^d}$ be lex order and let the order on $\mathscr{P}_r(d)$ be defined from lex order on $\mathbf{r^d}$ by means of η. Call this order *lex order on* $\mathscr{P}_r(d)$.

THEOREM 2.1. *Let* $(A_0, \cdots, A_{r-1})$ *be in* $\mathscr{P}_r(d)$ *and define* $b(k) = d - l - k$; *then*

$$\rho(A_0, \cdots, A_{r-1}) = \sum_{k=0}^{d-1} f(b) r^k,$$

where $f = \eta^{-1}(A_0, \cdots, A_{r-1})$. *Then* ρ *is the ranking function for* $\mathscr{P}_r(d)$ *in lex order.*

This result amounts to the simple observation that the functions $(\alpha_{d-1}, \cdots, \alpha_0)$ of $\mathbf{r^d}$ listed in lex order are the coefficients of the integers $0, 1, \cdots, (r^d - 1)$ expressed base r. The functions ρ and ρ^{-1} are computed in the usual manner for converting between base r and base 10.

The ranking of the following basic sublist of $\mathscr{P}_r(d)$ is more interesting. An ordered partition $(A_0, \cdots, A_{r-1})$ is said to *correspond to the multinomial index* $(a_0, \cdots, a_{r-1})$ if $|A_i| = a_i$ for $i = 0, 1, \cdots, r-1$. We now consider the listing and ranking of the partitions of $\mathscr{P}_r(d)$ corresponding to a given multinomial index $(a_0, \cdots, a_{r-1})$.

First we state two basic results. Consider the functions $(\alpha_{d-1}, \cdots, \alpha_0)$ in $\mathbf{r_{d-1}} \times \cdots \times \mathbf{r_0}$ when the r_i are positive integers. Suppose this set is linearly ordered by lex order (regard this set as a subset of $\mathbf{r^d}$ in lex order for $r = \max(r_{d-1}, \cdots, r_0)$). The first result is elementary but useful.

THEOREM 2.2. *Let* $(\alpha_{d-1}, \cdots, \alpha_0)$ *be in* $\mathbf{r_{d-1}} \times \cdots \times \mathbf{r_0}$ *and define*

$$\rho(\alpha_{d-1}, \cdots, \alpha_0) = \sum_{t=0}^{d-1} \alpha_t c_t,$$

where $c_t = r_{t-1} c_{t-1}$ *and* $c_0 = 1$. *Then* ρ *is the ranking function for the set* $\mathbf{r_{d-1}} \times \cdots \times \mathbf{r_0}$ *in lex order.*

We shall use this result in several instances below. Given m, the following algorithm produces the rank m function in the lex list of $\mathbf{r_{d-1}}\times\cdots\times\mathbf{r_0}$.

Step 1. $t\leftarrow d-1$, $x\leftarrow m$, $\alpha_j\leftarrow 0$ for $j=0,\cdots,d-1$.

Step 2. $\alpha_t\leftarrow\max\{z : zc_t\leqq x\}$.

Step 3. If $\alpha_t c_t = x$, then go to (4); else $x\leftarrow x-\alpha_t c_t$, $t\leftarrow t-1$ go to Step 2.

Step 4. Stop: $(\alpha_{d-1},\cdots,\alpha_0)$ is the rank m function of $\mathbf{r_{d-1}}\times\cdots\times\mathbf{r_0}$ in lex order.

The second result is due to D. H. Lehmer [3]. If $(\alpha_{d-1},\cdots,\alpha_0)$, $\alpha_{d-1}<\cdots<\alpha_0$, is a strictly increasing function in $\mathbf{r}^{\mathbf{d}}$, $d\leqq r$, then the correspondence $(\alpha_{d-1},\cdots,\alpha_0)\leftrightarrow\{\alpha_{d-1},\cdots,\alpha_0\}$ defines an obvious bijection between these functions and the subsets of $\mathbf{r}$ of size d. For example, (0, 1, 4) corresponds to the set {0, 1, 4}. Using this correspondence, we define lex order (or colex order) on subsets to be that order induced by lex (or colex order) on the strictly increasing functions. Assume from now on that *the elements in any subset* $\{\alpha_{d-1},\cdots,\alpha_0\}$ *of a linearly ordered set are written left to right in increasing order* $0\leqq\alpha_{d-1}<\alpha_{d-2}<\cdots<\alpha_0<r$. Let $\mathscr{C}_{d,r}$ denote the subsets of $\mathbf{r}$ of size d.

THEOREM 2.3 (Lehmer [3]). *Let* $\{\alpha_{d-1},\cdots,\alpha_0\}$ *be in* $\mathscr{C}_{d,r}$ *and define* $\rho\{\alpha_{d-1},\cdots,\alpha_0\}=\binom{\alpha_{d-1}}{1}+\cdots+\binom{\alpha_0}{d}$. *Then* ρ *is the ranking function for the set* $\mathscr{C}_{d,r}$ *in colex order. The function*

$$\rho\{\alpha_{d-1},\cdots,\alpha_0\}=\binom{r}{d}-1-\binom{\beta_{d-1}}{1}-\cdots-\binom{\beta_0}{d},$$

where $\beta_i=r-1-\alpha_{d-1-i}$, *is a ranking function for* $\mathscr{C}_{d,r}$ *in lex order.*

Given m, the following algorithm produces the rank m subset in colex order in $\mathscr{C}_{d,r}$.

Step 1. $t\leftarrow d-1$, $x\leftarrow m$, $\gamma_j\leftarrow j$ for $j=0,\cdots,d-1$.

Step 2. $\gamma t\leftarrow\max\{z : \binom{z}{t}\leqq x\}$.

Step 3. If $\binom{\gamma_t}{t}=x$, then go to (4); else $x\leftarrow x-\binom{\gamma_t}{t}$, $t\leftarrow t-1$, go to Step 2.

Step 4. Stop: $(\alpha_{d-1},\cdots,\alpha_0)$ where $\alpha_t=\gamma_{d-1-t}$ is the rank m subset in colex order on $\mathscr{C}_{d,r}$.

For example, the set $\mathscr{C}_{3,5}$ in colex order is 0 1 2, 0 1 3, 0 2 3, 1 2 3, 0 1 4, 0 2 4, 1 2 4, 0 3 4, 1 3 4, 2 3 4. The element 0 3 4 has rank $\binom{0}{1}+\binom{3}{2}+\binom{4}{3}=7$. We shall use colex order for listing subsets.

Given a multinomial index $(a_0,\cdots,a_{r-1})$, consider the functions in $\mathbf{b_0}\times\cdots\times\mathbf{b_{r-1}}$ where $b_0=\binom{d}{a_0}$ and $b_t=\binom{d-a_0-\cdots-a_{t-1}}{a_t}$ for $1\leqq t\leqq r-1$. Suppose this set is linearly ordered by lex order. Given a function $(\beta_0,\cdots,\beta_{r-1})$ in $\mathbf{b_0}\times\cdots\times\mathbf{b_{r-1}}$, define $\eta(\beta_0,\cdots,\beta_{r-1})=(A_0,\cdots,A_{r-1})$ where A_0 is the rank β_0 (colex order) subset of size a_0 of $\mathbf{d}$, A_1 is the rank β_1 subset of size a_1 of $\mathbf{d}-A_0$, and, in general, A_t is the rank β_t subset of size a_t of $\mathbf{d}-A_0-\cdots-A_{t-1}$. Define lex order on the ordered partitions of length r on $\mathbf{d}$ corresponding to the multinomial

index $(a_0, \cdots, a_{r-1})$ to be the order induced by applying η to $\mathbf{b_0} \times \cdots \times \mathbf{b_{r-1}}$ in lex order.

THEOREM 2.4. *Let* $(A_0, \cdots, A_{r-1})$ *be an ordered partition of length r on* $\mathbf{d}$ *corresponding to* $(a_0, \cdots, a_{r-1})$. *Define*

$$\rho(A_0, \cdots, A_{r-1}) = \sum_{t=0}^{r-1} c_t \xi(A_t),$$

where $c_{r-1} = 1$ *and* $c_t = \binom{d - \alpha_0 - \cdots - a_t}{a_{t+1}} c_{t+1}$ *for* $0 \leqq t \leqq r-2$ *and where* $\xi(A_t)$ *is the rank of* A_t *in colex order on subsets of size* a_t *of* $\mathbf{d} - A_0 - \cdots - A_{t-1}$. *Then* ρ *is a ranking function for the lex list of ordered partitions of length r on* $\mathbf{d}$ *corresponding to* $(a_0, \cdots, a_{r-1})$.

Given $m, \rho^{-1}(m)$ is computed by first applying the inversion algorithm following Theorem 2.2 and then applying the algorithm following Theorem 2.3 to each component of the resulting function.

As an example of Theorem 2.4, we find the rank of the ordered partition $(A_0, A_1, A_2, A_3) = (\{2, 5, 13\}, \{0, 3, 8, 11, 12\}, \{1, 10\}, \{4, 6, 7, 9\})$ on the lex list of ordered partitions of **14** corresponding to (3, 5, 2, 4). The rank of $\{2, 5, 13\}$ in the colex list of subsets of size 3 from **14** is (by Theorem 2.3) $\binom{2}{1} + \binom{5}{2} + \binom{13}{3} = 298$. The rank of $\{0, 3, 8, 11, 12\}$ in $\mathbf{14} - \{2, 5, 13\} = \{0, 1, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$ is $\binom{0}{1} + \binom{2}{2} + \binom{6}{3} + \binom{9}{4} + \binom{10}{5} = 399$ and the rank of $\{0, 10\}$ in $\{1, 4, 6, 7, 9, 10\}$ is $\binom{0}{1} + \binom{5}{2} = 10$. Thus $(\xi(A_0), \xi(A_1), \xi(A_2), \xi(A_3)) = (298, 399, 10, 0)$. We have $c_3 = 1, c_2 = \binom{4}{4} \cdot 1, c_1 = \binom{6}{2} \cdot 1 = 15$, $c_0 = \binom{11}{5} \cdot 15 = 6930$. Thus the rank of this ordered partition is $(298)(6930) + (399)(15) + (10)(1) + (0)(1) = 2{,}071{,}135$. Conversely, we find the rank 1,000,000 element in the above list: The rank 1,000,000 function in the lex list of functions in $\binom{\mathbf{14}}{\mathbf{3}} \times \binom{\mathbf{11}}{\mathbf{5}} \times \binom{\mathbf{6}}{\mathbf{2}} \times \binom{\mathbf{4}}{\mathbf{4}}$ is (144, 138, 10, 0). The rank 144 set of size 3 in colex order on **14** is $\{3, 7, 10\}$. The rank 138 subset of size 5 on $\mathbf{14} - \{3, 7, 10\}$ is $\{0, 4, 5, 6, 12\}$. Continuing in this manner, we obtain $\{\{3, 7, 10\}, \{0, 4, 5, 6, 12\}, \{1, 13\}, \{2, 8, 9, 11\}\}$.

3. Unordered partitions. We now consider unordered partitions, henceforth called simply "partitions." If $\{A_0, \cdots, A_p\}$ is a partition of $\mathbf{d}$, we shall adopt the convention that $0 \in A_0$ and the smallest integer not in $A_0 \cup \cdots \cup A_{t-1}$ is in A_t for $t = 1, \cdots, r-1$. *All partitions* will be assumed written in this order, which we call *standard order*. Given a partition $\{A_0, \cdots, A_p\}$, associate with it a function f defined by $f^{-1}(t) = A_t$ for $t = 0, \cdots, p$.

A function $f \in \mathbf{r}^{\mathbf{d}}$ will be called a function of *restricted growth* if $f(0) = 0$ and $f(t) \leqq (\max_{i<t} f(i)) + 1$ for $t = 1, \cdots, d-1$. Denote the set of such functions by $RG \subseteq \mathbf{r}^{\mathbf{d}}$.

THEOREM 3.1. *$f \in \mathbf{r}^{\mathbf{d}}$ is a function of restricted growth if and only if the blocks of the coimage are in standard order when written $\{f^{-1}(0), \cdots, f^{-1}(d-1)\}$.*

The result of Theorem 3.1 describes a bijection (f to coimage of f) between RG and the partitions of **d** with at most r blocks. We list $RG \subseteq \mathbf{r}^{\mathbf{d}}$ in lex order and define lex order on the partitions of **d** with at most r blocks to be the order defined by the above bijection. For $r = d = 5$, see Table 1. For algorithms based on this bijection see [2].

TABLE 1

ρ	f in $\mathbf{5}^{\mathbf{5}}$	Coimage of f	ρ	f in $\mathbf{5}^{\mathbf{5}}$	Coimage of f
0	0 0 0 0 0	(0, 1, 2, 3, 4)	26	0 1 1 0 1	(0, 3)(1, 2, 4)
1	0 0 0 0 1	(0, 1, 2, 3)(4)	27	0 1 1 0 2	(0, 7)(1, 2)(4)
2	0 0 0 1 0	(0, 1, 2, 4)(3)	28	0 1 1 1 0	(0, 4)(1, 2, 3)
3	0 0 0 1 1	(0, 1, 2)(3, 4)	29	0 1 1 1 1	(0)(1, 2, 3, 4)
4	0 0 0 1 2	(0, 1, 2)(3)(4)	30	0 1 1 1 2	(0)(1, 2, 3)(4)
5	0 0 1 0 0	(0, 1, 3, 4)(2)	31	0 1 1 2 0	(0, 4)(1, 2)(3)
6	0 0 1 0 1	(0, 1, 3)(2, 4)	32	0 1 1 2 1	(0)(1, 2, 4)(3)
7	0 0 1 0 2	(0, 1, 3)(2)(4)	33	0 1 1 2 2	(0)(1, 2)(3, 4)
8	0 0 1 1 0	(0, 1, 4)(2, 3)	34	0 1 1 2 3	(0)(1, 2)(3)(4)
9	0 0 1 1 1	(0, 1)(2, 3, 4)	35	0 1 2 0 0	(0, 3, 4)(1)(2)
10	0 0 1 1 2	(0, 1)(2, 3)(4)	36	0 1 2 0 1	(0, 3)(1, 4)(2)
11	0 0 1 2 0	(0, 1, 4)(2)(3)	37	0 1 2 0 2	(0, 3)(1)(2, 4)
12	0 0 1 2 1	(0, 1)(2, 4)(3)	38	0 1 2 0 3	(0, 3)(1)(2)(4)
13	0 0 1 2 2	(0, 1)(2)(3, 4)	39	0 1 2 1 0	(0, 4)(1, 3)(2)
14	0 0 1 2 3	(0, 1)(2)(3)(4)	40	0 1 2 1 1	(0)(1, 3, 4)(2)
15	0 1 0 0 0	(0, 2, 3, 4)(1)	41	0 1 2 1 2	(0)(1, 3)(2, 4)
16	0 1 0 0 1	(0, 2, 3)(1, 4)	42	0 1 2 1 3	(0)(1, 3)(2)(4)
17	0 1 0 0 2	(0, 2, 3)(1)(4)	43	0 1 2 2 0	(0, 4)(1)(2, 3)
18	0 1 0 1 0	(0, 2, 4)(1, 3)	44	0 1 2 2 1	(0)(1, 4)(2, 3)
19	0 1 0 1 1	(0, 2)(1, 3, 4)	45	0 1 2 2 2	(0)(1)(2, 3, 4)
20	0 1 0 1 2	(0, 2)(1, 3)(4)	46	0 1 2 2 3	(0)(1)(2, 3)(4)
21	0 1 0 2 0	(0, 2, 4)(1)(3)	47	0 1 2 3 0	(0, 4)(1)(2)(3)
22	0 1 0 2 1	(0, 2)(1, 4)(3)	48	0 1 2 3 1	(0)(1, 4)(2)(3)
23	0 1 0 2 2	(0, 2)(1)(3, 4)	49	0 1 2 3 2	(0)(1)(2, 4)(3)
24	0 1 0 2 3	(0, 2)(1)(3)(4)	50	0 1 2 3 3	(0)(1)(2)(3, 4)
25	0 1 1 0 0	(0, 3, 4)(1, 2)	51	0 1 2 3 4	(0)(1)(2)(3)(4)

We now describe the ranking function for the lex list of partitions of **d** with at most r blocks. For $n \geqq 0$, $m \geqq 0$, define $T^{(r)}(n, m)$ recursively:

(i) $T^{(r)}(0, m) = 1$ if $0 \leqq m \leqq r-1$ and $T^{(r)}(0, m) = 0$ if $m \geqq r$.

(ii) $T^{(r)}(n, m) = (m+1)T^{(r)}(n-1, m) + T^{(r)}(n-1, m+1)$.

Some of these numbers are given in Table 2. A combinatorial interpretation of these numbers is given in § 4 (proof of Theorem 3.2).

THEOREM 3.2. *Let $f = (\alpha_{d-1}, \cdots, \alpha_0)$ be a function in $RG \subseteq \mathbf{r}^{\mathbf{d}}$. Define*

$$\rho(f) = \sum_{t=0}^{d-2} \alpha_t T^{(r)}(t, m_t),$$

*where $m_t = \max\{\alpha_j : j > t\}$. Then ρ is the ranking function for the lex list of RG-functions and hence the ranking function for the lex list of partitions of **d** into at most r blocks.*

TABLE 2
(a) $T^{(3)}(n, m)$

n \ m	0	1	2	3	4	5	6	7
0	1	1	1	0	0	0	0	0
1	2	3	3	0	0	0	0	
2	5	9	9	0	0	0		
3	14	27	27	0	0			
4	41	81	81	0				
5	122	243	243					
6	365	729						
7	1094							

(b) $T^{(4)}(n, m)$

n \ m	0	1	2	3	4	5	6	7
0	1	1	1	1	0	0	0	0
1	2	3	4	4	0	0	0	
2	5	10	16	16	0	0		
3	15	36	64	64	0			
4	51	136	256	256				
5	187	528	1024					
6	715	2080						
7	2795							

(c) $T^{(8)}(n, m)$

n \ m	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	
2	5	10	17	26	37	50		
3	15	37	77	141	235			
4	52	151	372	799				
5	203	674	1915					
6	877	3263						
7	4140							

For example $\{\{0, 3\}, \{1, 4\}, \{2\}\}$ corresponds to $f = (\alpha_4, \alpha_3, \alpha_2, \alpha_1, \alpha_0) = (0, 1, 2, 0, 1)$. From Theorem 3.2 and Table 2(c), we obtain ($d = 5$)

$$\begin{aligned}\rho(f) &= 1 \cdot T^{(5)}(3, 0) + 2 \cdot T^{(5)}(2, 1) + 0 \cdot T^{(5)}(1, 2) + 1 \cdot T^{(5)}(0, 2) \\ &= 1 \cdot 15 + 2 \cdot 10 + 0 + 1 \cdot 1 = 36\end{aligned}$$

which is the rank of $\{\{0, 3\}, \{1, 4\}, \{2\}\}$ in the lex list of partitions of **5**.

Similarly, using Table 2(a), we obtain $\rho(f) = 1 \cdot 14 + 2 \cdot 9 + 1 \cdot 1 = 33$ which is the rank of this partition in the lex list of partitions of **5** into 3 or fewer blocks. These examples can be checked directly from Table 1.

A basic sublist of the list of all partitions of **d** into r or fewer blocks is the list of partitions of **d** into exactly r blocks. This list is treated in a manner analogous to the above case. For $n \geqq 0$, $m \geqq 0$, define integers $E^{(r)}(n, m)$ recursively according to the following rules:

(i) $$E^{(r)}(0, m) = \begin{cases} 1 & \text{if } m = r - 1, \\ 0 & \text{if } m \neq r - 1. \end{cases}$$

(ii) $$E^{(r)}(n, m) = (m + 1)E^{(r)}(n - 1, m) + E^{(r)}(n - 1, m + 1).$$

In Table 3(a) and (b) we give a partial list of these numbers for $r = 3$, and 4.

The partitions of **d** into exactly r parts correspond to the surjective *RG* functions. These functions are easily generated in lex order.

TABLE 3
(a) $E^{(3)}(n, m)$

n \ m	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0
1	0	1	3	0	0	0	0	
2	1	5	9	0	0	0		
3	6	19	27	0	0			
4	25	65	81	0				
5	90	211	243					
6	301	665						
7	966							

(b) $E^{(3)}(n, m)$

n \ m	0	1	2	3	4	5	6	7
0	0	0	0	1	0	0	0	0
1	0	0	1	4	0	0	0	
2	0	1	7	16	0	0		
3	1	9	37	64	0			
4	10	55	175	256				
5	65	285	781					
6	350	1351						
7	1701							

THEOREM 3.3. *Let* $f=(\alpha_{d-1}, \cdots, \alpha_0)$ *be a surjective function in* $RG \subseteq \mathbf{r}^{\mathbf{d}}$. *Define*

$$\rho(f)=\sum_{t=0}^{d-1} \alpha_t E^{(r)}(t, m_t),$$

where $m_t = \max\{\alpha_j : j > t\}$. *Then* ρ *is the ranking function for the lex list of surjective RG functions and hence the ranking function for the lex list of partitions of* $\mathbf{d}$ *into exactly r blocks.*

For example, using Table 3(a), the partition $\{\{0, 3\}, \{1, 4\}, \{2\}\}$ has rank $1 \cdot 6 + 2 \cdot 5 + 1 \cdot 1 = 17$ in the lex list of partitions of **5** into exactly 3 parts.

We now discuss the inversion algorithm for the ranking function ρ of Theorems 3.2 and 3.3.

We assume we are given an integer p and we wish to find the rank p partition of the lists referred to in Theorems 3.2 and 3.3. In the case of Theorems 3.2, fill $A(n, m) \leftarrow T^{(r)}(n, m)$ and in the case of Theorem 3.3, fill $A(n, m) \leftarrow E^{(r)}(n, m)$.

The following algorithm produces the rank p function $f = (\alpha_{d-1}, \cdots, \alpha_0)$ in the list.

Step 1. $n \leftarrow d-2$, $m \leftarrow 0$, $x \leftarrow p$, and $\alpha_i \leftarrow 0$ for all i.

Step 2. $\alpha_n \leftarrow \max\{z : zA(n, m) \leqq x, z \leqq m+1\}$.

Step 3. If $\alpha_n A(n, m) = x$, then go to (4); else $x \leftarrow x - \alpha_n A(n, m)$, $n \leftarrow n-1$, $m \leftarrow \max(m, \alpha_n)$, go to (2).

Step 4. Stop: $(\alpha_{d-1}, \cdots, \alpha_0)$ is the rank p function.

For example, to find the rank 500 partition of **8** into exactly 4 blocks, we use the above algorithm together with Table 3(b). The basic parameters progress as follows:

We have $p = 500$, $d = 8$, $A(n, m) \leftarrow E^{(4)}(n, m)$ (Table 3(b)).

Step 1. $n \leftarrow 6$, $m \leftarrow 0$, $x \leftarrow 500$, and $\alpha_i \leftarrow 0$ for $i = 0, \cdots, 7$.

Step 2. $\alpha_6 \leftarrow \max\{z : zA(6, 0) \leqq 500, z \leqq 1\}$, $\therefore \alpha_6 \leftarrow 1$.

Step 3. $n \leftarrow 5$, $m \leftarrow 1$, $x \leftarrow 150$.

Step 4. $\alpha_5 \leftarrow \max\{z : zA(5, 1) \leqq 150, z \leqq 2\}$, $\therefore \alpha_5 \leftarrow 0$.

Step 5. $n \leftarrow 4$, $m \leftarrow 1$, $x \leftarrow 150$.

Step 6. $\alpha_4 \leftarrow \max\{z : zA(4, 1) \leqq 150, z \leqq 2\}$, $\therefore \alpha_4 \leftarrow 2$.

Step 7. $n \leftarrow 3$, $m \leftarrow 2$, $x \leftarrow 40$.

Step 8. $\alpha_3 \leftarrow \max\{z : zA(3, 2) \leqq 40, z \leqq 3\}$, $\therefore \alpha_3 \leftarrow 1$.

Step 9. $n \leftarrow 2$, $m \leftarrow 2$, $x \leftarrow 3$.

Step 10. $\alpha_2 \leftarrow \max\{z : zA(2, 2) \leqq 3, z \leqq 3\}$, $\therefore \alpha_2 \leftarrow 0$.

Step 11. $n \leftarrow 1$, $m \leftarrow 2$, $x \leftarrow 3$.

Step 12. $\alpha_1 \leftarrow \max\{z : zA(1, 2) \leqq 3, z \leqq 3\}$, $\therefore \alpha_1 \leftarrow 3$.

Step 13. $\alpha_1 A(1, 2) = 3 \cdot 1 = 3 = x$ so STOP.

Thus we obtain the *RG* function $(\alpha_7, \alpha_6, \cdots, \alpha_0) = (0, 1, 0, 2, 1, 0, 3, 0)$ which gives $\{\{0, 2, 5, 7\}, \{1, 4\}, \{3\}, \{6\}\}$ as the rank 500 partition in the lex list of partitions of **8** into exactly 4 blocks.

It is interesting to note that the partitions dominating a given partition in the lattice of partitions are easily obtained from the associated *RG* functions. Let $f = (\alpha_{d-1}, \cdots, \alpha_0)$ be an *RG* function. The following algorithm produces in lex order the list of *RG* functions associated with the dominating partitions of f:

Step 1. $i \leftarrow 1$, $m \leftarrow \max\{\alpha_{d-1}, \cdots, \alpha_0\}$.

Step 2. $j \leftarrow 0$.

Step 3. $(y_{d-1}, \cdots, y_0) \leftarrow (\alpha_{d-1}, \cdots, \alpha_0)$.

Step 4. For $t = 0, \cdots, d-1$ if $\alpha_t > i$ then $y_t \leftarrow y_t - 1$.

Step 5. For $t = 0, \cdots, d-1$ if $\alpha_t = i$ then $y_t \leftarrow j$.

Step 6. Write $(y_{d-1}, \cdots, y_0)$.

Step 7. If $j < i-1$ then $j \leftarrow j+1$ and go to (5).

Step 8. If $i < m$ then $i \leftarrow i+1$ and go to (2).

Step 9. Stop.

For example, for $f=(0,1,0,2,1,0,3,0)$ corresponding to $\{\{0,2,5,7\},\{1,4\},\{3\},\{6\}\}$ we obtain:

i	j	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	Dominating partitions
1	0	0	0	0	1	0	0	2	0	$\{0,1,2,4,5,7\},\{3\},\{6\}$
2	0	0	1	0	0	1	0	2	0	$\{0,2,3,5,7\},\{1,4\},\{6\}$
2	1	0	1	0	1	1	0	2	0	$\{0,2,5,7\},\{1,3,4\},\{6\}$
3	0	0	1	0	2	1	0	0	0	$\{0,2,5,6,7\},\{1,4\},\{3\}$
3	1	0	1	0	2	1	0	1	0	$\{0,2,5,7\},\{1,4,6\},\{3\}$
3	2	0	1	0	2	1	0	2	0	$\{0,2,5,7\},\{1,4\},\{3,6\}$

Listing and ranking algorithms for two additional classes of partitions follow from the cases previously discussed. For the first class, we consider the list L of (unordered) partitions of $\mathbf{b}$ into k blocks of size t where $b=kt$. Listing the blocks in standard order, we must have the number 0 in the first block, leaving $\binom{b-1}{t-1}$ choices for the remaining elements of the block. The smallest element not in the first block must be in the second block, leaving $\binom{b-t-1}{t-1}$ choices for this block. Let $L'=\binom{\mathbf{b-1}}{\mathbf{t-1}}\times\binom{\mathbf{b-t-1}}{\mathbf{t-1}}\times\cdots\times\mathbf{1}$ be listed in lex order. Given a function $(\alpha_{k-1},\cdots,\alpha_0)$ in L', we construct an element $\{A_{k-1},\cdots,A_0\}$ of L as follows:

(a) $0\in A_{k-1}$ and $A_{k-1}-\{0\}$ is the rank α_{k-1} subset of $\mathbf{b}-\{0\}$ in colex order (or lex order).

(b) In general, given $A_{k-1},\cdots,A_{k-t}$, we put $x_t\in A_{k-1-t}$ where $x_t=\min(\mathbf{b}-\bigcup_{j=1}^{t}A_{k-j})$ and let $A_{k-t-1}-\{x_t\}$ be the rank a_{k-t-1} subset of $\mathbf{b}-\bigcup_{j=1}^{t}A_{k-j}-\{x_t\}$ in colex order (or lex order).

A calculation where lex order is used in the above correspondence is given after Theorem 3.4 below. Henceforth, we shall always assume *colex order* is used in reference to Theorem 3.4.

This construction defines a bijection φ between L' and L, and thus induces a linear order on L (not the same as lex order on partitions).

The generation and ranking algorithms for L' follow from Theorem 2.2.

THEOREM 3.4. *Let $\{A_{k-1},\cdots,A_0\}$ be a partition of $\mathbf{b}$, $b=kt$, into k blocks of size t. The function*

$$\rho\{A_{k-1},\cdots,A_0\}=\sum_{j=0}^{k-1}\alpha_j c_j,$$

where $c_0=1$, $c_{j+1}=\binom{b-(k-j-1)t-1}{t-1}c_j$ *for* $j=0,\cdots,k-2$, *and* $(\alpha_{k-1},\cdots,\alpha_0)=\varphi^{-1}\{A_{k-1},\cdots,A_0\}$ *is the ranking function for the list L of all such partitions.*

For example, consider $b=12$, $k=3$, $t=4$ and $\{A_2,A_1,A_0\}=\{\{0,6,8,11\},\{1,5,7,10\},\{2,3,4,9\}\}$. By Theorem 2.3, we first find the rank of $\{6,8,11\}$ in the lex list of subsets of $\mathbf{12}-\{0\}$ of size 3. The ranks of 6, 8 and 11 are

5, 7 and 10 in $\mathbf{12}-\{0\}$. We thus have $\varphi\{5, 7, 10\} = 165-1-\binom{0}{1}-\binom{3}{2}-\binom{5}{3} =$ 151. Similarly, we find the rank of $A_1-\{1\}$ in $\mathbf{12}-\{0, 1, 6, 8, 11\}$. The answer is 32. Thus $\varphi^{-1}\{A_2, A_1, A_0\} = (151, 32, 0)$. Theorem 3.4, we have $\rho\{A_2, A_1, A_0\} = 35(151)+1(32) = 5{,}317$. Conversely, to find the rank 1000 partition in this list, we first find rank 1000 function in the lex list $\binom{\mathbf{11}}{\mathbf{3}}\times\binom{\mathbf{7}}{\mathbf{3}}\times\binom{\mathbf{3}}{\mathbf{3}}$ which is (28, 20, 0). Next we find the rank 28 subset in lex order of $\mathbf{12}-\{0\}$ which is found by computing (using Theorem 2.3 inversion algorithm) the rank $165-1-28 = 136$ subset in colex order. This subset is $\{1, 6, 10\}$ so the rank 28 subset in lex order is $\{0, 4, 9\}$ in terms of ranks or $\{1, 5, 10\}$ in $12-\{0\}$. Thus the first block is $\{0, 1, 5, 10\}$. Similarly, we find the rank 20 subset in $\{3, 4, 6, 7, 8, 9, 11\}$ which in terms of ranks is $\{1, 3, 5\}$ corresponding to the subset $\{4, 7, 9\}$. Thus the rank 1000 partition is $\{\{0, 1, 5, 10\}, \{2, 4, 7, 9\}, \{3, 6, 8, 11\}\}$.

Suppose we are given a fixed ordered partition $\mathbf{A} = (A_1, \cdots, A_d)$ of $\mathbf{d}$. Using Theorem 3.4 together with Theorem 2.2, we obtain a ranking algorithm for the list $P(\mathbf{A})$ of partitions of the form

$$\{A_{11}, \cdots, A_{1k_1}; A_{21}, \cdots, A_{2k_2}; \cdots; A_{d1}, \cdots, A_{dk_d}\},$$

where $\{A_{t1}, \cdots, A_{tk_t}\}$ is a partition of A_t into k_t blocks of size t. By Theorem 3.4, we have, for each t, an algorithm for computing the rank of $\{A_{t1}, \cdots, A_{tk_t}\}$. This rank will be a number in the set $\mathbf{B_t}$ where $B_t = \binom{b_t-1}{t-1}\binom{b_t-t-1}{t-1}\cdots$. Thus we have a bijection between $P(\mathbf{A})$ and the set $\mathbf{B_1}\times\cdots\times\mathbf{B_d}$. By listing the latter set in lex order, we define an order on $P(\mathbf{A})$ (again not the same as lex order on partitions). Using Theorem 2.2, we have immediately

THEOREM 3.5. *Let* $\mathscr{C} = \{A_{11}, \cdots, A_{1k_1}; A_{21}, \cdots, A_{2k_2}, \cdots\}$ *be a partition in* $P(\mathbf{A})$, $\mathbf{A} = (A_1, \cdots, A_d)$. *Define*

$$\rho(\mathscr{C}) = \sum_{t=1}^{d} y_t c_t,$$

where y_t *is the rank of* $\{A_{t1}, \cdots, A_{tk_t}\}$ *in the lex list of partitions of* A_t *into* k_t *blocks of size* t, $c_d = 1$, $c_t = B_{t+1}\cdots B_d$, $t = 1, \cdots, d-1$ (B_t *as defined above*). *Then* ρ *is the ranking function for* $P(\mathbf{A})$ *with the order induced by lex order on* $\mathbf{B_1}\times\cdots\times\mathbf{B_d}$.

Finally, from Theorem 2.4 and Theorem 3.5, we obtain a listing and ranking algorithm for partitions corresponding to a given "type" $1^{k_1}2^{k_2}\cdots d^{k_d}$. Let $k = (k_1, \cdots, k_d)$. A partition of type $1^{k_1}2^{k_2}\cdots d^{k_d}$ is a partition with k_t blocks of size t, $t = 1, \cdots, d$. Such a partition can be specified by first choosing an ordered partition $\mathbf{A} = (A_1, \cdots, A_d)$ corresponding to the multinomial index $(a_1, \cdots, a_d)$ where $a_t = tk_t$ (see Theorem 2.4) and then choosing a partition in $P(\mathbf{A})$ according to Theorem 3.5. We list the pairs of ranks of these choices in lex order and thus define a linear order on $P(\mathbf{k})$, the partitions of type $1^{k_1}2^{k_2}\cdots d^{k_d}$. By applying Theorem 2.4, Theorem 3.5 and then a trivial application of Theorem 2.2 to the pairs of ranks, we obtain a ranking algorithm for $P(\mathbf{k})$.

For example, we determine the rank of $\{\{1\}, \{8\}, \{2, 7\}, \{5, 6\}, \{0, 3, 4\}\}$ in $P(\mathbf{k})$, $\mathbf{k} = (2, 2, 1, 0, \cdots, 0)$. These are the partitions of **9** of type $1^2 2^2 3$. This partition corresponds to the ordered partition $\mathbf{A} = (\{1, 8\}, \{2, 5, 6, 7\}, \{0, 3, 4\},$

$\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing$) of type $1^2 2^2 3$. The set $\{1, 8\}$ has rank 29 in the colex list of subsets of size 2 from **9**. The set $\{2, 5, 6, 7\}$ has rank 32 in the colex list of subsets of size 4 from 0, 2, 3, 4, 5, 6, 7, etc. Thus, converted to ranks, this ordered partition becomes $(29, 32, 0, 0, 0, \cdots, 0) = (\xi(A_1), \cdots, \xi(A_7))$. By Theorem 2.4, $\rho(\mathbf{A}) = 29 \cdot 35 + 32 \cdot 1 + 0 + \cdots + 0 = 1047$. We have $B_1 = \binom{1}{0}\binom{0}{0} = 1$, $B_2 = \binom{3}{1}\binom{1}{1} = 3$, $B_3 = \binom{2}{2} = 1$. By Theorem 3.5 (trivial in this case!), the partition $\{\{1\}, \{8\}\}$ has rank 0, the partition $\{\{2, 7\}, \{5, 6\}\}$ has rank 2, and all of the rest have rank 0. Thus in $\mathbf{B_1} \times \mathbf{B_2} \times \mathbf{B_3} \times \cdots \times \mathbf{B_7}$, we have the element $(0, 2, 0, \cdots, 0)$. Thus the pair of ranks associated with $\{\{1\}, \{8\}, \{2, 7\}, \{5, 6\}, \{0, 3, 4\}\}$ is $(1047, 2)$, so this partition has rank $3(1047) + 2 = 3143$ in the list of partitions of type $1^2 2^2 3$ of **9**. The process is easily inverted by applying the inversion algorithms for Theorems 2.2, 2.4 and 3.5. For example, we find the partition of $P(\mathbf{k})$ with rank 10,000,000, $\mathbf{k} = (0, 2, 2, 0, 1, 1, 0, \cdots, 0)$, $d = 21$. Here $(a_1, \cdots, a_d) = (0, 4, 6, 0, 5, 6, 0, \cdots, 0)$ and $|P(\mathbf{A})| = 30$. Find x, y, $30x + y = 10{,}000{,}000$. $(x, y) = (333333, 10)$. To find the rank 333,333 ordered partition corresponding to $(0, 4, 6, 0, 5, 6)$ we find the rank 333,333 function in lex order in $\binom{\mathbf{21}}{\mathbf{0}} \times \binom{\mathbf{21}}{\mathbf{4}} \times \binom{\mathbf{17}}{\mathbf{6}} \times \binom{\mathbf{11}}{\mathbf{0}} \times \binom{\mathbf{11}}{\mathbf{5}} \times \binom{\mathbf{6}}{\mathbf{6}}$. Thus we solve $\alpha\binom{17}{6}\binom{11}{5} + \beta\binom{11}{5} + \gamma = 333{,}333$. This gives $(\alpha, \beta, \gamma) = (0, 721, 231)$. The rank 0 subset of **21** of size 4 is $\{0, 1, 2, 3\}$. The rank 721 subset (colex) of $\{4, 5, \cdots, 20\}$ of size 6 is given by $\{x_1, \cdots, x_6\}$, where $\binom{x_1}{1} + \cdots + \binom{x_6}{6} = 721$ and x_i is the rank of the ith element. We find $x_1 = 0$, $x_2 = 2$, $x_3 = 3$, $x_4 = 5$, $x_5 = 10$, $x_6 = 11$ corresponding to the subset $\{4, 6, 7, 9, 14, 15\}$. Similarly, the rank 231 subset (colex) of the remaining 11 elements is $\{5, 8, 17, 18, 19\}$. Thus the rank 333,333 ordered partitions of **21** corresponding to the multinomial index $(0, 4, 6, 0, 5, 6, 0, \cdots, 0)$ is

$$\mathbf{A} = (\varnothing, \{0, 1, 2, 3\}, \{4, 6, 7, 9, 14, 15\}, \varnothing,$$
$$\{5, 8, 17, 18, 19\}, \{10, 11, 12, 13, 16, 20\}, \varnothing, \cdots, \varnothing).$$

We must now find the rank 10 element of $P(\mathbf{A})$. $P(\mathbf{A})$ is order isomorphic to $\binom{\mathbf{3}}{\mathbf{1}}\binom{\mathbf{1}}{\mathbf{1}} \times \binom{\mathbf{5}}{\mathbf{2}}\binom{\mathbf{2}}{\mathbf{2}} = \mathbf{3} \times \mathbf{10}$ in lex order. Thus we solve $\gamma \cdot 10 + \delta = 10$ and find the rank γ partition of $\{0, 1, 2, 3\}$ into 2 blocks of size 2 and the rank δ partition of $\{4, 6, 7, 9, 14, 15\}$ into 2 blocks of size 3. Here $\gamma = 1$, $\delta = 0$. The rank 1 partition of $\{0, 1, 2, 3\}$ is $\{\{0, 2\}, \{1, 3\}\}$ and the rank 0 partition of $\{4, 6, 7, 9, 14, 15\}$ is $\{\{4, 6, 7\}, \{9, 14, 15\}\}$. Thus the rank 10,000,000 partition of $P(\mathbf{k})$, $\mathbf{k} = (0, 2, 2, 0, 1, 1, 0, \cdots, 0)$ is

$$\{\{0,2\},\{1,3\},\{4,6,7\},\{9,14,15\},\{5,8,17,18,19\},\{10,11,12,13,16,20\}\}.$$

4. Proofs and additional comments. To prove Theorem 2.2, we let

$$S = \{(\beta_{d-1}, \cdots, \beta_0) : (\beta_{d-1}, \cdots, \beta_0) < (\alpha_{d-1}, \cdots, \alpha_0)\}$$

be the set of functions in $\mathbf{r_{d-1}}\times\cdots\times\mathbf{r_0}$ that are lexicographically less than $(\alpha_{d-1},\cdots,\alpha_0)$. Let $S_j=\{(\beta_{d-1},\cdots,\beta_0)$: where $\beta_{d-i}=\alpha_{d-i}$ for $i=1,\cdots,j-1$ and $\beta_{d-j}<\alpha_{d-j}\}$. Observe that the S_j are disjoint and $S=\bigcup_{j=1}^{d}S_j$. For any fixed value $\beta_{d-j}=k<\alpha_{d-j}$, there are $r_{d-j-1}r_{d-j-2}\cdots r_0=c_{d-j}$ functions in S_j. Thus $|S_j|=\alpha_{d-j}c_{d-j}$ and

$$|S|=(\alpha_{d-1},\cdots,\alpha_0)=\sum_{j=0}^{d}\alpha_{d-j}c_{d-j}=\sum_{t=0}^{d-1}\alpha_t c_t.$$

To prove that the inversion algorithm associated with Theorem 2.2 works, we proceed by induction. Suppose that the inversion algorithm is valid for the product of any $d-1$ coordinate space: $\mathbf{r_{d-2}}\times\cdots\times\mathbf{r_0}$. Suppose we are to find the rank m function of $\mathbf{r_{d-1}}\times\mathbf{r_{d-2}}\times\cdots\times\mathbf{r_0}$. Observe that $m\leqq r_{d-1}c_{d-1}-1$ so the α_{d-1} produced by the algorithm satisfies $\alpha_{d-1}<r_{d-1}$ or $\alpha_{d-1}\in\mathbf{r_{d-1}}$. By the maximality of α_{d-1}, we must have $x=m-\alpha_{d-1}c_{d-1}<c_{d-1}=r_{d-2}c_{d-2}$. Thus by the induction hypothesis and the recursive structure of the algorithm we have

$$\sum_{t=2}^{d}\alpha_{d-t}c_{d-t}=x,$$

where $(\alpha_{d-2},\cdots,\alpha_0)\in\mathbf{r_{d-2}}\times\cdots\times\mathbf{r_0}$. Thus $(\alpha_{d-1},\cdots,\alpha_0)$ produced by the algorithm is the rank m function in $\mathbf{r_{d-1}}\times\cdots\times\mathbf{r_0}$. Clearly, the algorithm works for $d=1$, so the proof is complete.

We now prove Theorem 3.1. Suppose first that f is in RG and show that the coimage written in the order $f^{-1}(0),f^{-1}(1),\cdots$ is in standard order. Suppose $f^{-1}(0),\cdots,f^{-1}(t-1)$ are in standard order and let j be the smallest integer not in $f^{-1}(0)\cup\cdots\cup f^{-1}(t-1)$. Thus $f(j)\geqq t$. But $f\in RG$ implies that $f(j)\leqq(\max_{i<j}f(i))+1\leqq(t-1)+1=t$. Thus $f(j)=t$ or $j\in f^{-1}(t)$. The single block $f^{-1}(0)$ is trivially in standard order so the result follows by induction.

Conversely, suppose $f\in\mathbf{r^d}$ and the blocks $f^{-1}(0),f^{-1}(1),\cdots$ are in standard order. We show that f is in RG. We show that if $t\in\mathbf{d}$ and $t\in f^{-1}(x)$, then $f^{-1}(z)\cap\mathbf{t}\neq\varnothing$ for any $z<x$. This, of course, implies that $\max_{j<t}f(j)\geqq x-1$ or $x\leqq\max_{j<t}f(j)+1$. For $S\subseteq\mathbf{d}$, let $S^c=\mathbf{d}-S$. For $z<x$, we have $\min f^{-1}(z)=\min(f^{-1}(0)\cup\cdots\cup f^{-1}(z-1))^c<\min(f^{-1}(0)\cup\cdots\cup f^{-1}\ (x-1))^c=\min f^{-1}(x)\leqq t$. Thus $f^{-1}(z)\cap\mathbf{t}\neq\varnothing$ and the proof is complete.

The basic Theorem 3.2 admits a simple combinatorial proof analogous to that of Theorem 2.2. First we interpret the numbers $T^{(r)}(n,m)$ combinatorially. Let $T^{(r)}(n,m)$ denote the cardinality of the set

$$\{(x_{n-1},\cdots,x_0)\colon(\alpha_{d-1},\cdots,\alpha_n,x_{n-1},\cdots,x_0)\in RG,\text{ where }\max\{\alpha_{d-1},\cdots,\alpha_n\}=m\}.$$

That is, $T^{(r)}(n,m)$ represents the number of ways of filling in the last n coordinates of an RG function in $\mathbf{r_d}$ when the maximum up to that point is m. If we take $x_{n-1}=m+1$, then there are $T(n-1,m+1)$ ways of computing $x_{n-2},\cdots,x_0$. If $x_{n-1}=0,\cdots,m$, there are $T(n-1,m)$ ways of filling in $x_{n-2},\cdots,x$. Thus we obtain the basic recursion

$$T(n,m)=(m+1)T(n-1,m)+T(n-1,m+1).$$

Clearly, $T(1, m) = m+1$ for $m < r-1$ and $T(1, r-1) = r-1$. We take $T(1, m) = 0$ if $m > r-1$.

To prove Theorem 3.2, we let

$$S' = \{(\beta_{d-1}, \cdots, \beta_0): (\beta_{d-1}, \cdots, \beta_0) < (\alpha_{d-1}, \cdots, \alpha_0)\}$$

denote the set of RG functions lexicographically less than the given RG function $(\alpha_{d-1}, \cdots, \alpha_0)$. Let $S'_j = \{(\beta_{d-1}, \cdots, \beta_0)$: where $\beta_{d-i} = \alpha_{d-i}$ for $i = 1, \cdots, j-1$ and $\beta_{d-j} < \alpha_{d-j}\}$. Again $S'_j \subseteq RG$. The S'_j are disjoint and $S' = \bigcup_{j=1}^{d} S'_j$. For any fixed value $\beta_{d-j} = k < \alpha_{d-j} \leqq m_{d-j}+1$, there are $T(d-j, m_{d-j})$ functions in S'_j giving $|S'_j| = \alpha_{d-j} T(d-j, m_{d-j})$. The result follows immediately.

It is instructive to look carefully at the relationship between the rule of succession in RG and the basic recursion

$$\text{(R)} \qquad T^{(r)}(n, m) = (m+1)T^{(r)}(n-1, m) + T^{(r)}(n-1, m+1).$$

Consider any function $f = (\alpha_{d-1}, \cdots, \alpha_0)$ in $RG \subseteq \mathbf{r}^{\mathbf{d}}$ other than the last function. Let t denote the largest index such that α_t is not maximal. There are two possibilities for f (setting $m = m_t$).

$$\text{(1)} \qquad \begin{aligned} &f = (\alpha_{d-1}, \cdots, \alpha_t, m+1, \cdots, m+t) \\ &\text{if } \max(\alpha_{d-1}, \cdots, \alpha_t) = m \quad \text{and} \quad m+t < r-1, \end{aligned}$$

or

$$\text{(2)} \qquad \begin{aligned} &f = (\alpha_{d-1}, \cdots, \alpha_t, m+1, \cdots, r-2, r-1, \cdots, r-1) \\ &\text{if } \max(\alpha_{d-1}, \cdots, \alpha_t) = m \quad \text{and} \quad m+t \geqq r-1. \end{aligned}$$

In case (2) $\alpha_0 = \alpha_1 = \cdots = \alpha_{m+t-r+1} = r-1$.

In each case the next element f' in the lex list RG is

$$f' = (\alpha_{d-1}, \cdots, \alpha_t, \alpha_t + 1, 0, \cdots, 0).$$

Thus we have (writing simply T for $T^{(r)}$)

$$\text{(1')} \qquad \begin{aligned} \rho(f') - \rho(f) = T(t, m) - ((m+1)T(t-1, m) + (m+2)T(t-2, m+1) \\ + \cdots + (m+t)T(0, m+t-1)), \qquad m+t < r-1, \end{aligned}$$

$$\text{(2')} \qquad \begin{aligned} \rho(f') - \rho(f) = T(t, m) - ((m+1)T(t-1, m) + (m+2)T(t-2, m+1) \\ + \cdots + (r-1)r^{m+t-r+1} + \cdots + (r-1)r + (r-1)). \end{aligned}$$

Observe that in (2') the sum

$$\begin{aligned} (r-1)r^{m-t-r+1} + \cdots + (r-1)r + (r-1) &= (r^{m+t-r+2} - 1) \\ &= (r-1)T(m+n-r+1, r-2) + T(m+n-r+1, r-1) - 1. \end{aligned}$$

Thus to show $\rho(f') - \rho(f) = 1$, we need to verify the two identities.

For $m+t < r-1$:

$$\text{(3)} \qquad \begin{aligned} T(t, m) = (m+1)T(t-1, m) + (m+2)T(t-2, m+1) \\ + \cdots + (m+t)T(0, m+t-1) + 1, \end{aligned}$$

For $m+t \geqq r-1$:

$$T(t, m) = (m+1)T(t-1, m) + (m+2)T(t-2, m+1) + \cdots + (r-1)T(m+n-r+1, r-2) + T(m+n-r+1, r-1). \tag{4}$$

These identities follow by repeated application of the recursion (R) above.

The proof of Theorem 3.3 is directly analogous to the proof of Theorem 3.2. Let RG^* denote the set of surjective $RG \subseteq \mathbf{r}^{\mathbf{d}}$ functions. Interpret $E^{(r)}(n, m)$ as the cardinality of the set

$$\{(x_{n-1}, \cdots, x_0): (\alpha_{d-1}, \cdots, \alpha_n, x_{n-1}, \cdots, x_0) \in RG^*\},$$

where $\max\{\alpha_{d-1}, \cdots, \alpha_n\} = m$. This cardinality is independent of the choice of $\alpha_{d-1}, \cdots, \alpha_n$ since the RG property implies that in fact the set $\{\alpha_{d-1}, \cdots, \alpha_n\} = \{0, 1, \cdots, m\}$ if $\max\{\alpha_{d-1}, \cdots, \alpha_n\} = m$. With these modifications, the proof of Theorem 3.2 becomes a proof of Theorem 3.3. The associated inversion algorithm is easily proved.

Schemes for ranking or randomly generating certain classes of partitions based on recursions for Bell numbers may be found in [4], [5].

5. Final remarks.[1] Many classes of combinatorial structures arise by the process of partitioning a finite set and then by constructing a "connected object" on each block of the partition. For example, a labeled graph with labels $1, 2, \cdots, n$ can be obtained by partitioning the set $\mathbf{n} = \{1, \cdots, n\}$ into blocks $\{B_1, \cdots, B_k\}$ and constructing a connected graph on each block B_i. A permutation on $\mathbf{n}$ may be constructed by putting a full cycle on each block B_i. A "labeled forest" on $\mathbf{n}$ may be constructed by putting a labeled rooted tree on each block B_i. A nonattacking configuration of $n-k$ rooks on an $(n-1) \times n$ board can be viewed as a construction of a partition $\{B_1, \cdots, B_k\}$ and then a specification of a permutation on each B_i. In cases where the connected objects to be constructed on each block admit a reasonable ranking algorithm, sets such as those above can be ranked by combining the previously mentioned ranking procedure for unordered partitions corresponding to a fixed type with this ranking algorithm for connected objects. This provides a good way, in general, to rank such structures of fixed "type". As a ranking procedure for the set as a whole, serious problems arise due to the fact that, in general, the "types" (corresponding to numerical partitions of n) must be generated. For example, this procedure is clearly not the way to rank the set of all permutations on $\mathbf{n}$ (see [3]), but is a reasonable way to rank all permutations of a fixed type. The connected objects in the latter three of the above four examples are all easily ranked. The connected graphs do not seem to admit a reasonable ranking function. All of these structures are special cases of a more general class of sets which admit an algebraic structure called a "prefab" by Bender and Goldman (our examples are actually "exponential" prefabs) [1]. The uniqueness of the prime decomposition in a prefab suggests possible ranking

[1] The possibility of extending these ranking algorithms to certain classes of prefabs was pointed out to the author by the referee.

functions for these structures in general. However, the multivalued character of the operation of composition in the prefab seems to rule out any very useful general approach to ranking prefabs. As in the case of permutations above, each set must be looked at carefully for the most easily ranked linear order. The notion of a prefab is itself one of several possible axiomatic approaches to generating functions. The axioms for prefabs are constructed to guarantee that the generating function associated with a certain class of sets can be written as a product of simpler generating functions. No assumptions about linear order are involved in a prefab, whereas the careful choice of a linear order is the key to a good ranking algorithm.

Acknowledgment. The author wishes to thank Rod Canfield for a careful reading of this paper.

REFERENCES

[1] E. BENDER AND J. GOLDMAN, *Enumerative uses of generating functions*, Indiana Univ. Math. J., 20 (1971), pp. 753–765.

[2] G. HUTCHINSON, *Partitioning algorithms for finite sets*, Comm. ACM, 6 (1963), pp. 613–614.

[3] D. H. LEHMER, *The machine tools of combinatorics*, Applied Combinatorial Mathematics, E. F. Beckenbach, ed., John Wiley, New York, 1964, pp. 5–31.

[4] M. B. WELLS, *Elements of Combinatorial Computing*, Pergamon Press, New York, 1971, pp. 150–154.

[5] A. NIJENHUIS AND H. WILF, *Combinatorial Algorithms*, Academic Press, New York, 1974.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

FAST PARALLEL MATRIX INVERSION ALGORITHMS*

L. CSANKY†

Abstract. The parallel arithmetic complexities of matrix inversion, solving systems of linear equations, computing determinants and computing the characteristic polynomial of a matrix are shown to have the same growth rate. Algorithms are given that compute these problems in $O(\log^2 n)$ steps using a number of processors polynomial in n. (n is the order of the matrix of the problem.)

Key words. complexity of parallel computation, parallel algorithms, functions of matrices

1. Introduction. The parallel arithmetic complexity of solving systems of linear equations has been an open question for several years. The gap between the complexity of the best algorithms ($2n + O(1)$, where n is the number of unknowns/equations) and the only proved lower bound ($2\log n$ (all logarithms in this paper are of base two)) was huge. Csanky [1] exhibited an algorithm for solving systems of linear equations in $2n - O(\log^2 n)$ steps. This also proved that Gaussian elimination, Jordan elimination, Jacobi's method and Strassen's method are not the fastest for parallel computation.

2. The model of parallel computation and some basic definitions. The *model* has an arbitrary number of identical processors with independent control and an arbitrarily large memory with unrestricted access. Each processor is capable of taking its operands from the memory, performing any one of the binary operations $+$, $-$, $*$, $/$ and storing the result in the memory in unit time. This unit time is called a *step*. (The bookkeeping overhead is ignored.) Before starting the computation, the input data is stored in the memory. The *parallel arithmetic complexity of the computation* is the least number of steps necessary to produce the result in the memory.

If C is a computational problem of size n, then the *parallel arithmetic computational complexity* $C(n)$ of C is the least number of steps necessary to compute C for any possible input.

Let A be an order n matrix. Let $\det(A)$, $\operatorname{adj}(A)$, $\operatorname{tr}(A)$ denote the determinant of A, the adjoint of A and the trace of A, respectively. Unless specified otherwise, capital letters denote matrices, lower case letters denote scalars.

Let $I(n), E(n), D(n), P(n)$ denote the parallel arithmetic complexity of inverting order n matrices, solving a system of n linear equations in n unknowns, computing order n determinants and finding the characteristic polynomials of order n matrices, respectively.

3. Results.

LEMMA 1. $2\log n \leqq I(n), E(n), D(n), P(n)$.

Proof. The proof is direct from the fact that in each case, at least one partial result is a nontrivial function of at least n^2 variables and fan in argument. Q.E.D.

* Received by the editors June 10, 1975, and in revised form September 3, 1975.

† Concord, California. This work was supported in part by National Science Foundation under Grant DCR72-03725-A02.

THEOREM 2. $I(n) = O(f(n)) \Leftrightarrow E(n) = O(f(n)) \Leftrightarrow D(n) = O(f(n)) \Leftrightarrow P(n) = O(f(n))$.

Proof. (i) $D(n) \leqq E(n) + \log n + O(1)$. Let D_t denote an order t determinant and let D_n be the determinant to be computed. Define $x_k = D_{n-k}/D_{n-k+1}$, where $1 \leqq k \leqq n-1$ and D_{n-k} is a properly chosen minor of D_{n-k+1}. Since $\prod_{(k)} x_k = D_1/D_n$, $D_n = D_1/\prod_{(k)} x_k$. Thus to compute D_n, compute x_k for all k in $E(n-k+1)$ parallel steps by solving the corresponding systems of equations, then in $\leqq \log n + O(1)$ additional steps, compute D_n.

(ii) $E(n) \leqq I(n) + 2$. Bring $Ax = b$ to the form $A'x = (I)_{*1}$ in two steps by row operations, then invert A' in $I(n)$ steps. $x = (A'^{-1})_{*1}$. ($(A)_{*j}$ denotes the jth column of A.)

(iii) $I(n) \leqq P(n) + 1$. Invert A by the classical inverse formula using that $g(0) = \det(B)$, where g is the characteristic polynomial of B.

(iv) $P(n) \leqq D(n) + \log n + O(1)$. To compute the characteristic polynomial g of A, first compute $g(w^i)$ for all distinct w^i parallel, where w is a primitive $(n+1)$st root of unity, by using the algorithm for computing determinants. This computation takes $D(n) + 1$ steps. Then compute the coefficients of g by fast fourier transform. This takes $\log n + O(1)$ steps.

From Lemma 1 and the four inequalities above, the theorem follows. Q.E.D.

THEOREM 3. $I(n) \leqq O(\log^2 n)$, *and the number of processors used in the algorithm is a polynomial in* n.

Proof. Let $\lambda_1, \cdots, \lambda_n$ denote the roots of the characteristic polynomial $f(\lambda)$ of A. Let

$$s_k = \sum_{i=1}^{n} \lambda_i^k \quad \text{for } 1 \leqq k \leqq n.$$

Then

$$s_k = \operatorname{tr}(A^k) \quad \text{for } 1 \leqq k \leqq n$$

and

$$\begin{bmatrix} 1 & & & & & & \\ s_1 & 2 & & & & & \\ s_2 & s_1 & 3 & & & & \\ s_3 & s_2 & s_1 & 4 & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \\ \vdots & \vdots & \vdots & \vdots & & \ddots & \\ s_{n-1} & s_{n-2} & s_{n-3} & s_{n-4} & \cdots & s_1 & n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ \vdots \\ \vdots \\ c_n \end{bmatrix} = - \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ \vdots \\ \vdots \\ s_n \end{bmatrix},$$

or in a more compact form,

$$S\mathbf{c} = -\mathbf{s}.$$

(These formulas constitute Leverrier's method (Faddeev–Faddeeva [2]) for finding the coefficients of the characteristic polynomial of a matrix.)

To invert A, first compute s_k for $1 \leqq k \leqq n$; this takes $\log^2 n + O(\log n)$ steps and $\frac{1}{2}n^4$ processors. Then invert triangular matrix S in $\log^2 n + O(\log n)$ steps using $O(n^3)$ processors (Heller [4] and Csanky [1] have independently developed such algorithms). Compute c_i for $1 \leqq i \leqq n$ in $\log n + O(1)$ steps from $\mathbf{c} = -S^{-1}\mathbf{s}$ using $O(n^2)$ processors. A is invertible $\Leftrightarrow c_n \neq 0$ and

$$A^{-1} = -\frac{A^{n-1} + c_1 A^{n-2} + \cdots + c_{n-1} I}{c_n}.$$

This formula, since $A^2, \cdots, A^{n-1}$ are already available, can be evaluated in $\log n + O(1)$ steps using $O(n^3)$ processors. Thus

$$I(n) \leqq 2\log^2 n + O(\log n)$$

and

$$p \leqq \tfrac{1}{2}n^4. \qquad \text{Q.E.D.}$$

Alternate proof. Let A be the order n matrix to be inverted. Let the characteristic polynomial $f(\lambda)$ of A be

$$f(\lambda) = \det(\lambda I - A) = \lambda^n + c_1\lambda^{n-1} + \cdots + c_{n-1}\lambda + c_n.$$

Let

$$\operatorname{adj}(\lambda I - A) = I\lambda^{n-1} + B_2\lambda^{n-2} + \cdots + B_{n-1}\lambda + B_n.$$

Using the idea of one of the simplest proofs of the Cayley–Hamilton theorem (Marcus–Ming [5]), the following formulas were obtained:

(1) $$B_1 = I,$$

(2) $$B_k = AB_{k-1} - \frac{I}{k-1}\operatorname{tr}(AB_{k-1}),$$

(3) $$c_i = -\frac{1}{i}\operatorname{tr}(AB_i)$$

and

(4) $$A^{-1} = n\frac{B_n}{\operatorname{tr}(AB_n)}.$$

(A number of people derived essentially the same formulas in recent years. Frame [3] seems to have been the first.)

Define operator T as

$$TN = \operatorname{tr}(N),$$

$$(I + MT)N = N + MTN = N + M\operatorname{tr}(N),$$

where M, N are order n matrices. Then

$$B_k = \left(I - \frac{I}{k-1}T\right)AB_{k-1}$$

and

$$B_n = \left(I - \frac{I}{n-1}T\right)\left\{A\left[\left(I - \frac{I}{n-2}T\right)\{A[\cdots(I - IT)\{A[I]\}\cdots]\}\right]\right\}. \tag{5}$$

In this formula, A can be thought of as a second operator, its action being multiplication on the left.

Observe that

$$T(AT) = (TA)T, \tag{6}$$

that is, operators T and A associate. This implies that the factors in the expression for B_n associate, i.e.,

$$B_n = \left(A - \frac{I}{n-1}TA\right)\left(A - \frac{I}{n-2}TA\right)\cdots\left(A - \frac{I}{2}TA\right)(A - ITA). \tag{7}$$

Thus B_n can be computed in roughly $\log n$ stages by the well-known binary tree method.

Stage 1. For all $i + 1 = 2t$, compute from $(A - [I/(i+1)]TA)(A - (I/i)TA)$ the form

$$A^2 - M^{(1)}_{1\,2t}TA^2 - M^{(2)}_{1\,2t}TA.$$

As a consequence of (6), matrices $M^{(h)}_{1\,2t}$ can be computed independently. The powers of A can also be computed independently.

⋮

Stage j. For all $2^j t$, compute from

$$(A^{2^{j-1}} - M^{(1)}_{j-1\;2^j t}TA^{2^{j-1}} - \cdots - M^{(2^{j-1})}_{j-1\;2^j t}TA)$$
$$\cdot(A^{2^{j-1}} - M^{(1)}_{j-1\;2^j(t-1)}TA^{2^{j-1}} - \cdots - M^{(2^{j-1})}_{j-1\;2^j(t-1)}TA)$$

the form

$$A^{2^j} - M^{(1)}_{j\,2^j t}TA^{2^j} - M^{(2)}_{j\,2^j t}TA^{2^{j-1}} - \cdots - M^{(2^j)}_{j\,2^j t}TA.$$

As a consequence of (6), matrices $M^{(h)}_{j\,2^j t}$ can be computed independently. The powers of A can also be computed independently.

The number of steps stage j takes is roughly

$$\log n + 1 + \log n + 1 + \log 2^j = \log n + \log n + 2 + j,$$

where the first two terms are the number of steps the multiplication on the right by $A^{2^{j-1}}$ and matrices $M^{(h)}$ takes, the third term is the number of steps computing the traces takes, multiplication by the trace takes 1 step and addition of the matrices takes j steps. Thus the total number of steps for all stages is roughly

$$(\log n)(2\log n + 2) + 1 + 2 + 3 + \cdots + \log n = 2.5\log^2 n + 2.5\log n.$$

Since $nB_n/(TAB_n)$ ($TAB_n \neq 0 \Leftrightarrow A$ is invertible) can be computed in $O(\log n)$ additional steps, we have

$$I(n) \leqq 2.5\log^2 n + O(\log n).$$

A very crude upper bound on the number of processors can be obtained by observing that in any stage, the multiplication phase takes the maximum number of processors.

In stage j, the number of matrix multiplications is roughly

$$2^{j-1} + \frac{n}{2^j} 2^{2j-2}.$$

Thus, for the number of processors p, we obtain

$$p \leqq \max_{(j)} \left\{ n^3 2^{j-1} + \frac{n^4}{2^j} 2^{2j-2} \right\} \approx \frac{1}{4} n^5 + \frac{1}{2} n^3. \qquad \text{Q.E.D.}$$

COROLLARY 4. $E(n), D(n), P(n) \leqq O(\log^2 n)$, *and the number of processors used in the algorithms is a polynomial in* n.

Proof. Solve $Ax = b$, where b is a column vector by inverting A; then $x = A^{-1}b$. Compute $\det(A)$ from $\det(A) = -c_n = (TAB_n)/n$. Compute $f(\lambda)$ from $c_i = -(TAB_i)/i$. Q.E.D.

Let PWR(n) denote the parallel arithmetic complexity of computing A^n. Then we have the next theorem.

THEOREM 5. $I(n) \leqq 2\text{PWR}(n) + O(\log n)$.

Proof. To invert A, first compute the triangular matrix S in PWR $(n) + \log n + O(1)$ steps. Then compute $S^2, S^3, \cdots, S^n$ in PWR (n) additional steps.

Since $\log n \leqq \text{PWR}(n)$ and one can compute the coefficients d_i of the characteristic polynomial $g(\lambda)$ of S from the roots of $g(\lambda)$ (these roots are $1, 2, \cdots, n$) in $2 \log n + O(1)$ steps (Csanky [1]), by the time the powers of S are computed, the coefficients of $g(\lambda)$ are also computed. Then

$$S^{-1} = -\frac{S^{n-1} + d_1 S^{n-2} + \cdots + d_{n-1} I}{d_n},$$

and S^{-1} can be computed in $\log n + O(1)$ additional steps. Compute c_i for $1 \leqq i \leqq n$ from $\mathbf{c} = S^{-1}\mathbf{s}$ in $\log n + O(1)$ steps.

$$A^{-1} = -\frac{A^{n-1} + c_1 A^{n-2} + \cdots + c_{n-1} I}{c_n}$$

can be computed in $\log n + O(1)$ additional steps. Q.E.D.

4. Conclusions. This work has decreased the gap between the lower and upper bounds on the parallel arithmetic complexity of problems I, E, D, P. Indeed, we have

$$O(\log n) \leqq I(n), E(n), D(n), P(n) \leqq O(\text{PWR}(n)) \leqq O(\log^2 n).$$

It is our belief that if $I(n) = O(\log n)$, then the algorithm which establishes this will have to use some new, surprising and fundamental result.

Acknowledgments. I am indebted to Professor Phil Spira for his friendship and constructive criticism during the initial stage of this research. I wish to thank Professor Manuel Blum for his encouragement and interest in this work.

REFERENCES

[1] L. CSANKY, *On the parallel complexity of some computational problems*, Ph.D. dissertation, Computer Sci. Div., Univ. of Calif., Berkeley, 1974.

[2] D. K. FADDEEV AND V. N. FADDEEVA, *Computational Methods of Linear Algebra*, W. H. Freeman, San Francisco, 1963.

[3] J. S. FRAME, *A simple recurrent formula for inverting a matrix*, Bull. Amer. Math. Soc., 55 (1949), p. 1045.

[4] D. HELLER, *A determinant theorem with applications to parallel algorithms*, Dept. of Computer Sci., Carnegie-Mellon Univ., Pittsburgh, 1973.

[5] M. MARCUS AND H. MING, *A Survey of Matrix Theory and Matrix Inequalities*, Allyn and Bacon, Boston, 1964.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

ON THE NUMBER OF MULTIPLICATIONS REQUIRED FOR MATRIX MULTIPLICATION*

ROGER W. BROCKETT† AND DAVID DOBKIN‡

Abstract. In this paper we give a new algorithm for matrix multiplication which for n large uses $n^2 + o(n^2)$ multiplications to multiply $n \times p$ matrices by $p \times n$ matrices provided $p \leqq \log_2 n$. Multiplication and division by 2 is necessary in this algorithm. This is to be compared with pn^2 for the standard algorithm and $\simeq p^{.58}n^2 + o(n^2)$ for an algorithm of Hopcroft and Kerr [1] which, however, requires no multiplication and division by 2.

Key words. matrix multiplication, complexity theory

1. $n \times 2$ by $2 \times n$ matrix multiplication. We consider here multiplication of $n \times 2$ by $2 \times n$ matrices. Our notation is set by the equation

$$\begin{bmatrix} x_1 & x_{n+1} \\ x_2 & x_{n+2} \\ \vdots & \vdots \\ x_n & x_{2n} \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_n \\ y_{n+1} & y_{n+2} & \cdots & y_{2n} \end{bmatrix}$$

$$= \begin{bmatrix} x_1y_1 + x_{n+1}y_{n+1} & x_1y_2 + x_{n+1}y_{n+2} \cdots \\ x_2y_1 + x_{n+2}y_{n+1} & x_2y_2 + x_{n+2}y_{n+2} \cdots \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ x_ny_1 + x_{2n}y_{n+1} & x_ny_2 + x_{2n}y_{n+2} \cdots \end{bmatrix}.$$

We now assume that $n = r^p$ for positive integers r and p. We define the vectors $x^1 = (x_1, x_2, \cdots, x_n), x^2 = (x_{n+1}, x_{n+2}, \cdots, x_{2n}), y^1 = (y_1, y_2, \cdots, y_n)$ and $y^2 = (y_{n+1}, y_{n+2}, \cdots, y_{2n})$ and index their components by p-digit r-ary numbers $(i, j, \cdots, k)$. Consider the set

$$L = \{(\alpha, \beta) : \alpha \in \mathbb{Z}, \beta \in \mathbb{Z}, 0 \leqq \alpha, \beta \leqq n - 1\}.$$

We define an equivalence relation in L according to $(\alpha, \beta) \sim (\gamma, \delta)$ if, in terms of the r-ary expansions of $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_p)$, $\beta = (\beta_1, \beta_2, \cdots, \beta_p)$, $\gamma = (\gamma_1, \gamma_2, \cdots, \gamma_p)$ and $\delta = (\delta_1, \delta_2, \cdots, \delta_p)$, we have for each i, $\{\alpha_i, \beta_i\} = \{\gamma_i, \delta_i\}$. The equivalence class that contains (α, β) will be denoted by $[(\alpha, \beta)]$. Notice that $(\alpha, \beta) \sim (\beta, \alpha)$. Define the *weakly diagonal* subset of L by

$$D = \{(\alpha, \beta) : \alpha_i = \beta_i \text{ for some } i, 1 \leqq i \leqq p\}.$$

* Received by the editors December 27, 1974, and in revised form November 10, 1975. This work was supported in part by the U.S. Office of Naval Research under the Joint Services Electronics Program under Contract N00014-67-A-0298-0006, and by the National Science Foundation under Grant GJ-43157.

† Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts 02138.

‡ Department of Computer Science, Yale University, New Haven, Connecticut 06520.

There are at most 2^p elements in an equivalence class, and if $(\alpha, \beta) \notin D$, there are exactly 2^p elements in the equivalence class $[(\alpha, \beta)]$.

We observe that for $p = 1$, the weakly diagonal set is merely the set of diagonal elements $\{(i, i) | 1 \leqq i \leqq n\}$, and a pair of elements belong to the same equivalence class if and only if they are transposes of each other (i.e., $(j, i) \in [(i, j)]$). For larger values of p, these definitions are extended to p-tuples. The total number of equivalence classes with 2^p elements is $r^p(r-1)^p 2^{-p}$; that is, to define an equivalence class, we pick a p-tuple $(i, j, \cdots, k)$ with elements from $0, 1, \cdots, r-1$ and a second p-tuple $(l, m, \cdots, n)$ with elements from $0, 1, \cdots, r-1$, subject to the restriction that the qth digit of the second p-tuple is not equal to the qth digit of the first p-tuple, and then acknowledge that there are the 2^p possible interchanges which give pairs defining the same equivalence class. There are $n^2 = r^{2p}$ elements in L, and in D we have $(r^{2p} - r^p(r-1)^p) = pr^{2p-1} - [p(p-1)/2]r^{2p-2} + \cdots \pm r^p$, as a simple combinatorial argument shows. (To choose an element *not* in D, there are r^p ways to choose the α and $(r-1)^p$ ways to choose the β given a choice of α. Thus the cardinality of D is $r^{2p} - r^p(r-1)^p$.)

Suppose that (α, β) is not weakly diagonal. Consider

$$\overline{\alpha\beta} \stackrel{\text{def}}{=} \{\gamma : (\gamma, \delta) \in [(\alpha, \beta)] \text{ for some } \delta\}.$$

Otherwise stated, $\overline{\alpha\beta}$ is the set of p-tuples whose ith entry equals either α_i or β_i. Clearly $\overline{\alpha_1\beta_1} = \overline{\alpha_2\beta_2}$ if $(\alpha_1, \beta_1) \backsim (\alpha_2, \beta_2)$. Define a function f on $\overline{\alpha\beta}$ by

$$f_{\overline{\alpha\beta}} = \left(\sum_{\gamma \in \overline{\alpha\beta}} x_\gamma^1 + x_\gamma^2\right)\left(\sum_{\gamma \in \overline{\alpha\beta}} y_\gamma^1 + y_\gamma^2\right),$$

and define g for $(\alpha, \beta) \notin D$ by

$$g_{\alpha\beta} = \left(\sum_{\gamma \in \overline{\alpha\beta}} x_\gamma^1 + x_\gamma^2 - 2x_\alpha^2\right)\left(\sum_{\gamma \in \overline{\alpha\beta}} y_\gamma^1 + y_\gamma^2 - 2y_\beta^1\right).$$

Observe that there is one f and 2^p g's for each equivalence class defined by a nondiagonal element of L.

The reason for this choice of f and g is expressed by the following identity:

$$\tfrac{1}{2}[f_{\overline{\alpha\beta}} - g_{\alpha\beta}] = x_\alpha^1 y_\beta^1 + x_\alpha^2 y_\beta^2 - x_\alpha^2\left(\sum_{\gamma \in \overline{\alpha\beta}} y_\gamma^1 + y_\gamma^2 - y_\beta^1 - y_\beta^2\right) - \left(\sum_{\gamma \in \overline{\alpha\beta}} x_\gamma^1 + x_\gamma^2 - x_\alpha^1 - x_\alpha^2\right) y_\beta^1,$$

which may be rewritten as

$$(*) \qquad x_\alpha^1 y_\beta^1 + x_\alpha^2 y_\beta^2 = \tfrac{1}{2}[f_{\overline{\alpha\beta}} - g_{\alpha\beta}] + x_\alpha^2 \sum_{\substack{\gamma \in \overline{\alpha\beta} \\ \gamma \neq \beta}} (y_\gamma^1 + y_\gamma^2) + \sum_{\substack{\gamma \in \overline{\alpha\beta} \\ \gamma \neq \alpha}} (x_\gamma^1 + x_\gamma^2) y_\beta^1.$$

Now observe—and it is this point that makes the algorithms work—that the only γ in $\overline{\alpha\beta}$ such that (γ, β) does not belong to D is $\gamma = \alpha$; otherwise γ agrees with β in one or more places and $(\gamma, \beta) \in D$.

We see that the matrix product can be evaluated by using (*) to compute the elements which are not weakly diagonal and using naive methods for the weakly

diagonal elements. That is, we compute $x_\alpha^1 y_\beta^1$, $x_\alpha^2 y_\beta^1$ and $x_\alpha^2 y_\beta^2$ for $(\alpha, \beta) \in D$ and then compute all the f's and g's. The $x_\alpha^1 y_\beta^2$ terms need not be computed since they do not appear. To carry this out takes $r^p(r-1)^p(1+2^{-p})$ multiplications for the f's and g's plus $3[r^{2p} - r^p(r-1)^p]$ multiplications for the diagonals. Consequently there are

$$\begin{aligned} d &= \{r^p(r-1)^p\}\{1+2^{-p}\} + 3(r^{2p} - r^p(r-1)^p) \\ &= r^p[3r^p - (2-2^{-p})(r-1)^p] \end{aligned}$$

multiplications in such a method. In terms of $n = r^p$ and p, this becomes

$$\begin{aligned} d &= n[3n - (2-2^{-p})(\sqrt[p]{n}-1)^p] \\ &= n\left[3n - (2-2^{-p})\left(n - \binom{p}{1}n^{1-1/p} + \cdots \pm 1\right)\right] \\ &= n^2[1 + 2^{-p} + (2-2^{-p})pn^{-1/p} + \cdots]. \end{aligned}$$

Thus we see that if we chose p as a function of n in such a way as to have p go to infinity with n going to infinity but at such a rate as to have $p/\sqrt[p]{n}$ go to zero, then this algorithm will require $n^2 + o(n^2)$ multiplications. Suitable choices for p for these constraints would be $p^p = \sqrt{n}$ or $2^{p^2} = n$; i.e., $p = \sqrt{\log_2 n}$.

For p small, on the other hand, we may compare this with known results. Thus for $p = 1$, we have $d = \frac{3}{2}n^2 + \frac{3}{2}n$, which is not as good as Hopcroft and Kerr's $\frac{3}{2}n^2 + n/2$. But with $p = 2$, we have

$$d = \tfrac{5}{4}n^2 + 3\tfrac{1}{2}n^{3/2} - \tfrac{7}{4}n$$

which is smaller than $\frac{3}{2}n^2 + n/2$ if n is larger than 196.

Now consider values of n which are not of the form r^p. Clearly any algorithm which multiplies $n \times 2$ by $2 \times n$ matrices also multiplies $m \times 2$ by $2 \times m$ matrices if $m \leqq n$. Thus the burden here is to show that there exists for each n a choice of r and p such that $\tilde{n}(n) = r^p$ upper bounds n with

$$\lim_{n \to \infty} (\tilde{n}(n)/n) = 1$$

so that the asymptotic expansions given above remain valid. Consider the choice

$$p = \lfloor (\log_2 n)^{1/2} \rfloor \quad \text{and} \quad r = \lceil n^{1/\lfloor (\log_2 n) \rfloor^{1/2}} \rceil$$

where $\lfloor\ \rfloor$ and $\lceil\ \rceil$ denote the "largest integer less than or equal to" and "smallest integer larger than or equal to", respectively. We see that

$$\begin{aligned} r^p/n &\leqq n^{-1}(n^{1/\lfloor (\log_2 n)^{1/2} \rfloor} + 1)^{\lfloor (\log_2 n)^{1/2} \rfloor} \\ &\leqq 1 + \lfloor (\log_2 n)^{1/2} \rfloor n^{-1/\lfloor (\log_2 n)^{1/2} \rfloor} + \cdots, \end{aligned}$$

where the dots indicate lower order terms. Now by examining the logarithm of the second term, we see that it does indeed go to zero as n goes to infinity. Thus, with the above choice of p and r, we have an $n^2 + o(n^2)$ algorithm for all n.

2. The main result. It is well known that any algorithm not utilizing commutativity for multiplying $n \times 2$ by $2 \times n$ matrices can be extended to yield an

algorithm for multiplying $n^t \times 2^t$ by $2^t \times n^t$ matrices merely by embedding the original algorithm in itself t times. If the original algorithm takes m multiplications, the new one takes m^t. Thus, using the algorithm of the previous section, we observe that $n^t[3n - (2 - 2^{-p})(\sqrt{n} - 1)^p]^t$ multiplications suffice for multiplying $n^t \times 2^t$ by $2^t \times n^t$ matrices. With a little extra work, we may establish the following slightly better result.

THEOREM. *There is an algorithm for multiplying an $n \times r(n)$ matrix by an $r(n) \times n$ matrix which requires only $n^2 + o(n^2)$ multiplications, provided that*

$$\lim_{n\to\infty} \frac{\log_2 r(n)}{\log_2 n} = 0.$$

Proof. Let $d(M_{n,r(n),n})$ be the number of multiplications required. Let $q = (\log_2 r(n))^{-1}$. Then, by the observation above,

$$d(M_{n,r(n),n}) \leqq [d(M_{n^q,2,n^q})]^{1/q} = \tilde{d}^{1/q},$$

and for a choice of $p = \lfloor\sqrt{\log_2 n/\log_2 r(n)}\rfloor$, the algorithm of the previous section shows that $\tilde{d}$ grows as $n^{2/\log_2 r(n)} + o(n^{2/\log_2 r(n)})$ with n.

These results are surprising since for the naive algorithm, the result is rn^2, and for the Hopcraft–Kerr algorithm, the result is $\sim r^{.58}n^2$. We have shown that the rate of growth for n large is independent of r and in fact equal to n^2. The algorithms presented here are clearly asymptotically optimal—a fact which puts the known lower bounds on the number of multiplications required in a new perspective. In particular, we have shown that $d(M_{n,n,\log_2 n}) = n^2 + o(n^2)$, which makes finding a nonlinear lower bound on the multiplication of square matrices appear to be an even more difficult task. It is interesting to note that while we have not computed the total multiplication requirements, the only other multiplications which occur here involve either positive or negative powers of 2 and can thus be done in a binary machine by shifting.

3. The commutative case. If we assume that $x_i y_j = y_j x_i$, then we can do better insofar as the $(n \times 2)$ by $(2 \times n)$ matrix multiplication is concerned. In this case, Winograd [2] observed that $n^2 + 2n$ multiplications are sufficient. However, it is possible to reduce this to $n^2 + 2n - 1$, as Waksman has indicated [3].

The notation is as above. Define n^2 functions

$$h_{ij}(x, y) = (x_i^1 + y_j^2)(y_j^1 + x_i^2) = x_i^1 y_j^1 + y_j^2 x_i^2 + y_j^2 y_j^1 + x_i^1 x_i^2, \qquad 1 \leqq i, \quad j \leqq n.$$

Define n functions as

$$l_{ii}(x, y) = (x_i^1 - y_i^2)(y_i^1 - x_i^2) = x_i^1 y_i^1 + y_i^2 x_i^2 - y_i^2 y_i^1 - x_i^1 x_i^2, \qquad 1 \leqq i \leqq n,$$

and define $n - 1$ functions

$$m_i(x, y) = (x_1^1 - y_i^2)(y_i^1 - x_1^2) = x_1^1 y_i^1 + y_i^2 x_1^2 - x_1^1 x_1^2 - y_i^2 y_i^1, \qquad 2 \leqq i \leqq n.$$

Now we see that

$$\varphi_{ii}(x, y) = \tfrac{1}{2}(h_{ii}(x, y) - l_{ii}(x, y)) = y_i^2 y_i^1 + x_i^1 x_i^2, \qquad 1 \leqq i \leqq n,$$

and that

$$\psi_{ii}(x, y) = \tfrac{1}{2}\{h_{1i}(x, y) - m_i\} = y_i^2 y_i^1 + x_1^1 x_1^2, \qquad 2 \leqq i \leqq n,$$
$$\psi_{11}(x, y) = \tfrac{1}{2}\{h_{11}(x, y) - l_{11}(x, y)\} = x_1^1 x_1^2 + y_1^1 y_1^2.$$

Thus

$$x_i^1 y_j^1 + x_i^2 y_j^2 = h_{ij}(x, y) - \varphi_{ii}(x, y) + \psi_{ii}(x, y) - \psi_{jj}(x,y), \qquad 1 \leqq i, \quad j \leqq n,$$

and we can compute the matrix product with $n^2 + 2n - 1$ multiplications.

Notice that this permits the multiplication of 4×4 matrices in 46 multiplications and 3×3 in 23, as opposed to 49 and 24, respectively, by the best noncommutative algorithms now known.[1]

If we now imbed this algorithm in itself t times, we obtain a procedure for evaluating a family of bilinear n^{2t} bilinear forms. This family has many formal similarities with the family of bilinear forms which occurs in $n^t \times 2^t$ by $2^t \times n^t$ matrix multiplication. It requires $(n^2 + 2n - 1)^t$ multiplications. In terms of $m = n^t$, we have a family of bilinear forms with the basic features of $m \times r$ by $r \times m$ matrix multiplication and which requires $m^2 + tm^{2-1/t} + \cdots$ multiplications. The significant point is that successive applications of the commutative algorithm which is asymptotically very near to the bounds given in § 2 for matrix multiplication, thus suggesting a limit on the kind of improvement one can expect using commutativity.

REFERENCES

[1] J. HOPCROFT AND L. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[2] S. WINOGRAD, *A new algorithm for inner product*, IEEE Trans. Computers, C-17 (1968), pp. 693–694.

[3] A. WAKSMAN, IEEE Trans. Computers, C-19, pp. 360–361.

[1] We have recently received a manuscript of J. Laderman which gives an algorithm for noncommutative 3×3 matrix multiplication using 23 multiplications.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

GENERAL RESULTS ON TOUR LENGTHS IN MACHINES AND DIGRAPHS*

TAKAO ASANO,† MICHIRO SHIBUI† AND ITSUO TAKANAMI‡

Abstract. A tour in a sequential machine is a shortest input sequence taking the machine from some initial state, through all of its remaining states and back again into its initial state. A. K. Dewdney and A. L. Szilard [2] found the best upper bound for tour length for two classes of machines: the class of n-state sequential machines with unrestricted input alphabet and the class of n-state sequential machines with a two-letter input alphabet. We define a strong circulation on a strong digraph D whose volume is equal to the tour length of D. Characterizing the volume of strong circulation, we derive the best upper bound for tour length for n-state sequential machines with an r-letter input alphabet. Putting $r = 2$ or $r = \infty$, we have the same results as those obtained by A. K. Dewdney and A. L. Szilard.

Key words. directed graph, spanning walk, Hamiltonian cycle, finite state machine, diagnosis sequence, tour, strong circulation

1. Introduction. A tour in a sequential machine is a shortest input sequence taking the machine from some initial state, through all of its remaining states and back again into its initial state. A. K. Dewdney and A. L. Szilard [2] introduced the concepts of a tour in a machine and a circulation on a strong digraph and found the best tour length upper bound for two classes of machines: the class of n-state sequential machines with unrestricted input alphabet and the class of n-state sequential machines with a two-letter input alphabet. They also suggested that a similar analysis would help to prove analogous results for the class of n-state machines with an r-letter input alphabet.

In this paper, we show that their suggestion works. However, we have found a problem with their argument. Namely, they said in [2] that their Theorem 3 implies that the question "what is the maximum tour length over all strong digraphs in $\mathscr{D}$?" can be replaced by the question "what is the maximum volume of circulation over all circulation digraphs in $\mathscr{D}$?". In the next section however, we show that if the support of circulation on a strong digraph D is not strong, the tour length of D is greater than the volume of circulation. So we define a strong circulation on D whose volume is equal to the tour length of D. Then we show that the question "what is the maximum tour length over all strong digraphs in $\mathscr{D}$?" can be replaced by the question "what is the maximum volume of strong circulation over all strong circulation digraphs in $\mathscr{D}$?". Characterizing the volume of strong circulation, we derive the best upper bound for tour length for the class of n-state sequential machines with an r-letter input alphabet. Putting $r = 2$ or $r = \infty$, we have the same results as those obtained by A. K. Dewdney and A. L. Szilard.

* Received by the editors June 25, 1974, and in final revised form November 20, 1975.
† Department of Electrical Communications, Tohoku University, Sendai, 980 Japan.
‡ Department of Electrical Engineering, Yamaguchi University, Ube, 755 Japan.

2. Strong circulation and tour length. This section follows the terminology of [2] and [3]. A *digraph* D consists of a finite set $V(D)$ of points and a collection $X(D)$ of ordered pairs of distinct points. Any such pair $x = (u, v)$ is called an *arc* or *directed line* and will usually be denoted uv. The arc $x = uv$ goes from u to v and is *incident* with u and v. We denote $x^+ = u$ and $x^- = v$. Points u and v are called the *initial* and *final* points of x, respectively. We also say that u is *adjacent to* v and v is *adjacent from* u. The *outdegree* $d^+(v)$ of a point v is the number of points adjacent from it, and the *indegree* $d^-(v)$ is the number of points adjacent to it.

A (*directed*) *walk* W in a digraph D is an alternating sequence of points and arcs, $v_0, x_1, v_1, \cdots, x_n, v_n$ in which each arc x_i is $v_{i-1}v_i$. In particular, $n = 0$ implies W is a walk consisting of only one point. The *length* of such a walk W is n, the number of occurrences of arcs in it, and is denoted by $l(W)$. A *closed walk* is a walk with the same first and last points, and a *spanning walk* is a walk containing all the points. A *circuit* is a nontrivial closed walk with all points distinct (except the first and last).

A *tour* is a closed spanning walk of minimum length. D will be called a *circuit digraph* when its tour is a circuit. If there is a walk from u to v, then v is said to be *reachable* from u. A *semiwalk* is again an alternating sequence $v_0, x_1, v_1, \cdots, x_n, v_n$ of points and arcs, but each arc x_i may be either $v_{i-1}v_i$ or v_iv_{i-1}. A digraph is *strongly connected*, or *strong*, if every two points are mutually reachable; it is *weakly connected*, or *weak*, if every two points are joined by a semiwalk. Clearly, a digraph has a tour if and only if it is strong. The *maximum tour problem* for a class $\mathscr{D}$ of strong digraphs is the problem of finding the least upper bound for tour length over all members of $\mathscr{D}$.

Let D be a strong digraph. A *semi-chain* f on D is a function on the arcs of D into the nonnegative integers. Equality and sums for semi-chains are defined as for functions; $f \geqq g$ denotes that there is a semi-chain h such that $f = g + h$. Obviously, such an h is unique and is denoted by $f - g$. Furthermore, when $h = 0$, the *trivial semi-chain*, then $f = g$. Otherwise $f > g$. The subgraph $|f|$ of D, called the *support* of f, consists of all the arcs $x \in X(D)$ for which $f(x) > 0$, along with their initial and final points. The *volume* of a semi-chain f is the number $\sum_{x \in X(D)} f(x)$ and is denoted by $\mathrm{vol}(f)$.

A circulation introduced in [2] by A. K. Dewdney and A. L. Szilard is a semi-chain f on D which satisfies the following conditions:

C1: $\sum_{x^+ = v} f(x) = \sum_{x^- = v} f(x)$ for all $v \in V(D)$,

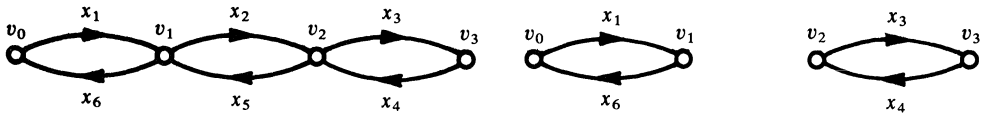
C2: $V(|f|) = V(D)$,

C3: $\mathrm{vol}(f)$ is a minimum over all semi-chains which satisfy conditions C1 and C2.

A nontrivial semi-chain c on D is a *circuit semi-chain* if $|c|$ is a circuit subdigraph in D and $c \leqq 1$, where 1 denotes the function $:X(D) \to \{1\}$, of course. Note that there is a 1–1 correspondence between the set of circuit subdigraphs in D and the set of circuit semi-chains on D.

The following example shows that the volume of a circulation on a strong digraph does not necessarily coincide with the length of its tour: a semi-chain f on a strong digraph Δ_1 such that $f(x_i) = 1$ for $i = 1, 3, 4, 6$ and $f(x_j) = 0$ for $j = 2, 5$ is clearly a circulation on Δ_1. The volume of f is 4, but the tour length of

Δ_1 is clearly 6. See Fig. 1. So we define a strong circulation whose volume exactly agrees with the tour length.



A strong digraph Δ_1 *The support* $|f|$ *of* f

FIG. 1

Let D be a strong digraph. A semi-chain f on D is a *strong circulation* if we have the following conditions:

S1: $\sum_{x^+=v} f(x) = \sum_{x^-=v} f(x)$ for all $v \in V(D)$,

S2: $V(|f|) = V(D)$,

S3: $|f|$ is weak,

S4: $\mathrm{vol}(f)$ is a minimum over all semi-chains which satisfy conditions S1, S2 and S3.

Clearly, if a semi-chain f satisfies the conditions S1 and S3, then its support $|f|$ is strong. Therefore the condition S3 may be replaced by the condition

S3′: $|f|$ is strong.

Conditions S1 and S2 make it possible to extend f to the points of D by defining $f(v) = \sum_{x^+=v} f(x)$.

A semi-chain g on the foregoing digraph Δ_1 such that $g(x_i) = 1$ for all x_i is clearly a strong circulation. Therefore, a strong circulation is not necessarily a circulation. Furthermore, since a circulation satisfies the conditions S1 and S2 but S3 is not implied by these conditions, the volume given by S4 is not less than that given by C3. The following theorem shows that the volume of a strong circulation on a digraph is equal to the length of its tour.

For any walk W in a digraph D, the *semi-chain* f_W on D is defined to have the value $f_W(x)$ equal to the number of times the arc x appears in W. Consequently $l(W) = \mathrm{vol}(f_W)$ holds.

THEOREM 1. *Let D be a strong digraph and f a semi-chain on D. f is a strong circulation on D if and only if there is a tour T on D such that $f = f_T$.*

Proof. Let f be a strong circulation on D. Since the support $|f|$ of f is strong and f satisfies the conditions S1 and S2, it can readily be shown by induction on the number of points of D that there is a closed walk T in $|f|$ in which the arc x appears $f(x)$ times and every point of D also appears [3, p. 204]. Clearly, $f_T = f$. Assume T is not a tour on D but T^* is a tour. Then we have

$$l(T^*) < l(T). \tag{1}$$

Since the semi-chain f_{T^*} corresponding to T^* satisfies the conditions S1, S2 and S3,

$$l(T^*) = \mathrm{vol}(f_{T^*}) \geqq \mathrm{vol}(f) = l(T). \tag{2}$$

But (1) contradicts (2), and our assumption is invalid. Therefore T is a tour on D.

For the proof of the "if" part, let T be a tour on D and let f_T be a semi-chain corresponding to T. Assume f_T is not a strong circulation on D and assume f is a strong circulation. Since the function f_T satisfies the conditions S1, S2 and S3,

$$\text{vol}(f) < \text{vol}(f_T). \tag{3}$$

By the foregoing portion of this proof, there exists a tour T' corresponding to f such that $f_{T'} = f$. Since both T and T' are tours,

$$\text{vol}(f) = l(T') = l(T) = \text{vol}(f_T). \tag{4}$$

We have a contradiction from (3) and (4). Hence f_T is a strong circulation on D. □

A strong digraph D is called a *strong circulation digraph* if D has a strong circulation f for which $|f| = D$. Since the volume of a strong circulation f on D is equal to the volume of a strong circulation on $|f|$, it may be assumed that D is a strong circulation digraph. Then Theorem 1 says that the question "what is the maximum tour length over all strong digraphs in $\mathscr{D}$?" can be replaced by the question "what is the maximum volume of strong circulation over all strong circulation digraphs in $\mathscr{D}$?". The volume of a strong circulation on a digraph with only one point is clearly 0. In what follows, unless stated otherwise, we assume that a digraph has at least two points.

Let f' be a semi-chain on a strong subdigraph D' of a strong digraph D. Furthermore, let f be the semi-chain on D such that $f(x) = f'(x)$ for $x \in X(D')$ and 0 otherwise. Then f is called the *extension* of f' to D or $X(D)$. On the other hand, let g be a semi-chain on D and g' the semi-chain on a strong subdigraph D' of D such that $g'(x) = g(x)$ for $x \in X(D')$. Then g' is called the *restriction* of g to D' or $X(D')$.

THEOREM 2. *Let D be a strong circulation digraph, and let f be a strong circulation on D such that $|f| = D$. Then there is a circuit semi-chain c on D such that either $f = c$ or $|f - c|$ is a strong circulation digraph and $(f - c)'$ is a strong circulation, where $(f - c)'$ is the restriction of $f - c$ to $|f - c|$.*

Proof. Theorem 1 shows that for any strong circulation f on D, there is a tour T on D to which f corresponds. Let $T = v_0x_1v_1 \cdots x_nv_n$ and $v_0 = v_n$. Then there exist i and j, $0 \leqq i < j \leqq n$, for which $v_i = v_j$ and $v_k \neq v_l$ for any k and l, $i < k < l < j$. Hence, $W = v_ix_{i+1} \cdots v_j$ forms a circuit to which a circuit semi-chain c corresponds. If $i = 0$ and $j = n$, then $f = c$. If either $i \neq 0$ or $j \neq n$, then $T' = v_0x_1 \cdots v_ix_jv_{j+1} \cdots v_n$ is a closed walk and $f_{T'} = f - c$. Therefore, the support $|f - c|$ of $f - c$ is strong. Let $\bar{f}$ be a strong circulation on $|f - c|$ and $\bar{f}^*$ the extension of $\bar{f}$ to D. Since $\bar{f}^* + c$ satisfies S1, S2 and S3 on D,

$$\text{vol}(\bar{f}) + \text{vol}(c) = \text{vol}(\bar{f}^* + c) \geqq \text{vol}(f). \tag{5}$$

On the other hand, let $(f - c)'$ be the restriction of $f - c$ to $|f - c|$. Then

$$\text{vol}(\bar{f}) \leqq \text{vol}((f - c)') = \text{vol}(f) - \text{vol}(c). \tag{6}$$

From (5) and (6), we have vol. $(\bar{f}) = \text{vol}((f - c)')$. Therefore $(f - c)'$ is a strong circulation on $|f - c|$. □

THEOREM 3. *Let f be a strong circulation on a strong circulation digraph D, and let $|f| = D$. Then f may be decomposed into a sum of circuit semi-chains*

$$f = c_1 + c_2 + \cdots + c_n.$$

Proof. Using Theorem 2 repeatedly, we can readily prove the theorem. □

It was shown in [2] that any decomposition of a circulation satisfying the hypothesis of our Theorem 3 has the property that each circuit digraph $|c_i|$ contains a point not in any other circuit digraph $|c_j|$. But the following example shows that such a decomposition of a strong circulation does not necessarily have the same property as that of a circulation. Consider the foregoing digraph Δ_1. The function g such that $g(x_i) = 1$ for all x_i is a strong circulation. Let $c_1(x_i) = 1$ if $i = 1, 6$ and 0 otherwise. Let $c_2(x_i) = 1$ if $i = 2, 5$ and 0 otherwise. Let $c_3(x_i) = 1$ if $i = 3, 4$ and 0 otherwise. Then $g = c_1 + c_2 + c_3$. But $V(|c_2|) \subseteq V(|f - c_2|)$. Note that $f - c_2$ is a circulation on Δ_1 but $|f - c_2|$ is not strong.

Let f be a strong circulation on D, and let $f = c_1 + c_2 + \cdots + c_n$ be a decomposition of f into a sum of circuit semi-chains. $|f - c_i|$ is not necessarily strong. Let $D_1, \cdots, D_t$ be the maximal strong components of $|f - c_i|$. Then it can be seen that $i \neq j$ implies that $V(D_i) \cap V(D_j) = \varnothing$ and $X(D_i) \cap X(D_j) = \varnothing$.

LEMMA 1. *Let f be a strong circulation on a strong circulation digraph D and let $|f| = D$. Let $f = c_1 + c_2 + \cdots + c_n$ be a decomposition of f into a sum of circuit semi-chains. Let $f - c_n \neq 0$ and $D_1, \cdots, D_t$ be the maximal strong components of $|f - c_n|$. Let f_i be the restriction of $f - c_n$ to $X(D_i)$. Then $|f_i| = D_i$ and f_i is a strong circulation on D_i. Thus D_i is a strong circulation digraph.*

Proof. Let f_i^* be the extension of f_i to D. Then $f - c_n$ can be expressed as a sum of f_i^*'s: $f - c_n = f_1^* + f_2^* + \cdots + f_t^*$. Since $f_i^*(x) = (f - c_n)(x) > 0$ if $x \in X(D_i)$ and 0 otherwise, $|f_i^*| = |f_i|$ contains every arc and point of D_i but none of any other component D_j. Therefore $|f_i| = D_i$. Since $\sum_{x^+ = v} f_i(x) = \sum_{x^- = v} f_i(x)$ for any point v of each D_i and $V(|f_i|) = V(D_i)$, and $|f_i|$ is strong, f_i satisfies the conditions S1, S2 and S3 on D_i. Let $\bar{f}_i$ be a strong circulation on D_i. Then

$$\text{vol}(\bar{f}_i) \leqq \text{vol}(f_i). \tag{7}$$

Let $\bar{f}_i^*$ be the extension of $\bar{f}_i$ to D. Clearly, $\text{vol}(\bar{f}_i) = \text{vol}(\bar{f}_i^*)$. Let $\bar{f} = \bar{f}_1^* + \bar{f}_2^* + \cdots + \bar{f}_t^* + c_n$. $\bar{f}$ clearly satisfies the conditions S1, S2 and S3 on D. Since f is a strong circulation on D,

$$\text{vol}(f) \leqq \text{vol}(\bar{f}) = \text{vol}(\bar{f}_1^*) + \cdots + \text{vol}(\bar{f}_t^*) + \text{vol}(c_n). \tag{8}$$

On the other hand,

$$\text{vol}(f) = \text{vol}(f_1^*) + \cdots + \text{vol}(f_t^*) + \text{vol}(c_n). \tag{9}$$

From (7), (8) and (9), $\text{vol}(f_i) = \text{vol}(f_i^*) = \text{vol}(\bar{f}_i^*) = \text{vol}(\bar{f}_i)$. Therefore f_i is a strong circulation on D_i. Since $|f_i| = D_i$, D_i is a strong circulation digraph. □

Let $\mathscr{D}_p^r$ denote the class of strong digraphs with p points in which the out-degree $d^+(v)$ of any point v does not exceed r. Note that $p \geqq 2$ implies $r \geqq 1$.

LEMMA 2. *Let D be a strong circulation digraph in $\mathscr{D}_p^r$. Let f be a strong circulation on D and let $f = c_1 + \cdots + c_n$ $(n \geqq 2)$ be a decomposition of f into a sum of circuit semi-chains. Then there is a point v in $V(D)$ such that*

$$2 \leqq d^+(v) = f(v) \leqq r.$$

Proof. Note that $f(v) \geqq d^+(v)$ for any point v in D since $|f| = D$. Assume no such points exist. Then either $d^+(v) = 1$ for all points v in $V(D)$ or $f(v) > d^+(v)$ for all points v in $V(D)$ such that $d^+(v) \geqq 2$. In the former case, D is a circuit digraph since D is strong. This contradicts that $n \geqq 2$. Hence there is a point u_1 such that its outdegree $d^+(u_1) \geqq 2$. By assumption, $2 \leqq d^+(u_1) \leqq r$ and $f(u_1) > d^+(u_1)$. Construct an infinite sequence $u_1, u_2, u_3, \cdots$ of points such that $f(u_i) \geqq 2$. If u_m is the mth point of the sequence, the $(m+1)$st, u_{m+1}, may be found as follows. Let x_m be the arc leaving u_m on which f is maximum. Since $f(u_m) \geqq 2$, $f(x_m) \geqq 2$ if there is only one arc leaving u_m. If there are two arcs leaving u_m, then $f(u_m) > d^+(u_m)$ by assumption, and $f(x_m) \geqq 2$ in any case. Taking $u_{m+1} = x_m^-$, then $f(u_{m+1}) \geqq 2$. Since D has only a finite number of points, the sequence so formed must contain a circuit, say $u_i, x_{i+1}, u_{i+1}, \cdots, x_j, u_j$. If c is defined as the semi-chain having the value 1 on each arc $x_{i+1}, \cdots, x_j$ of this circuit, and 0 otherwise, then $f - c$ fulfills conditions S1, S2 and S3 and violates the minimality of f. □

THEOREM 4. *Let $D \in \mathscr{D}_p^r$ be a strong circulation digraph with f being a strong circulation, and let $|f| = D$. Let $f = c_1 + c_2 + \cdots + c_n$ be a decomposition of f into a sum of circuit semi-chains such that each circuit digraph $|c_i|$ contains a point v_i not in any other circuit digraph $|c_j|$. Let $v_1, v_2, \cdots, v_p$ be an arrangement of points of $V(D)$ for which $i < j$ implies $f(v_i) \leqq f(v_j)$. Then the following hold*:

(i) *$f(v_i) = 1$ for all i, $1 \leqq i \leqq n$;*

(ii) *for j such that $f(v_{n+j}) = 1$ and $f(v_{n+j+1}) \neq 1$, $f(v_{n+j+k}) \leqq k(r-1) + 1$ for all k, $1 \leqq k \leqq p - (n+j)$.*

Proof. Part (i) is clearly true since each circuit digraph $|c_i|$ contains a point v_i not in any other circuit digraph $|c_j|$ and $f(v_i) = 1$. The proof of part (ii) proceeds by induction on n.

If $n = 1$, $f(v_i) = 1$ for all i, $1 \leqq i \leqq p$, and the statement of the theorem is vacuously true. If $2 \leqq n \leqq r$, the statement of the theorem is also obviously true since $f(v) \leqq n \leqq r$ for all points v.

Suppose the theorem holds for all $(r \geqq)\ n < m$ but not $n = m$. Then it will be shown that we have a contradiction. There exists a strong circulation digraph $D \in \mathscr{D}_p^r$ with f being a strong circulation, which does not satisfy the statement of the theorem. Let $|f| = D$ and $f = c_1 + c_2 + \cdots + c_m$ be a decomposition of f into a sum of circuit semi-chains, where each circuit digraph $|c_i|$ contains a point not in any other circuit digraph $|c_j|$. By Lemma 2 there is a point v such that $r \geqq f(v) = d^+(v) \geqq 2$ since $m \geqq 2$. Let v_{m+j+1} be the point such that $f(v_{m+j}) = 1$ and $f(v_{m+j+1}) = n' > 1$. Then $r \geqq f(v) \geqq n' \geqq 2$. It may be assumed that the (ii) holds for $v_{m+j+1}, \cdots, v_{m+j+k-1}$ but not for v_{m+j+k}. Then $k \geqq 2$ and furthermore,

$$
\begin{aligned}
&f(v_{m+j+q}) \leqq q(r-1) + 1 \quad \text{for all } q, \quad 1 \leqq q \leqq k-1, \\
&f(v_{m+j+q}) > k(r-1) + 1 \quad \text{for all } q, \quad q \geqq k.
\end{aligned} \tag{10}
$$

Then there exist n' circuit semi-chains $c_{i(1)}, \cdots, c_{i(n')}$ such that $c_{i(s)}(v_{m+j+1}) = 1$ for all s, $1 \leqq s \leqq n'$. Without loss of generality, it can be assumed that $c_{i(1)}, \cdots, c_{i(n')}$ are $c_1, \cdots, c_{n'}$. Thus exactly n' circuits $|c_1|, \cdots, |c_{n'}|$ pass through v_{m+j+1} in D. Let $D_1, \cdots, D_t$ be the maximal strong components of $|f - c_1|$. Let f_i be the restriction of $f - c_1$ to $X(D_i)$. By Lemma 1, each D_i is a strong circulation digraph with

f_i a strong circulation and $|f_i| = D_i$. Let $f_i = c_{i(1)} + \cdots + c_{i(h(i))}$ be a decomposition of f_i into a sum of circuit semi-chains for which each $c^*_{i(k)} = c_j$ for some j, $2 \leqq j \leqq m$, where $c^*_{i(k)}$ is the extension of $c_{i(k)}$ to D. Repeating the same process as above, we have maximal strong components $\Gamma_1, \cdots, \Gamma_{t'}$ of $|f - c_1 - c_2 - \cdots - c_{n'-1}|$, where each Γ_i is a strong circulation digraph and the restriction g_i of $g = f - (c_1 + \cdots + c_{n'-1})$ to $X(\Gamma_i)$ is a strong circulation on Γ_i and $|g_i| = \Gamma_i$. Note that only one circuit $|c_{n'}|$ passes through v_{m+j+1} in $|f - c_1 - c_2 - \cdots - c_{n'-1}|$, hence $g(v_{m+j+1}) = 1$. Since $g = c_{n'} + \cdots + c_m$, each g_i^* which is the extension of g_i to D may be a sum of some of these circuit semi-chains, $g = g_1^* + \cdots + g_{t'}^*$ and $N(1) + \cdots + N(t') = m - n' + 1$, where g_i^* is a sum of $N(i)$ circuit semi-chains. For any $k' \geqq k$,

$$g(v_{m+j+k'}) \geqq f(v_{m+j+k'}) - (n' - 1) > k(r-1) + 1 - (n' - 1) \geqq (k-1)(r-1) + 1, \tag{11}$$

since $n' \leqq r$. Let u be a point of the set $\{v_{m+j+k'} | k' \geqq k\}$ in which $g(u)$ is the smallest. From (11),

$$g(u) > (k-1)(r-1) + 1. \tag{12}$$

Without loss of generality, we can assume that u is contained in Γ_1. Let

$$w_1, \cdots, w_{N(1)}, w_{N(1)+1}, \cdots, w_{p_1} \tag{*}$$

be an arrangement of points of $V(\Gamma_1)$ such that $i < j$ implies $g_1(w_i) \leqq g_1(w_j)$ and the point u appears before all points v such that $g_1(v) = g_1(u)$, where p_1 is the number of points of Γ_1. Thus no point v in $\{v_{m+j+k'} | k' \geqq k\}$ appears before u. Therefore the number of points v appearing before u in the sequence (*) such that $g_1(v) \geqq 2$ is at most $k - 2$ since $g(v_{m+j+1}) = 1$. Note that $N(1) < m$ since $n' \geqq 2$. Hence, by hypothesis of induction, the statement of the theorem is true for Γ_1 and

$$g(u) = g_1(u) \leqq (k-1)(r-1) + 1. \tag{13}$$

We have a contradiction that (12) and (13) occur simultaneously. □

LEMMA 3. *Let $D \in \mathscr{D}_p^r$ be a strong circulation digraph with f being a strong circulation, and let $|f| = D$. Let $f = c_1 + c_2 + \cdots + c_n$ be a decomposition of f into a sum of circuit semi-chains such that each circuit digraph $|c_i|$ contains a point not in any other circuit digraph $|c_j|$. Then*

$$p \geqq n + Q(n, r), \qquad Q(n, r) = \left[\frac{n-1}{r-1}\right],$$

where $[x]$ is a largest integer not more than x and $[0/0] = 1$.

Proof. If $r = 1$, then $n = 1$ and $p \geqq 2$, and therefore the lemma holds. Let $r \geqq 2$ and let v_i be a point of $V(|c_i|)$ not in any other $V(|c_j|)$. Note that $f(v_i) = 1$, therefore $d^+(v_i) = d^-(v_i) = 1$, and there is no arc connecting v_i and v_j. It may be easily shown that the subgraph $D - \{v_1, v_2, \cdots, v_n\}$ of D which is obtained by deleting $v_1, \cdots, v_n$ and the arcs incident with them from D is weakly connected. Hence there are at least $p - n - 1$ arcs in $D - \{v_1, \cdots, v_n\}$. Assume that $p = n + Q(n, r) - t$ for some positive integer t. Let Z be the total number of arcs incident from points of $V(D) - \{v_1, \cdots, v_n\}$ in D. Since the outdegree of each

point of $V(D) - \{v_1, \cdots, v_n\}$ is at most r, $Z \leqq (p - n)r = (Q(n, r) - t)r$. n arcs of the Z arcs go to $v_1, \cdots, v_n$, and the other $Z - n$ arcs go to the points of $V(D) - \{v_1, \cdots, v_n\}$. Therefore $p - n - 1 \leqq Z - n$. Since $Z \leqq (Q(n, r) - t)r$,

$$n - 1 + (r - 1)t \leqq Q(n, r)(r - 1). \tag{14}$$

On the other hand, in view of inequalities $n - 1 \geqq Q(n, r)(r - 1)$ and $(r - 1)t \geqq r - 1 > 0$,

$$n - 1 + (r - 1)t > Q(n, r)(r - 1). \tag{15}$$

Now, we have a contradiction: that (14) and (15) hold simultaneously. Therefore $t \leqq 0$, and the theorem holds. □

THEOREM 5. *Under the same conditions as in Theorem* 4, *we have the following*:

(i) $f(v_i) = 1$ *for all* i, $1 \leqq i \leqq n$;

(ii) $f(v_{n+k}) \leqq k(r - 1) + 1$ *for all* k, $1 \leqq k \leqq Q(n, r)$ *if* $Q(n, r) \geqq 1$;

(iii) $f(v_{n+Q(n,r)+k}) \leqq n$ *for all* k, $1 \leqq k \leqq p - n - Q(n, r)$ *if* $p - (n + Q(n, r)) \geqq 1$.

Proof. The theorem follows from Theorem 4 and Lemma 3. □

The difference between the results of Theorem 4 and Theorem 5 is depicted in Fig. 2.

Let $g(n, p, r) = n + \sum_{i=1}^{Q(n,r)} [i(r - 1) + 1] + (p - n - Q(n, r))n$. Then, for a strong circulation f in Theorem 5, we have

$$\text{vol}(f) \leqq g(n, p, r) \leqq h(p, r),$$

where $h(p, r) = \max_n \{g(n, p, r) | 1 \leqq n + Q(n, r) \leqq p\}$.

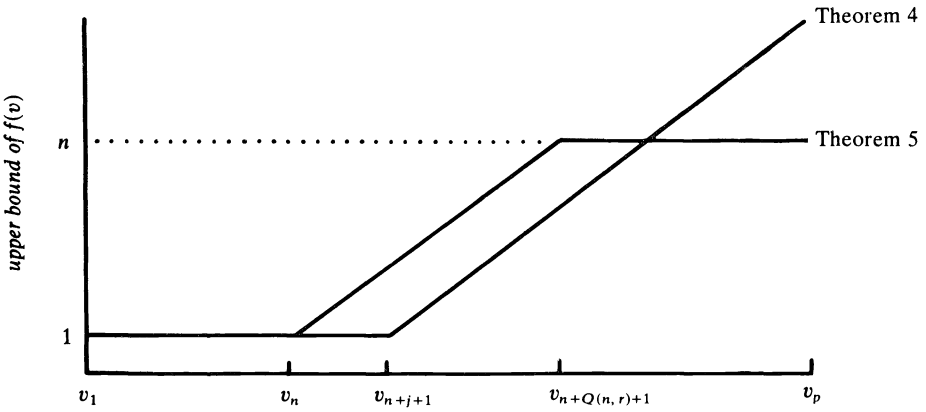


FIG. 2

THEOREM 6. *Let* $p = s(2(r - 1) + 1) + t$, $0 \leqq t \leqq 2(r - 1)$, *and let* $p \geqq 2$ and $r \geqq 1$. *Then* $g(n, p, r)$ *is maximum at* $n = s(r - 1) + 1 + \{(t - 2)/2\}$, *and*

$$h(p, r) = pn + n - n^2 + \tfrac{1}{2}s(s(r - 1) + r + 1 - 2n),$$

where $\{x\}$ *is the smallest integer not less than* x *for* $x > 0$ *and* $\{x\} = 0$ *otherwise.*

Proof. See Appendix A. □

Now we have shown that a tour length of any strong digraph satisfying the conditions of Theorem 4 or 5 is bounded by $h(p, r)$. In what follows, we show that a tour length of any strong digraph which does not satisfy the conditions of Theorem 4 or 5 is also bounded by $h(p, r)$.

LEMMA 4.

(i) $h(q, r) \geqq h(q-1, r) + 2$ *for* $q \geqq 4$ *and* $r \geqq 2$;
(ii) $h(q, 2) \geqq h(q-1, 2) + 3$ *for* $q \geqq 6$;
(iii) $h(q, 2) \geqq h(q-1, 2) + 4$ *for* $q \geqq 9$;
(iv) $h(q, 2) \geqq h(q-2, 2) + 5$ *for* $q \geqq 6$;
(v) $h(q, 2) \geqq h(q-2, 2) + 6$ *for* $q \geqq 7$;
(vi) $h(q, 2) = h(q-3, 2) + q + 1$ *for* $q \geqq 3$.

Proof. See Appendix B. □

LEMMA 5. $h(p_1 + p_2, r) \geqq h(p_1, r) + h(p_2, r) + p_1 + p_2 - 2$ *for* $p_1 \geqq 4$ *and* $p_2 \geqq 5$ *if* $r = 2$, *and for* $p_1 \geqq 3$ *and* $p_2 \geqq 3$ *if* $r \geqq 3$.

Proof. See Appendix C. □

In the foregoing discussion, we have assumed $r \geqq 1$ and the function $h(p, r)$ has been defined at $p \geqq 2$ and $r \geqq 1$. Now we define $h(p, r) = 0$ for $p \leqq 1$. Then we have the main theorem as follows.

THEOREM 7. *Let* $D \in \mathscr{D}_p^r$ *be a strong circulation digraph with* f *being a strong circulation, and let* $|f| = D$. *Then* $\operatorname{vol}(f) \leqq h(p, r)$.

Proof. The proof proceeds by induction on p.

If $p = 1$, the theorem clearly holds since $\operatorname{vol}(f) = 0$.

If $p = 2$, only the digraph Δ_2 (Fig. 3) may be considered as a strong circulation digraph.

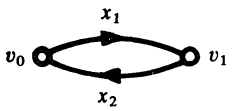
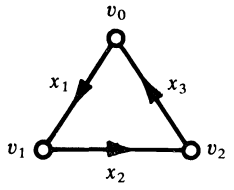


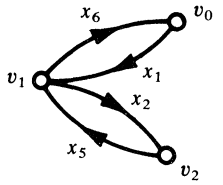
FIG. 3. *A strong circulation digraph* Δ_2

The volume of strong circulation on Δ_2 is clearly 2. On the other hand, $h(2, r) = 2$. Therefore, the theorem holds.

If $p = 3$, only the two digraphs Δ_3 and Δ_4 (Fig. 4) may be considered as strong circulation digraphs.



A strong digraph Δ_3



A strong digraph Δ_4

FIG. 4

The volumes of strong circulations on Δ_3 and Δ_4 are 3 and 4, respectively. On the other hand, $h(3, 1) = 3$ and $h(3, 2) = 4$. Therefore the theorem holds.

Assume the theorem holds for all p less than q for $q \geqq 4$. Let $D \in \mathscr{D}_q^r$ with f being a strong circulation, and let $|f| = D$. If $f = c_1 + c_2 + \cdots + c_n$ is a decomposition of f into a sum of circuit semi-chains such that each circuit digraph $|c_i|$ contains a point not in any other circuit digraph $|c_j|$, we have $\mathrm{vol}(f) \leqq h(q, r)$ by Theorem 5. Hence we consider the other case. Then $r \geqq 2$, and for such a decomposition of f, there is a circuit semi-chain c_i such that $V(|c_i|) \subseteq V(|f - c_i|)$. For such c_i, $|f - c_i|$ is not strong because otherwise $f - c_i$ satisfies the conditions S1, S2 and S3 on D and $\mathrm{vol}(f - c_i) < \mathrm{vol}(f)$ holds, which contradicts the hypothesis that f is a strong circulation. Let $D_1, \cdots, D_t$ be the maximal strong components of $|f - c_i|$. Let f_k be the restriction of $f - c_i$ to $X(D_k)$. Then, by Lemma 1, each D_k is a strong circulation digraph with f_k a strong circulation such that $|f_k| = D_k$, and $f - c_i = f_1^* + \cdots + f_t^*$, where each f_k^* is the extension of f_k to D. $q = q_1 + \cdots + q_t$ holds, where $q_k = \#V(D_k)$, the number of elements of $V(D_k)$. Note that $t \geqq 2$, $V(|f - c_i|) = V(D_1) \cup \cdots \cup V(D_t)$, $V(D_k) \cap V(D_j) = \varnothing$ if $k \neq j$, and each point of $V(|c_i|)$ is contained in some $V(D_j)$ since $V(|c_i|) \subseteq V(|f - c_i|)$. Furthermore, each $V(D_k)$ contains a point u_k not in $V(|c_i|)$ because otherwise $f - f_k^*$ would satisfy the conditions S1, S2 and S3 on D and violate the minimality of f, which contradicts the hypothesis that D is a strong circulation digraph.

Case (i). If there is a D_k for which $q_k = 2$, the D_k and $|f - f_k^*|$ have only one point in common and $|f - f_k^*|$ is also a strong circulation digraph with $(f - f_k^*)'$ a strong circulation, where $(f - f_k^*)'$ is the restriction of $f - f_k^*$ to $|f - f_k^*|$. $\#V(|f - f_k^*|) = q - 1$. Hence, by hypothesis of induction, $\mathrm{vol}(f - f_k^*) \leqq h(q - 1, r)$ and therefore $\mathrm{vol}(f) \leqq h(q - 1, r) + 2$. By $q \geqq tq_k \geqq 4$ and Lemma 4, we have $h(q - 1, r) + 2 \leqq h(q, r)$.

Case (ii). In this case, $r \geqq 3$ and there are no D_k's such that $q_k = 2$. Since $q_k < q$ and each $V(D_k)$ contains a point not in any other $V(D_j)$ and not in $V(|c_i|)$, $\mathrm{vol}(c_i) \leqq q - t$, and by hypothesis of induction, $\mathrm{vol}(f_k) \leqq h(q_k, r)$ for all f_k. Hence

$$\mathrm{vol}(f) = \mathrm{vol}(c_i) + \mathrm{vol}(f_1) + \cdots + \mathrm{vol}(f_t) \leqq q - t + h(q_1, r) + \cdots + h(q_t, r).$$

Then, by Lemma 5, we have $\mathrm{vol}(f) \leqq h(q, r)$ since each $q_k \geqq 3$.

Case (iii). In this case, $r = 2$.

Case (iii-1). There is no D_j for which $q_j = 2$, but there is a D_k for which $q_k = 3$. It suffices to consider the two cases: D_k and $|c_i|$ have in common (a) only one point or (b) two points. Since D_k is a strong circulation digraph, it must be one of the digraphs Δ_3 or Δ_4. The case (a): since D_k and $|c_i|$ has only one point in common, $\#V(|f - f_k^*|) = q - 2$ and clearly, $|f - f_k^*|$ is a strong circulation digraph with $(f - f_k^*)'$ a strong circulation, where $(f - f_k^*)'$ is the restriction of $f - f_k^*$ to $|f - f_k^*|$. Then, by hypothesis of induction, $\mathrm{vol}(f - f_k^*) \leqq h(q - 2, 2)$. Therefore

$$\mathrm{vol}(f) = \mathrm{vol}(f - f_k^*) + \mathrm{vol}(f_k^*) \leqq h(q - 2, 2) + 4.$$

Since each $q_j \geqq 3$ and hence $q \geqq 3t \geqq 6$, we have $h(q - 2, 2) + 4 \leqq h(q, 2)$ by Lemma 4. The case (b): we may assume that there is no such D_k which has only one point in common with $|c_i|$. Suppose $D_k = \Delta_4$. Common points of $|c_i|$ and D_k are either v_0 and v_2 or v_0 and v_1 of Δ_4. Let g be a semi-chain on D such that $g(v_0v_1)$

$= g(v_1v_0) = 1$ and 0 otherwise. Then $f - g$ satisfies the conditions S1, S2 and S3 on D and violates the minimality of f, which contradicts the hypothesis that D is a strong circulation digraph. Therefore the case $D_k = \Delta_4$ can not occur. Let $D_k = \Delta_3$. Then $\#V(|f - f_k^*|) = q - 1$. Furthermore, $(f - f_k^*)'$, the restriction of $f - f_k^*$ to $|f - f_k^*|$, satisfies the conditions S1, S2 and S3 on $|f - f_k^*|$. Assume g is a strong circulation on $|f - f_k^*|$. Then $\mathrm{vol}(g) \leqq \mathrm{vol}(f - f_k^*)$. On the other hand, $\mathrm{vol}(g^* + f_k^*) = \mathrm{vol}(g) + \mathrm{vol}(f_k^*) \geqq \mathrm{vol}(f)$ since $g^* + f_k^*$ satisfies the conditions S1, S2 and S3 on D, where g^* is the extension of g to D. Therefore $\mathrm{vol}(g) = \mathrm{vol}(f - f_k^*)$, and hence $|f - f_k^*|$ is a strong circulation digraph with $(f - f_k^*)'$ a strong circulation. Thus, by hypothesis of induction, we have

$$\mathrm{vol}(f) = \mathrm{vol}(f - f_k^*) + \mathrm{vol}(f_k^*) \leqq h(q - 1, 2) + 3.$$

In this case, $q \geqq 6$ must hold. Then we have $h(q - 1, 2) + 3 \leqq h(q, 2)$ by Lemma 4. That is, $\mathrm{vol}(f) \leqq h(q, 2)$.

Case (iii-2). There is no D_j such that $q_j = 2$ or 3, but there is a D_k such that $q_k = 4$. Since D_k is a strong circulation digraph, D_k is one of the following four digraphs [3]. We consider the three cases: the number of common points of D_k and $|c_i|$ is (a) one, (b) two, or (c) three.

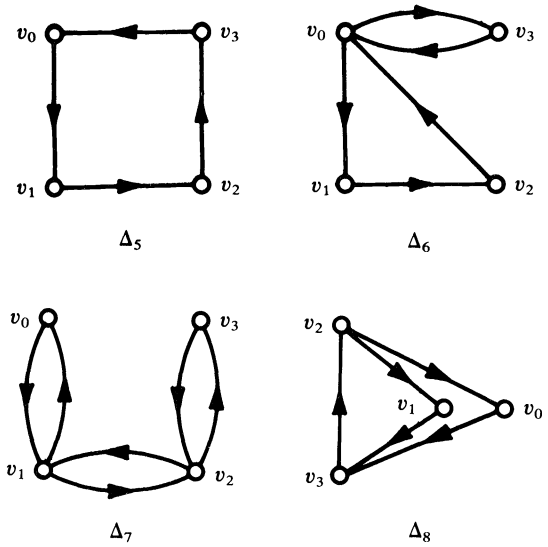


FIG. 5

The case (a): clearly, $|f - f_k^*|$ is a strong circulation digraph with $(f - f_k^*)'$ a strong circulation. Then, by hypothesis of induction, we have

$$\mathrm{vol}(f) = \mathrm{vol}(f - f_k^*) + \mathrm{vol}(f_k^*) \leqq h(q - 3, 2) + 6.$$

Since $q_j \geqq 4$ for all D_j, $q \geqq 8$. Then, by Lemma 4, $h(q - 3, 2) + 6 \leqq h(q, 2)$.

The case (b): $\#V(|f - f_k^*|) = q - 2$. Just as in the case (b) of Case (iii-1), we can show $|f - f_k^*|$ is a strong circulation digraph with $(f - f_k^*)'$ a strong circulation. Therefore

$$\mathrm{vol}(f) = \mathrm{vol}(f - f_k^*) + \mathrm{vol}(f_k^*) \leqq h(q - 2, 2) + 6.$$

Since $q \geqq 8$, by Lemma 4, $h(q-2,2)+6 \leqq h(q,2)$. The case (c): since D is a strong circulation digraph, just as in the case (b) of Case (iii-1) it may be shown that D_k must be Δ_5 (Fig. 5). It may be assumed that the common points of D_k and $|c_i|$ are v_0, v_1, v_2. Then $\#V(|f-f_k^*|)=q-1$. Furthermore, just as in the case (b) of Case (iii-1), we can show $|f-f_k^*|$ is a strong circulation digraph with $(f-f_k^*)'$ a strong circulation. Then, by hypothesis of induction,

$$\mathrm{vol}(f)=\mathrm{vol}(f-f_k^*)+\mathrm{vol}(f_k^*)\leqq h(q-1,2)+4.$$

If $q \geqq 9$, by Lemma 4, $h(q-1,2)+4 \leqq h(q,2)$. If $q=8$, we also have the result that $t=2$, $q_1=q_2=4$ and $D_1=D_k$. Both D_1 and D_2 have three points in common with $|c_i|$ for otherwise this case could be reduced to the case (a) or (b). Therefore, $D_1=D_2=\Delta_5$, and $|c_i|$ contains 6 points. Hence

$$\mathrm{vol}(f)=\mathrm{vol}(f_1)+\mathrm{vol}(f_2)+\mathrm{vol}(c_k)=4+4+6<h(8,2)=17.$$

The above discussion shows that $\mathrm{vol}(f)\leqq h(q,2)$ for Case (iii).

Case (iv). In this case, there is no D_k such that $q_k \leqq 4$. $\mathrm{vol}(f)=\mathrm{vol}(f_1)+\cdots+\mathrm{vol}(f_t)+\mathrm{vol}(c_i)$. Since each $q_j<q$, by hypothesis of induction, $\mathrm{vol}(f_j)\leqq h(q_j,2)$. Therefore

$$\mathrm{vol}(f)\leqq h(q_1,2)+\cdots+h(q_t,2)+q-t, \qquad t\geqq 2.$$

Then by Lemma 5, we have $\mathrm{vol}(f)\leqq h(q,2)$.

Now we have completed the proof of Theorem 7. □

Figure 6 shows a strong digraph $D_p^r \in \mathscr{D}_p^r$ whose tour length is $h(p,r)$. The outdegree of each point v_i, $1 \leqq i \leqq n$, is 1 and each arc incident from them is incident to v_p. If $s=0$, $d^+(v_{n+1})=n$ and the n arcs from v_{n+1} are to $v_1,\cdots,v_n$. If $s \geqq 1$, the outdegree of each point v_{n+k}, $1 \leqq k \leqq s$, is r and $d^+(v_{n+s+1})=\{(t-2)/2\}+1$. The r arcs from v_{n+1} are to $v_1,\cdots,v_r$, and the r arcs from v_{n+k}, $2 \leqq k \leqq s$ are to $v_{n+k-1}, v_{(k-1)(r-1)+2}, v_{(k-1)(r-1)+3},\cdots, v_{k(r-1)+1}$. The $\{(t-2)/2\}+1$ arcs from v_{n+s+1} are to $v_{n+s}, v_{s(r-1)+2}, v_{s(r-1)+3},\cdots, v_n$. For $s \geqq 0$, if $n+s+1<p$, each outdegree of $v_{n+s+2},\cdots,v_p$ is 1, and an arc from v_{n+s+k}, $k \geqq 2$, is to $v_{n+s+k-1}$. The set of arcs in D_p^r consists of only these arcs.

Let f be a strong circulation on D_p^r. Since $V(|f|)=V(D_p^r)$, it can readily be shown that:

$f(v_k)\geqq 1$ for each k, $1 \leqq k \leqq n$,

$f(v_{n+k}) \geqq k(r-1)+1$ for each k, $1 \leqq k \leqq s$,

$f(v_{n+s+k}) \geqq n$ for each k, $1 \leqq k \leqq p-(n+s)$,

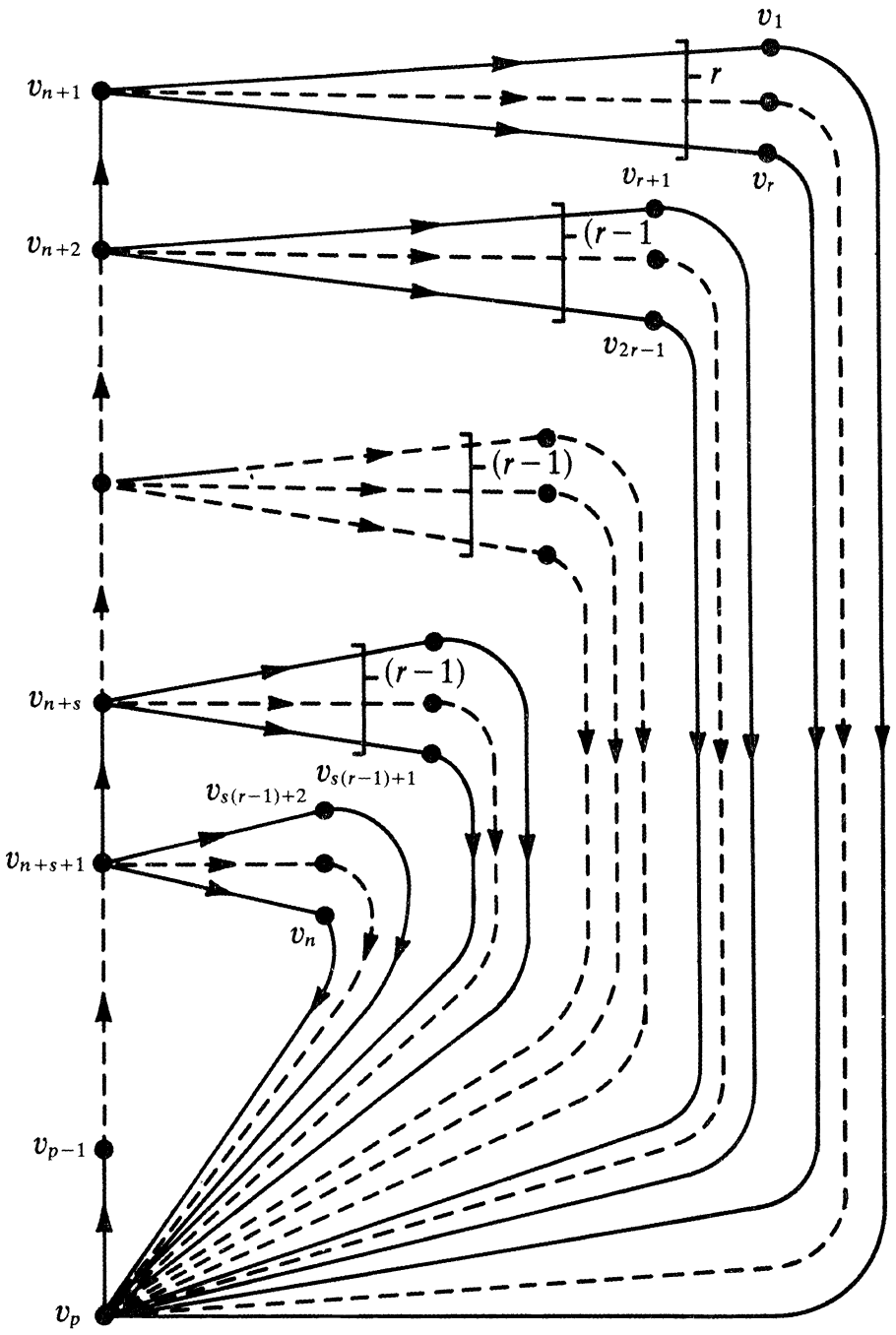
and we have $\mathrm{vol}(f)\geqq h(p,r)$. Therefore, $\mathrm{vol}(f)=h(p,r)$.

By Theorem 1, Theorem 7 and Fig. 6, we have the following theorem.

THEOREM 8. *The maximum tour length over all strong digraphs in $\mathscr{D}_p^r$ is $h(p,r)$.*

COROLLARY. $h(p,\infty)=[\frac{1}{4}(p+1)^2]$, *and* $h(p,2)=(p+1-\frac{3}{2}[p/3])[(p+3)/3]-1$.

Appendix A. Proof of Theorem 6. Let $p=s(2(r-1)+1)+t$, $0 \leqq t \leqq 2(r-1)$ and let $n_0=s(r-1)+1+\{(t-2)/2\}$. If $r=1$, only $n=1$ satisfies $1 \leqq n+Q(n,r) \leqq p$ and $n_0=1$, where $Q(n,r)=[(n-1)/(r-1)]$. Hence the theorem holds and $h(p,1)=p$. If $r=2$ and $p=2$, only $n=1$ satisfies $1 \leqq n+$



$$p = s(2(r-1)+1)+t, \quad 0 \leqq t \leqq 2(r-1) \quad \textit{and} \quad n = s(r-1)+1+\left\{\frac{t-2}{2}\right\}$$

FIG. 6. *A strong digraph whose tour length is* $h(p, r)$

$Q(n, r) \leqq 2$ and $n_0 = 1$. If $r \geqq 3$ and $p = 2$, only $n = 1$ or 2 satisfies $1 \leqq n + Q(n, r) \leqq 2$ and $n_0 = 1$. But $g(1, 2, r) = g(2, 2, r) = 2$. Hence for $r \geqq 2$ and $p = 2$, the theorem holds.

Consider the case where $r \geqq 2$ and $p \geqq 3$. Let $f(n) = g(n + 1, p, r) - g(n, p, r)$. Then $f(n) = p - 2n - k$ for all n, $k(r - 1) \leqq n - 1 \leqq (k + 1)(r - 1) - 2$ or $n - 1 = (k + 1)(r - 1) - 1$. Hence $f(n) = p - 2n - Q(n, r)$. Since $f(n_0) = t - 2 - 2\{(t - 2)/2\}$,

$$f(n_0) = \begin{cases} 0 & \text{if } t \geqq 2 \text{ and } t \text{ is even}, \\ -1 & \text{if } t > 2 \text{ and } t \text{ is odd}, \quad \text{or } t = 1, \\ -2 & \text{if } t = 0. \end{cases}$$

Since $f(n_0 - 1) = f(n_0) + 2 + Q(n_0, r) - Q(n_0 - 1, r)$,

$$f(n_0 - 1) = \begin{cases} 2 & \text{if } t \geqq 4 \text{ and } t \text{ is even}, \quad \text{or } t = 1, \\ 1 & \text{if } t \geqq 3 \text{ and } t \text{ is odd}, \quad \text{or } t = 0, \\ 3 & \text{if } t = 2. \end{cases}$$

Since $1 \leqq n_0 + Q(n_0, r) \leqq p$ and $f(n)$ is monotonously decreasing, we have

$$g(1, p, r) < \cdots < g(n_0 - 1, p, r) < g(n_0, p, r) \geqq g(n_0 + 1, p, r) > \cdots.$$

Therefore $g(n, p, r)$ is a maximum at $n = n_0$ and clearly $h(p, r) = g(n_0, p, r)$.

Appendix B. Proof of Lemma 4.

(i). Table 1 shows the values of t, s and n in Theorem 6. t_q, s_q and n_q correspond to the case where $p = q$, while t_{q-1}, s_{q-1} and n_{q-1} to the case where $p = q - 1$.

TABLE 1

t		s		n	
t_q	t_{q-1}	s_q	s_{q-1}	n_q	n_{q-1}
0	$2r - 2$	s	$s - 1$	$s(r - 1) + 1$	$s(r - 1)$
1	0	s	s	$s(r - 1) + 1$	$s(r - 1) + 1$
2	1	s	s	$s(r - 1) + 1$	$s(r - 1) + 1$
3	2	s	s	$s(r - 1) + 2$	$s(r - 1) + 1$
4	3	s	s	$s(r - 1) + 2$	$s(r - 1) + 2$
5	4	s	s	$s(r - 1) + 3$	$s(r - 1) + 2$
6	5	s	s	$s(r - 1) + 3$	$s(r - 1) + 3$
7	6	s	s	$s(r - 1) + 4$	$s(r - 1) + 3$
⋮	⋮	⋮	⋮	⋮	⋮
$2r - 3$	$2r - 4$	s	s	$s(r - 1) + r - 1$	$s(r - 1) + r - 2$
$2r - 2$	$2r - 3$	s	s	$s(r - 1) + r - 1$	$s(r - 1) + r - 1$

From Table 1 it suffices to consider the following three cases.

(a) If $n_q = n_{q-1}$, $s_q = s_{q-1}$ and hence $h(q,r) - h(q-1,r) = n_q$. If $s_q = 0$, $t_q = q \geqq 4$ and hence $n_q \geqq s_q(r-1) + 2 \geqq 2$. If $s_q \geqq 1$, $n_q \geqq s_q(r-1) + 1 \geqq 2$. In any case, $h(q,r) - h(q-1,r) \geqq 2$.

(b) If $n_q - 1 = n_{q-1}$ and $s_q - 1 = s_{q-1}$, $t_q = 0$, $s_q \geqq 1$ and $h(q,r) - h(q-1,r) = s_q(r-1) + 1 \geqq 2$.

(c) If $n_q - 1 = n_{q-1}$ and $s_q = s_{q-1}$, t_q is odd and $t_q \geqq 3$, and hence $h(q,r) - h(q-1,r) = s_q(r-1) + (t_q + 1)/2 \geqq 2$.

In Theorem 6, if $r = 2$, $p = 3s + t$, $0 \leqq t \leqq 2$ and $n = s + 1$.

(ii). $q \geqq 6$ implies $s_q \geqq 2$. Then, from Table 1, $h(q,2) - h(q-1,2) = n_q = s_q + 1 \geqq 3$.

(iii). $q \geqq 9$ implies $s_q \geqq 3$. Then $h(q,2) - h(q-1,2) = n_q = s_q + 1 \geqq 4$.

(iv) and (v). We have Table 2. $q \geqq 6$ implies $s_q \geqq 2$. $q \geqq 7$ implies either $s_q \geqq 3$, or $s_q = 2$ and $t_q \geqq 1$.

TABLE 2

t		s		n	
t_q	t_{q-2}	s_q	s_{q-2}	n_q	n_{q-2}
0	1	s	$s-1$	$s+1$	s
1	2	s	$s-1$	$s+1$	s
2	0	s	s	$s+1$	$s+1$

If $s_q = s_{q-2}$, $n_q = n_{q-2} = s_q + 1$ and $h(q,2) - h(q-2,2) = 2n_q = 2(s_q + 1) \geqq 6$. If $s_q - 1 = s_{q-2}$, $n_q = s_q + 1$ and $n_{q-2} = s_q$ and $h(q,2) - h(q-2,2) = q + 1 - s_q = 2s_q + t_q + 1 \geqq 5$ for $q \geqq 6$ and $\geqq 6$ for $q \geqq 7$.

(vi). We have Table 3.

TABLE 3

		s		n	
t_q	t_{q-3}	s_q	s_{q-3}	n_q	n_{q-3}
0	0	s	$s-1$	$s+1$	s
1	1	s	$s-1$	$s+1$	s
2	2	s	$s-1$	$s+1$	s

Then we have $h(q,2) - h(q-3,2) = q + 1$.

Appendix C. Proof of Lemma 5. The proof proceeds by induction on p_1 or p_2. Since $h(9,2) = 21$, $h(5,2) = 8$ and $h(4,2) = 6$, $h(9,2) \geqq h(5,2) + h(4,2) + 5 + 4 - 2$. Since $h(6,3) = 12$ and $h(3,3) = 4$, $h(6,3) \geqq h(3,3) + h(3,3) + 3 + 3 - 2$. Let $\Delta h(p_1 + p_2) = h(p_1 + p_2 + 1, r) - h(p_1 + p_2, r)$ and $\Delta h(p_1) = h(p_1 + 1, r) - h(p_1, r)$. If $\Delta h(p_1 + p_2) - \Delta h(p_1) \geqq 1$ and $h(p_1 + p_2, r) \geqq h(p_1, r) + h(p_2, r) +$

$p_1 + p_2 - 2$, $h(p_1 + p_2 + 1, r) \geqq h(p_1 + 1, r) + h(p_2, r) + (p_1 + 1) + p_2 - 2$. Moreover, $h(p_1 + p_2, r) - [h(p_1, r) + h(p_2, r) + p_1 + p_2 - 2]$ is symmetric in terms of p_1 and p_2. Therefore, it suffices to prove that $\Delta h(p_1 + p_2) - \Delta h(p_1) \geqq 1$ for $p_1, p_2 \geqq 3$ and $r \geqq 2$.

Let

$$\Delta H = \Delta h(p_1 + p_2) - \Delta h(p_1),$$

$$p_i = s_i(2r - 1) + t_i, \quad n_i = s_i(r - 1) + 1 + \left\{\frac{t_i - 2}{2}\right\}, \qquad i = 1, 2,$$

$$p_1 + 1 = s_1'(2r - 1) + t_1', \quad n_1' = s_1'(r - 1) + 1 + \left\{\frac{t_1' - 2}{2}\right\},$$

$$p_1 + p_2 = s(2r - 1) + t, \quad n = s(r - 1) + 1 + \left\{\frac{t - 2}{2}\right\},$$

and

$$p_1 + p_2 + 1 = s'(2r - 1) + t', \quad n' = s'(r - 1) + 1 + \left\{\frac{t' - 2}{2}\right\},$$

$$0 \leqq t_1, t_2, t_1', t, t' \leqq 2r - 2.$$

All the cases which must be considered are in Table 4.

TABLE 4

				s	t	s'	t'	s_1'	t_1'	
I	$t_1 + t_2 < 2r - 2$			$s_1 + s_2$	$t_1 + t_2$	s	$t + 1$	s_1	$t_1 + 1$	
II	$t_1 + t_2 = 2r - 2$	1	$t_1 = 2r - 2$	$s_1 + s_2$	$t_1 + t_2$	$s + 1$	0	$s_1 + 1$	0	$t_2 = 0$
		2	$t_1 < 2r - 2$	$s_1 + s_2$	$t_1 + t_2$	$s + 1$	0	s_1	$t_1 + 1$	
III	$t_1 + t_2 > 2r - 2$	1	$t_1 = 2r - 2$	$s_1 + s_2 + 1$	$t_1 + t_2 - 2r + 1$ $(= t_2 - 1)$	s	$t + 1$ $(= t_2)$	$s_1 + 1$	0	
		2	$t_1 < 2r - 2$	$s_1 + s_2 + 1$	$t_1 + t_2 - 2r + 1$	s	$t + 1$	s_1	$t_1 + 1$	

Case I. The following cases must be considered; either (A) $n' = n$, or (B) $n' = n + 1$, and either (a) $n_1' = n_1$, or (b) $n_1' = n_1 + 1$. ΔH for each combination of these cases is in Table 5. If $s_2 \geqq 1$, $\Delta H \geqq 1$ for all cases. If $s_2 = 0$, $t_2 = p_2 \geqq 3$ and we can easily show $\Delta H \geqq 1$ for all cases.

Case (II-1). From Table 1, $s' = s + 1$ and $t' = 0$ imply $n' = n + 1 = s'(r - 1) + 1$. Similarly, $n_1' = n_1 + 1 = s_1'(r - 1) + 1$. Then $\Delta H = s_2(r - 1)$. If $s_2 = 0$, $t_2 = p_2 \geqq 3$, and this contradicts $t_2 = 0$. If $s_2 \geqq 1$, $\Delta H \geqq 1$.

Case (II-2). Either (a) $n_1' = n_1$ or (b) $n_1' = n_1 + 1$.

(a) $\Delta H = s_2(r - 1) + (r - 1) - \{(t_1' - 2)/2\}$, $t_1' \leqq 2r - 2$, therefore $\Delta H \geqq 1$.

(b) $\Delta H = s_2(r - 1) + (r - 1) + 1 - (t_1' + 1)/2$, $3 \leqq t_1' \leqq 2r - 3$ and t_1' is odd, therfore $\Delta H \geqq 1$.

TABLE 5

	(a)	(b)
(A)	$s_2(r-1)+\left\{\frac{t_1+t_2-1}{2}\right\}-\left\{\frac{t_1-1}{2}\right\}$	$s_2(r-1)+\left\{\frac{t_1+t_2-1}{2}\right\}-\frac{t_1}{2}$ $2r-4 \geqq t_1 \geqq 2$ and t_1 is even
(B)	$s_2(r-1)+\frac{t_1+t_2}{2}-\left\{\frac{t_1-1}{2}\right\}$	$s_2(r-1)+\frac{t_2}{2}$

Case (III-1). Either (a) $n' = n$, or (b) $n' = n + 1$. From Table 1, $t'_1 = 0$ implies $n'_1 = n_1 + 1 = s'_1(r-1) + 1$.

(a) $\Delta H = s_2(r-1) + \{(t'-2)/2\} = s_2(r-1) + \{(t_2-2)/2\}$. $s_2 = 0$ implies $t_2 = p_2 \geqq 3$, therefore $\Delta H \geqq 1$.

(b) $\Delta H = s_2(r-1) + (t'+1)/2 - 1$, $3 \leqq t' \leqq 2r-3$ and t' is odd, therefore $\Delta H \geqq 1$.

Case (III-2). The following cases must be considered: either (A) $n' = n$, or (B) $n' = n+1$, and either (a) $n'_1 = n_1$, or (b) $n'_1 = n_1 + 1$. ΔH for each combination of these cases is in Table 6. We can easily show $\Delta H \geqq 1$ for all cases.

We have completed the proof of Lemma 5.

TABLE 6

	(a)	(b)
(A)	$(s_2+1)(r-1)+\left\{\frac{t'-2}{2}\right\}-\left\{\frac{t'_1-2}{2}\right\}$	$(s_2+1)(r-1)+1+\left\{\frac{t'-2}{2}\right\}-\frac{t'_1+1}{2}$ $3 \leqq t'_1 \leqq 2r-3$ and t'_1 is odd
(B)	$(s_2+1)(r-1)+\frac{t'+1}{2}-1-\left\{\frac{t'_1-2}{2}\right\}$ $3 \leqq t'$ and t' is odd	$(s_2+1)(r-1)+\frac{t'-t'_1}{2}$ $3 \leqq t', t'_1 \leqq 2r-3$ and t' and t'_1 are odd

REFERENCES

[1] T. ASANO, M. SHIBUI AND I. TAKANAMI, *The least upper bounds for tour lengths in Moore machines*, Papers of Technical Group on Computers, I.E.C.E., Japan, Feb. 1974.

[2] A. K. DEWDNEY AND A. L. SZILARD, *Tours in machines and digraphs*, IEEE Trans. Comput., C-22 (1973), pp. 635–638.

[3] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[4] F. C. HENNIE, *Finite-state models for logical machines*, John Wiley, New York, 1968.

[5] T. ASANO AND I. TAKANAMI, *The length of shortest closed walks passing all transition arcs in machines*, Trans. Institute of Electronics and Communications Engineers of Japan, 58-D (1975), pp. 17-22.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

ON SETS COOK-REDUCIBLE TO SPARSE SETS*

ROBERT M. SOLOVAY†

Abstract. Lynch showed that any set many-one reducible in polynomial time to arbitrarily sparse sets is polynomial time computable. We prove the analogous theorem for polynomial time Turing reproducibility.

Key words. polynomial time computable, Turing reducibility

Introduction. We settle a technical question left open in [2]. In that paper, Lynch showed that any set many-one reducible in polynomial time to arbitrarily sparse sets lay in $\mathscr{P}$, and conjectured the analogous result for reducibility in Cook's sense. We prove her conjecture by a combination of the methods in [2] and those used to prove Theorem 6 of [1].

This paper is organized as follows. In § 1, we recall the basic definitions. We follow [2] except that we identify the binary string s with the integer whose binary representation is $1s$. In § 2, we state our theorem precisely and reformulate Lynch's result on the core of a set in a more convenient form. In § 3, we outline our proof. The remainder of the paper is devoted to the details.

1. Notation and definitions. All sets will be sets of finite strings over $\Sigma = \{0, 1\}$. If $x \in \Sigma^*$, $|x|$ will represent the length of the string x. We shall identify the string x with the integer whose binary representation is $1x$. In this way, the set Σ^* is identified with the set of positive integers. Note that $|x|$ is simply the integral part of $\log x$. (All logarithms in this paper are to the base 2.)

For a set A, $|A|$ will represent the cardinality of A and $\mathscr{C}_A$ its characteristic function.

λ represents the empty string. (It will also be used as in Church's lambda notation.)

We write "i.o. (x)" to denote "for infinitely many x", and "a.e. (x)" to denote "for all except possibly finitely many x". When no confusion is likely, we write simply i.o. or a.e.

If A is a set and $t : \Sigma^* \to N$ is a recursive function, we write Comp $A \leqq t$ if there is a deterministic Turing machine computing $\mathscr{C}_A$, which uses not more than $t(x)$ steps on the string x. (The machine runs in "time $t(x)$".) Our machines are not restricted as to their number of tapes or worktape symbols.

We write $A \in \mathscr{P}$ if for some polynomial p, Comp $A \leqq \lambda x[p(|x|)]$.

We write $A \leqq_T^p B$ (A is polynomial time Turing reducible to B) for Cook's reducibility. Namely, $A \leqq_T^p B$ iff there is an oracle Turing machine M and a polynomial p such that:

$$x \in A \Leftrightarrow M \text{ with input } x \text{ and oracle } B \text{ accepts within } p(|x|) \text{ steps.}$$

* Received by the editors February 13, 1975, and in revised form September 29, 1975.

† Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

We write $A \leqq_m^p B$ (A is polynomial time many-one reducible to B) for Karp's reducibility. Namely, $A \leqq_m^p B$ iff there is a polynomial time computable function f such that:

$$x \in A \Leftrightarrow f(x) \in B.$$

2. Sparse sets and cores. We now recall Lynch's notion of a set B being s-sparse. In Lynch's notion, what is really "sparse" about B is the set of places where B is difficult to compute. Lemma 2.1 will reduce the study of such B's to that of B's which are sparse in the usual sense of the word.

Let $s : N \to N$ be a recursive function. Let B be recursive. We say B is *s-sparse* if there is a Turing machine M computing $\mathscr{C}_B$ and a polynomial p such that for any integer x, M runs in time greater than $\lambda y[p(|y|)]$ for at most x strings of length $\leqq s(x)$.

THEOREM. *The following conditions on an $A \subseteq \Sigma^*$ are equivalent*:

(i) $A \in \mathscr{P}$;

(ii) ($\forall$ *recursive* $s : N \to N$) ($\exists$ *recursive* $B \subseteq \Sigma^*$) [*B is s-sparse and* $A \leqq_T^p B$].

Of course, the implication "(i) $\Rightarrow$ (ii)" is trivial.

LEMMA 2.1. *Let A satisfy* (ii) *of the theorem. Let $s : N \to N$ be recursive. Then there are recursive sets B and C with the following properties*:

(i) $A \leqq_T^p B$;

(ii) $B \subseteq C$;

(iii) *if C is infinite and if $\{c_n : n \in N\}$ is an enumeration of C in increasing order, then for all n, $s(n) \leqq c_n$* (i.e., *C is, informally speaking, sparse*);

(iv) $C \in \mathscr{P}$.

Proof. Let B_1 be provided by clause (ii) of the theorem applied to s. Let M be the machine guaranteed by the fact that B_1 is s-sparse, and p the corresponding polynomial. Let

$$C = \{x \mid M \text{ takes} > p(|x|) \text{ steps to compute } \mathscr{C}_{B_1}(x)\}.$$

Let $B = B_1 \cap C$. Then properties (ii) and (iv) are clear. Note that $B_1 \leqq_T^p B$. (Use a B-oracle to decide "$x \in B_1$?" if $x \in C$; use M, if $x \notin C$. This division into cases can be done in polynomial time, since $C \in \mathscr{P}$.) Since $A \leqq_T^p B_1$, (i) is clear. Since B_1 is s-sparse, C has at most n members of length $\leqq s(n)$. Deleting the least member of C from C and B will not upset (i), (ii), or (iv), and we will now have $c_n \geqq s(n)$ as stated by (iii) since now

$$|C \cap [0, s(n))| \leqq n - 1.$$

The following lemma is proved by Lynch in [2, Lem. 1, p. 342].

LEMMA 2.2. *Let A be a set with $A \notin \mathscr{P}$. Then there is an infinite recursive set X with the following property. Let M be a machine that computes $\mathscr{C}_A$ (with time function Φ). Let p be a polynomial. Then for a.e. x in X,*

$$\Phi(x) > p(|x|).$$

(Following Lynch, we call X a *core* for A.)

LEMMA 2.3. *Let A be a set of strings not in $\mathscr{P}$. Let X be a core for A. Let M be a Turing machine that computes the partial function φ. We suppose that $\varphi(x)$, whenever defined, is equal to $\mathscr{C}_A(x)$. Let $\Phi(x)$ be the time M takes on input x. (Thus*

if $\varphi(x)$ is undefined, $\Phi(x) = \infty$.) Then if p is any polynomial, we have

$$\Phi(x) \geqq p(|x|) \quad \textit{a.e. on } X.$$

Proof. If not, let M_1 be a machine that computes $\mathscr{C}_A$. We run M_1 and M in parallel, getting a machine M_2 (with time function Φ_2, say) such that M_2 computes $\mathscr{C}_A$ and $\Phi_2(x) \leqq 2\Phi(x)$. So by our assumption on M, M_2 computes x in time $\leqq 2p(|x|)$ for infinitely many x in X. But this contradicts X being a core for A.

3. Outline of the proof. At the suggestion of the referee, we indicate after each step of our outline the portion of § 4 to which it corresponds.

Let A satisfy clause (ii) of the theorem. We assume $A \notin \mathscr{P}$ and will derive a contradiction. Let X be a suitably sparse core for A. We are going to choose a rapidly growing function s, and let B, C be given by Lemma 2.1.

If C were finite, the conclusion $A \in \mathscr{P}$ would follow from (i) and (ii) of Lemma 2.1. Thus C is infinite. Our choice of s will guarantee the following happens for infinitely many n:

(a) Let

$$e(x) = 2^{2^{2x}}.$$

Let $l(x)$ be the largest z such that $e(z) \leqq x$. (If $x \leqq 3$, $l(x) = 0$.)

Then $X \cap [e(c_n), l(c_{n+1}))$ $(= W_n$, say) will have at least $e(c_n)$ members. (Cf. §§ 4.1 and 4.2.)

(b) No computation of "$x \in A$?" from B by the program provided by Lemma 2.1 will ever consult a member of $C > c_n$, if $x \in W_n$. This is because $e(x) < c_{n+1}$. (Cf. § 4.3.)

(c) There are at most 2^{n+1} possible candidates for $B \cap [0, c_n]$ (since $B \subseteq C$). Let us call two members of W_n equivalent if for all possible B's, the oracle makes the same predictions for x, y. A pigeonhole argument now shows that there are x, y in W_n unequal but equivalent. Say $y < x$. (Cf. § 4.6.)

(d) The following procedure will contradict Lemma 2.3. We look, given x for an equivalent $y < x$. Having found y, check using some recursive algorithm for A whether $y \in A$. Then $x \in A$ iff $y \in A$. (Cf. §§ 4.4 and 4.5.)

Of course, this outline leaves many loose ends dangling. We turn to the detailed proof.

4. The proof.

4.1. Fix M_0 that computes $\mathscr{C}_A$. Let $g_0(n)$ be the longest time M_0 takes on any input of length at most n. Let $g(n) = \max(g_0(n), e(n))$. Note that g is monotone increasing in n. We assume $A \notin \mathscr{P}$ and that clause (ii) of the theorem holds. We shall derive a contradiction. We select a core, X, for A. By shrinking X to some infinite subset if need be, we arrange that

$$g(x_n) < \log(x_{n+1}). \tag{1}$$

(Here $\langle x_n : n \in N \rangle$ is an enumeration of X in increasing order.)

Before stating the following lemma, we need to develop some notation. Let C be an infinite subset of N. Let $\langle c_n : n \in N \rangle$ be an enumeration of C in increasing

order. Let

$$W_n = \{x \in X : g(c_n) \leqq x < g(x) \leqq c_{n+1}\}.$$

(This definition is slightly different from the provisional definition given in § 3.) It will be important for the proof of Lemma 4.1 that W_n is known once the members c_n and c_{n+1} are known.

LEMMA 4.1. *There is a recursive function $s : N \to N$ such that, if C is any infinite set such that for all n, $s(n) \leqq c_n$, then for infinitely many n, we have*

$$(\mathscr{A}_n) \qquad |W_n| \geqq g(c_n).$$

Proof. Let $h_0(n)$ be the least integer m such that $\{x \in X : g(n) \leqq x < g(x) \leqq m\}$ has at least $g(n)$ elements. Such an m clearly exists since X is infinite. Moreover, it is immediate from the definitions of W_n and of $(\mathscr{A}_n)$ that if $c_{n+1} \geqq h_0(c_n)$, then $(\mathscr{A}_n)$ holds.

Let $h_1 : N \to N$ be a strictly increasing recursive function such that $h_1(n) \geqq h_0(n)$, for all n. From the last sentence of the preceding paragraph, we see that if $(\mathscr{A}_n)$ does not hold, then $c_{n+1} < h_1(c_n)$.

Let $h_2 : N \to N$ be defined by: $h_2(0) = 0$, $h_2(n+1) = h_1(h_2(n))$. Thus if $(\mathscr{A}_n)$ fails and $c_n \leqq h_2(n+k)$, then $c_{n+1} \leqq h_2(n+k+1)$.

Suppose that a.e. (n), $(\mathscr{A}_n)$ fails. We shall show then that for some k, we have

$$c_n \leqq h_2(n+k), \quad \text{all } n.$$

Indeed, suppose that $(\mathscr{A}_n)$ fails for $n \geqq n_0$. Pick k so large that $c_n \leqq h_2(n+k)$ for all $n \leqq n_0$. Since $(\mathscr{A}_n)$ fails for $n \geqq n_0$, we can now prove that $c_n \leqq h_2(n+k)$ for $n > n_0$ by induction on n, using the remark of the preceding paragraph.

Now let $s(n) = h_2(2n)$. We show that this choice of s makes the Lemma 4.1 true. Suppose, towards a contradiction, that we can choose the infinite set C so that $c_n \geqq s(n)$ for all n, but a.e. (n), $(\mathscr{A}_n)$, fails. But then by the preceding paragraph, we have, for a suitable k, and all $n \geqq k+1$,

$$c_n \leqq h_2(n+k) < h_2(2n) = s(n) \leqq c_n.$$

This gives us $c_n < c_n$, which is the desired contradiction.

4.2. We now invoke Lemma 2.1 for the s just constructed, getting B and C, with C infinite. We let M be the oracle machine that computes A from B in time $\lambda x[p(|x|)]$ for some polynomial p.

LEMMA 4.2. *There is an oracle machine M^* that computes A from B in polynomial time and has the following additional properties.*

(i) *There is a fixed polynomial p^* such that for any oracle $Y \subseteq \Sigma^*$ and any input $x \in \Sigma^*$, Y halts in time $p^*(|x|)$, with output in $\{0, 1\}$.*

(ii) *For any Y, x, as above, M^* will never ask the oracle: "Is $z \in Y$?" for any z not in C.*

Proof. M^* will simulate the operation of M. Whenever M presents its oracle with a query about z, M^* determines by a polynomial time algorithm if $z \in C$. If not, M^* tells its version of M the answer no. If so, M^* consults its own oracle and transmits the answer to its M.

At the same time, M^* is keeping a count of the number of steps M has taken. If M attempts to take more than $p(|x|)$ steps, M^* halts and gives output zero.

Similarly, if M gives an output other than 0 or 1, M^* gives output 0. Otherwise, M^* gives the same output as its version of M.

Since $C \in \mathscr{P}$, M^* runs in polynomial time; i.e., (i) holds. Since $B \subseteq C$, with oracle B, M^* correctly simulates M and so computes $\mathscr{C}_A$. By construction, M^* has property (ii) of the lemma.

To simplify notation, we drop the *'s and assume our original M and p have the properties (i) and (ii) of Lemma 4.2.

4.3. By our choice of s, $\mathscr{A}_n$ holds i.o. (n). In the remainder of the proof, n is an integer for which $(\mathscr{A}_n)$ holds, and when we say "a.e. (n)" we mean "for all but finitely many n for which $(\mathscr{A}_n)$ holds".

Let W_n be as in § 4.1. Let $x \in W_n$. Then a.e. (n), no computation of M on any input x can consult any member of $C > c_n$. This is because a.e. (n), we have

$$p(|x|) \leqq 2^{|x|} < |c_{n+1}|$$

(since $e(|x|) < g(x) \leqq c_{n+1}$).

It follows that for $x \in W_n$, $\{c_0, \cdots, c_n\}$ is a kernel for x in the sense of the following definition:

A finite subset C_1 of C is a *kernel* for x if for no oracle Y does M on input x consult Y about any z not in C_1.

LEMMA 4.3. *There is a Turing machine M_1 which on input the pair of numbers $\langle y, x\rangle$ with $y < x$, will do the following*:

(i) *print out a list of the members of $\{n \in C : n \leqq \log\log x\}$ (call this set E)*;

(ii) *determine whether or not E is a kernel for y; moreover, M_1 will run in time $\leqq p_1(|x|)$ for some polynomial p_1.*

Proof. M_1 proceeds as follows.

1. It computes $\log\log x$. (This can be done in time $O(|x|)$.)

2. It computes E and prints a list of E on its output tape (cf. (i) of the current lemma). (There are $O(\log\log x)$ numbers to check and each check takes $O(\log\log\log x)^k)$ steps for some k. Thus the whole procedure is easily $O(|x|)$.)

3. It lists all subsets of E, say $E_1, \cdots, E_r$. (This is easily $O(|x|)^k$.) Note $r = O(\log x) = O(|x|)$.

4. For each E_i, we determine if a computation on input y using E_i as an oracle will ever inquire of a $z \notin E$. If so, E is not a kernel for y.

Suppose, on the other hand, that for each i, the computation on input y with the oracle E_i does not inquire about z's not in E. Then E is a kernel for y. Indeed, if $Y \subseteq \Sigma^*$ and $E_j = Y \cap E$, then our assumption at the beginning of this paragraph guarantees that the computations on input y using the oracles Y and E_j will coincide, whence the Y oracle will never be questioned about a z not in E.

This last phase has to examine $O(|x|)$ subsets. Each examination takes time $O(|x|^k)$ for some suitably large k (independent of the E_i being examined!). Thus the whole procedure runs in time polynomial in $|x|$. This proves the lemma.

4.4. We now describe a certain Turing machine M_2. M_2 will have the following two properties.

(i) M_2 computes a partial function, φ, mapping into $\{0, 1\}$. If $\varphi(x)$ is defined, then $\varphi(x) = \mathscr{C}_A(x)$.

(ii) There is a polynomial q such that if $\mathscr{A}_n$ holds (and n is sufficiently large),

then for some $x \in W_n$, we will have $\varphi(x)$ defined, and M_2 computes $\varphi(x)$ in $\leqq q(|x|)$ steps.

Of course, once we construct M_2 satisfying (i) and (ii), we will have contradicted Lemma 2.3; i.e., our assumption that $A \notin \mathscr{P}$ will have led to a contradiction and our proof of our main theorem will be complete.

We proceed to our description of M_2. Let the input x be given. M_2 proceeds as follows.

1. Compute $C \cap [1, \log\log x]$ (call it E), and check that E is a kernel for x. (cf. Lemma 4.3.) (If not, halt.)

2. Now search for the least $y \leqq \log\log x$ satisfying the following conditions (and if there is no such y, then halt):

(a) $y \geqq \max(E)$.

(b) E is a kernel for y.

(c) For each subset S of E, $\varphi_M^S(y) = \varphi_M^S(x)$.

(Here M is our oracle machine such that $\varphi_M^B = \mathscr{C}_A$.)

3. We now use our given procedure, M_0, for computing $\mathscr{C}_A(y)$. We output $\mathscr{C}_A(y)$ as the value M_2 computes on input x.

This completes our description of M_2

4.5. Let φ be the partial function computed by M_2. Since there are "error halts" in the definition of M_2, φ need not be total.

LEMMA 4.5. *If $\varphi(x)$ is defined, $\varphi(x) = \mathscr{C}_A(x)$.*

Proof. Since $\varphi(x)$ is defined, E is a kernel for x. Let y be the number computed in step 2 of the computation of $\varphi(x)$. Then E is also a kernel for y. Thus $\mathscr{C}_A(x) = \varphi_i^{B\cap E}(x) = \varphi_i^{B\cap E}(x)$ (since E is a kernel for x), and similarly $\mathscr{C}_A(y) = \varphi_i^{B\cap E}(y)$. Clause (c) in our choice of y insures that $\varphi_i^{B\cap E}(x) = \varphi_i^{B\cap E}(y)$. The upshot is that $\mathscr{C}_A(x) = \mathscr{C}_A(y)$, and $\varphi(x) = \mathscr{C}_A(y) = \mathscr{C}_A(x)$. This proves the lemma.

4.6. We have verified claim (i) of 4.4, and turn to verification of claim (ii). Recall that our choice of s was designed to ensure that $\mathscr{A}_n$ held for infinitely many n (cf. Lemma 4.1). The analysis that follows will be valid for all but finitely many n's such that $\mathscr{A}_n$ holds. Thus at any point, our claims are permitted to hold with a finite number of exceptions.

Suppose $\mathscr{A}_n$ holds. Then there is a set $W_n \subseteq X$ such that (a) $|W_n| \geqq 2^{2^{2n}}$; (b) $x \in W_n$ implies

$$g(c_n) \leqq x < g(x) \leqq c_{n+1}.$$

We have remarked in § 4.3 that $g(x) \leqq c_{n+1}$ entails that no computation of $\varphi_M^S(x)$ can consult c_{n+1} (or any larger member of C). Thus $C \cap [1, \log\log(x)] = E = \{c_1, \cdots, c_n\}$ is a kernel for x if $x \in W_n$.

For each $x \in W_n$, define a map $s_x : P(E) \to \{0, 1\}$ by $s_x(D) = \varphi_M^D(x)$, $D \subseteq E$. (Here $P(E)$ is, of course, $\{D : D \subseteq E\}$.) Now there are 2^{2^n} maps of $P(E)$ into $\{0, 1\}$, and by (a) above, $|W_n| > 2^{2^n}$. Thus there are elements z and x of W_n such that $z < x$ and $s_z = s_x$ (pigeonhole principle!). We shall see that for this x, $\varphi(x)$ is defined.

In the first place, since $g(c_n) < x$, and $C \cap [1, \log\log x] = \{c_1, \cdots, c_n\}$ is a kernel for x, step 1 of the computation of φ is completed without an error halt. Next, since by our choice of X, $g(z) < \log(x)$ (cf. § 4.1), E is a kernel for z, and

$s_z = s_x$, we see that step 2 will not error halt. (Since z meets the requirements for y and $z \leqq \log\log x$.)

Our last claim is that for some polynomial p, independent of n or x, we can estimate the time M_2 takes on input x as $\leqq p(|x|)$.

Let us examine each step in the program for M_2 in turn. That step 1 can be done in time polynomial in $|x|$ is the content of Lemma 4.3.

For step 2, we have at most $\log(|x|)$ candidates for y. Thus it suffices to see we can check out each candidate in time polynomial in $|x|$. Clause (a) causes no trouble, and clause (b) is handled by Lemma 4.3. As for clause (c), we have to examine at most $2^{\log|x|}$ subsets of E since clearly E has at most $\log|x|$ members. Since $2^{\log|x|} = |x|$, and φ_M runs in polynomial time, there is no difficulty completing step 2 in time polynomial in x.

How about step 3? By our choice of X in § 4.1, $g_0(z) \leqq g(z) \leqq |x|$, since $z, x \in X$ and $z < x$. But since $|y| \leqq |z|$, M_0 computes $\mathscr{C}_A(y)$ in time at most $g_0(z) \leqq |x|$. Thus step 3 can be done in time polynomial in $|x|$.

This completes the verification of claim (ii) of § 4.4 for M_2. As explained in § 4.4, this establishes the theorem we set out to prove.

REFERENCES

[1] T. Baker, J. Gill and R. Solovay, *Relativizations of the $\mathscr{P} = ?\ \mathscr{NP}$ question*, this Journal, 4 (1975), pp. 431–442.

[2] N. Lynch, *On reducibility to complex or sparse sets*, J. Assoc. Comput. Mach., 22 (1975), pp. 341–345.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

AUGMENTATION PROBLEMS*

KAPALI P. ESWARAN† AND R. ENDRE TARJAN‡

Abstract. This paper considers problems in which the object is to add a minimum-weight set of edges to a graph so as to satisfy a given connectivity condition. Simple characterizations of the minimum number of edges necessary to make a directed graph strongly connected and to make an undirected graph bridge-connected or biconnected are given. Efficient algorithms for finding such minimum sets of edges are discussed. It is shown that the weighted versions of these problems are NP-complete.

Key words. algorithm, augmentation, biconnectivity, bridge-connectivity, connectivity, graph, strong connectivity, NP-complete problem

1. Introduction. A common computational problem in graph theory is that of determining how many vertices or edges must be removed from a graph in order to satisfy some connectivity property. For instance, we might ask how many vertices must be removed from a graph to disconnect it. If the answer is zero, the graph itself is disconnected; if the answer is one or more, the graph is connected; if the answer is two or more, the graph is biconnected, and so on. Many good algorithms have been developed for solving such problems [6], [12], [14].

We can turn this idea around and ask questions about how many edges must be added to a graph to satisfy a given connectivity property. Several well-known problems, and also several new ones, can be stated in this form. This paper proposes a theoretical framework for studying such augmentation problems, gives well-known examples of such problems, and analyzes in detail the strong connectivity, bridge-connectivity and biconnectivity augmentation problems.

1.1. Definitions. A (finite) graph $G = (\mathscr{V}, \mathscr{E})$ is a finite set of *vertices* $\mathscr{V}$ and a finite set of *edges* $\mathscr{E}$ such that either all elements of $\mathscr{E}$ are ordered pairs of distinct elements of $\mathscr{V}$ (G is *directed*) or all elements of $\mathscr{E}$ are unordered pairs of distinct elements of $\mathscr{V}$ (G is *undirected*). A directed edge (v, w) has *head* w and *tail* v, *enters* w, and *leaves* v. Generally, we let V denote the number of vertices and E the number of edges in G. If $\mathscr{E}$ contains all possible edges, then G is *complete*. A graph $G_1 = (\mathscr{V}_1, \mathscr{E}_1)$ is a *subgraph* of $G = (\mathscr{V}, \mathscr{E})$ if $\mathscr{V}_1 \subseteq \mathscr{V}$ and $\mathscr{E}_1 \subseteq \mathscr{E}$. G_1 is a *spanning* subgraph of G if $\mathscr{V}_1 = \mathscr{V}$.

A *path* p from v_1 to v_n is a sequence of edges $(v_1, v_2), (v_2, v_3), \cdots, (v_{n-1}, v_n)$. A path is *closed* if $v_1 = v_n$. A *cycle* is a closed path such that $v_1, v_2, \cdots, v_{n-1}$ are all distinct. A path may contain no vertices, but a cycle must contain at least two vertices. A cycle is *Hamiltonian* if it is spanning. A graph is *acyclic* if it has no cycles.

If G is an undirected graph, G is *connected* if there is a path between every pair of vertices. If x is a vertex of G such that there are vertices $v, w \neq x$ for which

* Received by the editors May 31, 1974, and in final revised form October 6, 1975. This work was supported in part by the Naval Electronic Systems Command under Contract N00039-71-C-0255, by the National Science Foundation under Grant GJ-35604X at the University of California, Berkeley and under Grant MCS75-21870 at Stanford University, and by a Miller Research Fellowship.

† IBM Research Laboratories, San Jose, California 95193.

‡ Computer Science Department, Stanford University, Stanford, California 94305.

every path from v to w contains x, and there is a path from v to w, then x is a *cutnode*. If G is connected and contains no cutnodes, it is *biconnected*. If (x, y) is an edge such that every path between x and y contains (x, y) then (x, y) is a *bridge*. (An edge is a bridge if and only if no cycles contain it.) If G is connected and contains no bridges, it is *bridge-connected*. If G is a directed graph, G is *strongly connected* if, for all pairs of vertices v and w, there is a path from v to w. The *connected* (*biconnected*, *bridge-connected*, *strongly connected*) *components* of a graph are its maximal connected (biconnected, bridge-connected, strongly connected) subgraphs.

A *tree* is an undirected, connected, acyclic graph. A *spanning tree* of a graph G is a spanning subgraph which is a tree. An *arborescence* is an acyclic directed graph with one vertex, the *root*, having no entering edge, and all other vertices having exactly one entering edge. A *spanning arborescence* of a directed graph G is a spanning subgraph which is an arborescence.

Let C be a property defined either for directed or undirected graphs. We shall assume that C is *monotone increasing*; i.e., if G_1 is a spanning subgraph of G_2 and C holds on G_1, then C holds on G_2. Many interesting properties, such as "G is connected", "G is nonplanar", are monotone increasing, but their negations ("G is not connected", "G is planar") are monotone decreasing and are not allowed here.

Let $\mathcal{V}$ be any set of vertices and let $f(v, w)$ be a real-valued cost function defined on all possible edges between vertices of $\mathcal{V}$. We shall allow f to have value infinity on some edges. If C is any monotone increasing graph property, then the *weighted augmentation problem* with respect to C is the problem of finding a spanning subgraph G of the complete graph on vertex set $\mathcal{V}$ such that G satisfies C and the total cost of G's edges is minimum. That is, we want to find a minimum-cost set of edges $\mathcal{E}$ between vertices $\mathcal{V}$ such that $G = (\mathcal{V}, \mathcal{E})$ satisfies C. If f is a 0-1 valued function, then the augmentation problem is said to be *unweighted*.

Because C is monotone increasing, we may assume without loss of generality that f is nonnegative and that $\mathcal{E}_0 \subseteq \mathcal{E}$, where $\mathcal{E}_0 = \{(v, w) | v, w \in \mathcal{V}, f(v, w) = 0\}$. Thus an equivalent way to look at an augmentation problem is to assume that a graph $G_0 = (\mathcal{V}, \mathcal{E}_0)$ and a cost $f(v, w)$ on edges not in $\mathcal{E}_0$ are given, and we desire a minimum-cost set of edges $\mathcal{E}'$ such that $G = (V, \mathcal{E}_0 \cup \mathcal{E}')$ satisfies C. If the augmentation problem is unweighted, we are interested in finding the minimum *number* of edges which must be added to a given graph G_0 so that it satisfies C.

Let $G^* = (\mathcal{V}, \mathcal{E}^*)$, where $\mathcal{E}^* = \{(v, w) | f(v, w) \text{ is finite}\}$. If C holds on G^*, then we may restrict our attention to G^* when trying to solve the weighted augmentation problem on vertex set $\mathcal{V}$ with cost function f. If C does not hold on G^*, then any graph with vertex set $\mathcal{V}$ which satisfies C has infinite cost. Thus, yet another way to look at an augmentation problem is to assume that a graph $G^* = (\mathcal{V}, \mathcal{E}^*)$ and a finite cost function $f(v, w)$ on edges in $\mathcal{E}^*$ are given, and we desire a minimum-cost spanning subgraph of G^* which satisfies C. For any unweighted augmentation problem, G^* is the complete graph on vertex set $\mathcal{V}$. We shall switch freely between these three ways of presenting an augmentation problem.

One class of hard-to-solve problems deserves special attention. This is the class of NP-*complete problems*, as studied by Cook [1], Karp [8], and others. Let Σ^* be the set of all finite strings of 0's and 1's. A subset of Σ^* is a *language*.

Let $\mathcal{P}(\mathcal{NP})$ be the class of languages recognizable in polynomial time on one-tape deterministic (nondeterministic) Turing machines. Let Π be the class of functions from Σ^* into Σ^* computable in polynomial time by deterministic one-tape Turing machines. If L and M are languages, we say L *is reducible to* M if there is a function $f \in \Pi$ such that $f(x) \in M$ iff $x \in L$. A language L is NP-*complete* if (i) $L \in \mathcal{NP}$ and (ii) every language in $\mathcal{NP}$ is reducible to L.

Reducibility is transitive, so in order to show (ii) for a given language L, we need only show that some known NP-complete problem is reducible to L. The NP-*complete* problems are computationally related in the sense that their time bounds are polynomial functions of one another; that is, either all these problems have polynomial-time algorithms, or none do. Such famous problems as the tautology problem, the traveling salesman problem and the chromatic number problem are all NP-complete [8].

1.2. Examples of augmentation problems. Several well-known problems are augmentation problems. For instance, let G^* be an undirected graph with a finite cost on each edge. If C is "a given vertex x is connected to a given vertex y", the resulting augmentation problem asks for the minimum-cost path in G^* between x and y. Efficient algorithms for solving this problem are discussed in [7]. If C is "G is connected", the augmentation problem asks for the minimum-cost spanning tree in G^*. Various researchers have developed algorithms to find minimum-cost spanning trees [9], [16], [17].

An analogous problem for directed graphs G^* uses as C: "all vertices in G are reachable from some single vertex". This augmentation problem asks for a minimum-cost spanning arborescence in the graph G^*. Edmonds [2] has proposed an algorithm for this problem; the algorithm is quite efficient if implemented properly [15].

Eswaran [3] has considered the problem of adding a minimum-cost set of edges to a directed graph G_0 so that there is a cycle which contains all the edges of G_0. He gives efficient algorithms for solving both the weighted version and the unweighted version of this problem. Goodman and Hedetniemi [5] have given an $O(V^2)$ algorithm to find a minimum number of edges which must be added to a tree so that the resulting graph has a Hamiltonian cycle. More recently, they have improved the algorithm to $O(V)$.

Not all augmentation problems have efficient algorithms. Let G^* be an undirected graph with a finite cost on each edge, let $S \subseteq \mathcal{V}$, and suppose C is "there is a path in G between any two vertices in S". This augmentation problem is called the *Steiner tree problem.* It asks for the minimum-cost tree in G^* containing all vertices in S; other vertices may or may not be included in the tree. Karp [8] has shown that the Steiner tree problem is NP-complete.

A much more general kind of augmentation problem has been studied by Frank and Chou [4]. Given a $V \times V$ symmetric matrix $[r_{ij}]$ with $r_{ii} = 0$ for all i, they ask for an undirected graph with fewest edges on vertex set $\mathcal{V} = \{1, 2, \cdots, V\}$ such that there are at least r_{ij} edge-disjoint paths between vertices i and j. They provide a polynomial-time algorithm for solving this unweighted augmentation problem, but they give no exact time bound. The weighted version of this problem is NP-complete (as we shall see). They do not discuss what happens in the

unweighted problem when some of the edges have cost zero instead of one; presumably the problem becomes much harder. Here we shall consider the special case when $r_{ij} = 2$ for all $i \neq j$ and some of the edges have cost zero.

2. The strong connectivity augmentation problem. Let $G_0 = (\mathscr{V}, \mathscr{E}_0)$ be a directed graph and let $f(u, v)$ be a function defined on edges not in $\mathscr{E}_0$ between vertices of $\mathscr{V}$. The *(weighted) strong connectivity augmentation problem* is the problem of finding a minimum-cost set of edges $\mathscr{E}$ such that $G = (\mathscr{V}, \mathscr{E}_0 \cup \mathscr{E})$ is strongly connected. If $f(u, v) = 1$ for all edges, the augmentation problem is *unweighted.* The weighted strong connectivity augmentation problem is NP-complete, but the unweighted version has an $O(V + E)$ algorithm.

THEOREM 1. *Let $\mathscr{V}$ be a set of vertices, f a cost function on ordered pairs of distinct vertices, and F a total cost. The problem of determining whether there exists a set, with cost F or less, of edges which strongly connect the vertices of $\mathscr{V}$ is* NP-*complete.*

Proof. (i) It is easy to construct a nondeterministic Turing machine which in polynomial time will guess a set of edges with total cost F or less and check whether the edges strongly connect the vertices of $\mathscr{V}$. Thus the strong connectivity augmentation problem is solvable on a nondeterministic polynomial-time-bounded Turing machine.

(ii) We prove that the directed Hamiltonian cycle problem (which is NP-complete [8]) is reducible to the strong connectivity augmentation problem. Let $G = (\mathscr{V}, \mathscr{E})$ be a directed graph. Construct a strong connectivity augmentation problem on vertex set $\mathscr{V}$ with costs $f(v, w) = 1$ if $(v, w) \in \mathscr{E}$, $f(v, w) = 2$ otherwise, and $F = V$. This augmentation problem has a solution with cost F or less if and only if G contains a Hamiltonian cycle. This construction is obviously computable in polynomial time, so the weighted strong connectivity augmentation problem is NP-complete. □

It should be noted that only the most general version of the strong connectivity augmentation problem is NP-complete. For instance, if G_0, the graph we wish to strongly connect, has the property that some vertex x can be reached from every vertex, then the strong connectivity augmentation problem reduces to that of finding a minimum weight spanning arborescence with root x, which is efficiently solvable using Edmonds' algorithm [2], [15].

Consider the unweighted version of the strong connectivity augmentation problem. We are given a graph G_0 to which we want to add a minimum number of edges to form a graph G which is strongly connected. We can reduce this problem to a simpler one by converting G_0 into a directed graph G_0' which contains one vertex for each strongly connected component (subgraph) of G_0. Two vertices representing components are joined by an edge in G_0' if there is an edge in G_0 from a vertex in one component to a vertex in the other. G_0' is called the *condensation* of G_0 (with respect to strong components). G_0' is well-defined, acyclic, and may be constructed in $O(V + E)$ time by using depth-first search [12].

Let $G_0 = (\mathscr{V}, \mathscr{E}_0)$ be a directed graph and let $G_0' = (\mathscr{V}', \mathscr{E}_0')$ be its condensation. For $v \in \mathscr{V}$, let $\alpha(v)$ be the vertex in G_0' corresponding to the strong component in G_0 containing v. For $x \in \mathscr{V}'$, let $\beta(x)$ be any vertex in the strong component of G_0 corresponding to x. Clearly $\alpha(\beta(x)) = x$ for all $x \in \mathscr{V}'$. The following result is obvious.

LEMMA 1. *Let A be an augmenting set of edges which strongly connects G_0. Then $\alpha(A) = \{(\alpha(v), \alpha(w)) | (v, w) \in A, \alpha(v) \neq \alpha(w)\}$ is a set of edges which strongly connects G_0'. Conversely, let B be an augmenting set of edges which strongly connects G_0'. Then $\beta(B) = \{(\beta(x), \beta(y)) | (x, y) \in B\}$ is a set of edges which strongly connects G_0.*

Because of Lemma 1, we can restrict our attention to the acyclic graph G_0'. This graph contains zero or more *sources* (vertices with no entering edges and one or more exiting edges), zero or more *sinks* (vertices with no exiting edges and one or more entering edges), and zero or more isolated vertices (no entering or exiting edges). The following theorem gives a lower bound on the numbering of augmenting edges needed to make G_0' strongly connected.

THEOREM 2. *Let G_0' be an acyclic directed graph with s sources, t sinks, and q isolated vertices, where $s + t + q > 1$. Then at least $\max(s, t) + q$ edges are needed to make G_0' strongly connected.*

Proof. After G_0' is made strongly connected, each source and each isolated vertex must have at least one entering edge. Thus at least $s + q$ edges must be added. Similarly, each sink and each isolated vertex must have at least one exiting edge. Thus at least $t + q$ edges must be added. Combining these two results gives the theorem. □

The bound in Theorem 2 is attainable; the rest of this section gives an efficient algorithm for finding a set of $\max(s, t) + q$ edges which will strongly connect G_0'. We may assume without loss of generality that $s \leqq t$. (Otherwise, reverse the directions of all the edges in G_0', solve the augmentation problem on the resulting graph, and reverse the direction of all edges in the minimum augmenting set.)

Let $v(1), \cdots, v(p)$ be sources of G_0' and $w(1), \cdots, w(p)$ be sinks of G_0' such that

(i) there is a path from $v(i)$ to $w(i)$ for $1 \leqq i \leqq p$;
(ii) for each source v there is a path from v to some $w(i)$; and
(iii) for each sink w there is a path from some $v(i)$ to w.

Let $v(p+1), \cdots, v(s)$ be the remaining sources of G_0', let $w(p+1), \cdots, w(s)$ be $s - p$ of the remaining sinks, and let $x(1), \cdots, x(q + t - s)$ be the isolated vertices and the sinks not among the $w(i)$. Let $G_0'' = \{V', E_0' \cup A\}$, where

$$A = \{(w(i), v(i+1)) | 1 \leqq i < p\} \cup \{w(i), v(i)) | q + 1 \leqq i \leqq s\}$$
$$\cup \begin{cases} \{(w(p), v(1))\} & \text{if } q + t - s = 0 \\ \{(w(p), x(1)), (x(q + t - s), v(1))\} \cup \{(x(i), x(i+1)) | 1 \leqq i < q + t - s\} \\ & \text{if } q + t - s > 0. \end{cases}$$

Note that A contains $t + q$ edges.

LEMMA 2. *G_0'' is strongly connected.*

Proof. Clearly G_0'' has a cycle which contains vertices $v(i)$, $w(i)$ such that $1 \leqq i \leqq p$, and vertices $x(i)$, $1 \leqq i \leqq q + t - s$ if $q + t - s > 0$; thus all of these vertices are mutually reachable. Consider any source $v(i)$ with $p + 1 \leqq i \leqq s$. By property (ii) there is a path from $v(i)$ to some vertex on the cycle and hence to all vertices on the cycle. Also $v(i)$ is reachable via added edge $(w(i), v(i))$ from $w(i)$. By (iii), $w(i)$ is reachable from some vertex on the cycle. Hence $v(i)$ is reachable from any vertex on the cycle. Similarly, any sink $w(i)$ with $p + 1 \leqq i \leqq s$ is mutually reachable with any vertex on the cycle. □

Thus A is a minimal augmenting set of edges for G'_0. The following algorithm, expressed in ALGOL-like notation, finds a set of sources and sinks which satisfy properties (i)–(iii):

```
algorithm ST: begin
  procedure SEARCH(x);
    if x is unmarked then
      begin
        if x is a sink and (w = 0) then w := x;
        mark x;
        for each y such that (x, y) is an edge do SEARCH(y);
      end SEARCH
  initialize all vertices to be unmarked;
  i := 0;
  while some sink is unmarked do begin
    choose some unmarked source v;
    w := 0;
```

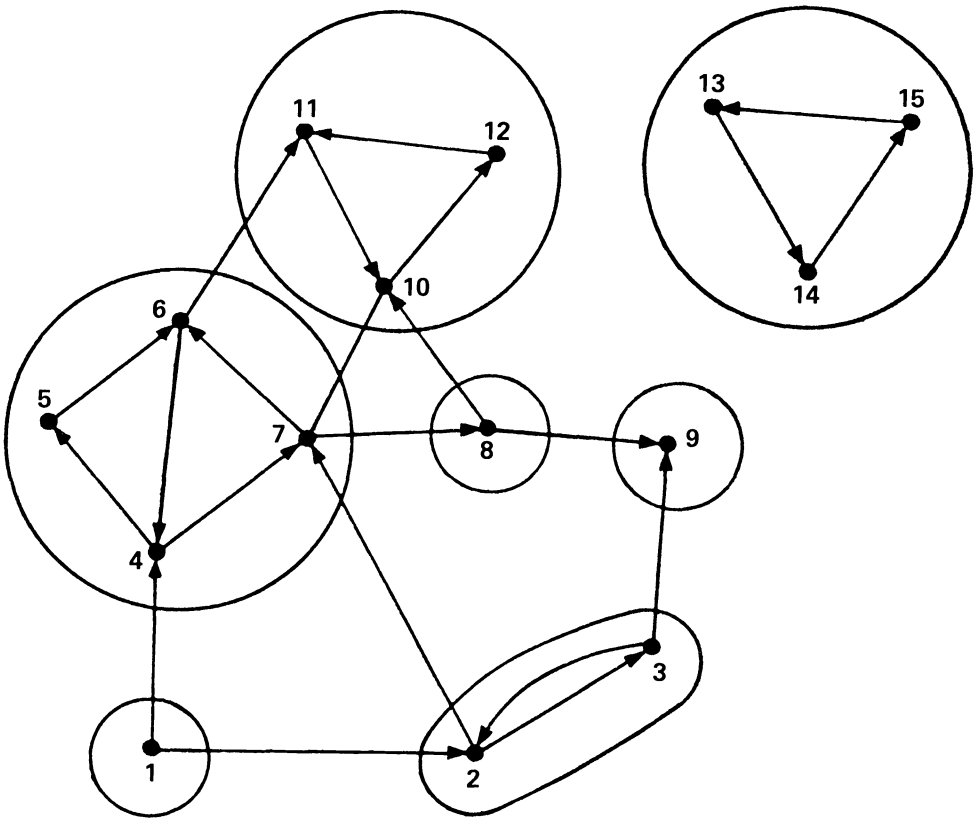


FIG. 1. *A directed graph G_0 which we wish to make strongly connected. Strongly connected components are circled.*

```
    SEARCH(v);
    if w ≠ 0 then begin
      i := i + 1;
      v(i) := v;
      w(i) := w;
  end end;
  p := i;
end ST;
```

It is obvious that this algorithm finds a sequence of sources and sinks satisfying (i)–(iii). It follows that the following algorithm will find a minimum augmenting set of edges for an arbitrary directed graph G_0:

algorithm STRONGCONNECT: **begin**
SC1: Use depth-first search to form the condensation G_0' of G_0, identifying the sources, sinks, and isolated vertices of G_0';
SC2: Apply algorithm ST to G_0' to find a set of sources and sinks satisfying (i)–(iii);
SC3: Construct the corresponding augmenting set of edges A;
SC4: Convert A into an augmenting set of edges for G_0, using Lemma 1;
end STRONGCONNECT;

It is easy to implement this algorithm to run in $O(V + E)$ time if the method described in [12] is used for step SC1. Algorithm ST clearly requires time proportional to the number of vertices and edges in G_0', and steps SC3 and SC4 clearly require time proportional to the number of vertices in G_0'. Figures 1–2 show the application of this algorithm to a directed graph.

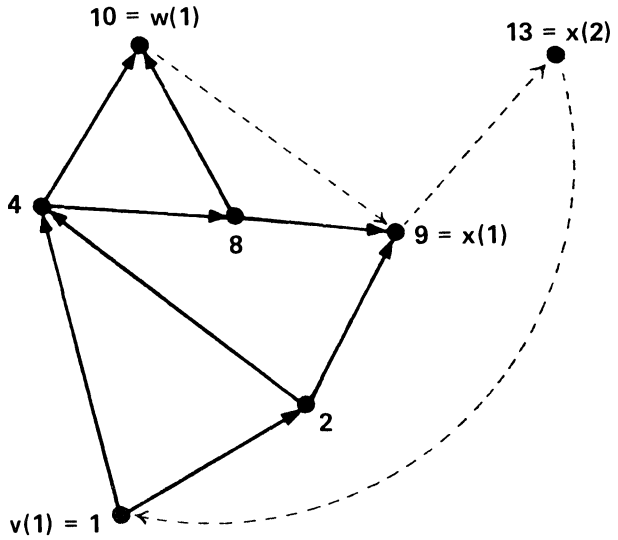


FIG. 2. *Acyclic directed graph corresponding to graph in Fig.* 1. *Numbers are those of arbitrary vertices in the corresponding strongly connected components. Dotted edges give a minimum strongly connecting augmentation, which also works for graph in Fig.* 1.

3. The bridge-connectivity and biconnectivity augmentation problems. Let G_0 be an undirected graph, and consider the problem of adding a minimum-cost set of edges to G_0 to make it (i) bridge-connected or (ii) biconnected. Both of these weighted augmentation problems are NP-complete, while the unweighted versions of these problems have $O(V + E)$ algorithms.

THEOREM 3. *Let $\mathscr{V}$ be a set of vertices, f a cost function on unordered pairs of distinct vertices, and F a total cost. The problem of determining whether there exists a set, with cost F or less, of edges which bridge-connect the vertices of $\mathscr{V}$ is* NP-*complete.*

Proof. (i) It is easy to construct a nondeterministic Turing machine which solves the bridge-connectivity augmentation problem in polynomial time.

(ii) We prove that the undirected Hamiltonian cycle problem, which is NP-complete [8], is reducible to the bridge-connectivity augmentation problem. Let $G = (\mathscr{V}, \mathscr{E})$ be an undirected graph. Construct a bridge-connectivity augmentation problem on vertex set $\mathscr{V}$ with costs $f(v, w) = 1$ if $(v, w) \in \mathscr{E}$, $f(v, w) = 2$ if $(v, w) \notin \mathscr{E}$, and $F = V$. This augmentation problem has a solution with cost F or less if and only if G has a Hamiltonian cycle. This construction is obviously polynomial-time, so the weighted bridge-connectivity problem is NP-complete. □

THEOREM 4. *Let $\mathscr{V}$ be a set of vertices, f a cost function on unordered pairs of distinct vertices and F a total cost. The problem of determining whether there exists a set, with cost F or less, of edges which biconnect the vertices of $\mathscr{V}$ is* NP-*complete.*

Proof. The undirected Hamiltonian cycle problem is reducible to the weighted biconnectivity augmentation problem using the same transformation as in the proof of Theorem 3. □

These constructions can be improved to show that even if the graph G_0 is connected, both the bridge-connectivity and biconnectivity augmentation problems are NP-complete. Thus these weighted augmentation problems are different from the weighted strong connectivity augmentation problem, for which an interesting special case has an efficient algorithm. However, the unweighted bridge-connectivity and biconnectivity problems both have good algorithms. Here we present an $O(V + E)$ algorithm for the bridge-connectivity problem; Pecherer and Rosenthal have developed an $O(V + E)$ algorithm for the biconnectivity problem [11].

3.1. Bridge-connectivity. Let G_0 be any undirected graph. If G_0 has exactly two vertices, then there is no way to bridge-connect the vertices without allowing multiple edges. Thus suppose that $G_0 = (\mathscr{V}, \mathscr{E}_0)$ with $V \geqq 3$.

To find a minimum augmenting set for G_0, we first find the bridge-connected components of G_0 and shrink each to a single vertex to form a graph G_0', called the *condensation of* G_0 *(with respect to bridge-connectivity)*. This transformation requires $O(V + E)$ time [12], [13]. G_0' is a set of trees; each edge in G_0' corresponds to a bridge of G_0 and each vertex in G_0' corresponds to a bridge-component of G_0. Any set of edges which bridge-connects G_0 corresponds in an obvious way to a set of edges which bridge-connects G_0', as described in the result below.

Let $G_0' = (\mathscr{V}', \mathscr{E}_0')$ be the condensation of $G_0 = (\mathscr{V}, \mathscr{E}_0)$. For $v \in \mathscr{V}$, let $\alpha(v)$ be the vertex in G_0' corresponding to the bridge-component in G_0 containing v. For $x \in \mathscr{V}'$, let $\beta(x)$ be any vertex in the bridge-component of G_0 corresponding to x. Clearly $\alpha(\beta(x)) = x$ for all $x \in \mathscr{V}'$.

LEMMA 3. *Let A be a set of edges which bridge-connects G_0. Then $\alpha(A) = \{(\alpha(v), \alpha(w)) | (v, w) \in A, \alpha(v) \neq \alpha(w)\}$ is a set of edges which bridge-connects G_0'. Conversely, if B is a set of edges which bridge-connects G_0', then $\beta(B) = \{(\beta(x), \beta(y)) | (x, y) \in B\}$ is a set of edges which bridge-connects G_0.*

Because of Lemma 3 we can concentrate on G_0'. G_0' contains zero or more vertices with exactly one incident edge, called *pendants*, and zero or more vertices with no incident edges, called *isolated vertices*. The following theorem gives a lower bound on the number of edges needed to make G_0' bridge-connected.

THEOREM 5. *Let G_0' contain p pendants and q isolated vertices, where $p + q > 1$. Then at least $\lceil p/2 \rceil + q$ edges are needed to make G_0' bridge-connected, where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.*

Proof. After G_0' is bridge-connected, each pendant will have at least one new incident edge, and each isolated vertex will have at least two new incident edges. Each new edge can satisfy at most two of these requirements. Thus at least $\lceil p/2 \rceil + q$ edges are needed to bridge-connect G_0'. □

The bound in Theorem 5 is attainable. The rest of this section gives a method for efficiently finding a set of $\lceil p/2 \rceil + q$ edges which bridge-connect G_0'. The method first adds enough edges to make G_0' into a tree and then pairs pendants in a simple fashion.

Let $G_0' = (V', E_0')$. Let n be the number of trees in G_0'. Let $v(i)$, $1 \leqq i \leqq 2n$, be a set of vertices of G_0' such that

(i) $v(2i - 1)$ and $v(2i)$ are each a pendant or an isolated vertex in the ith tree of G_0', for each $1 \leqq i \leqq n$;

(ii) $v(2i - 1) = v(2i)$ if and only if the ith tree of G_0' is an isolated vertex.

Let $T = (V', E_0' \cup A_1)$, where $A_1 = \{v(2i), v(2i + 1) | 1 \leqq i < n\}$. The following result is obvious.

LEMMA 4. *T is a tree having $p' = p + 2q - 2(n - 1)$ pendants and no isolated vertices.*

Note that A_1 contains $n - 1$ edges. Thus all that remains is to find a set of $\lceil p'/2 \rceil$ edges to biconnect any tree T with p' pendants. We assume $p' \geqq 3$; $p' = 2$ is easy.

To accomplish this, we convert T into an arborescence T' by picking an arbitrary nonpendant vertex as root and directing all the edges so that there is a directed path from the root to every vertex. The pendants of T are exactly the vertices with no exiting edges in T'. We then number the vertices in *preorder* [10]. This numbering scheme corresponds to applying the following algorithm to T':

```
algorithm PREORDER: begin
  procedure DFS(v); begin
    number(v) := i := i + 1;
    for w such that (v, w) is an edge do DFS(w);
  end;
  i := 0;
  DFS(r);
  comment r is the root of T';
  end;
```

It is easy to construct T' and number the vertices in preorder in $O(V)$ time [13].

Let $v(1), \cdots, v(p')$ be the pendants of T, ordered so that number$(v(i)) <$ number$(v(i+1))$ for $1 \leqq i \leqq p'$. Let $A_2 = \{(v(i), v(i + \lfloor p'/2 \rfloor)) \mid 1 \leqq i \leqq \lceil p'/2 \rceil\}$, where $\lfloor x \rfloor$ is the greatest integer not greater than x, and $\lceil x \rceil$ is the least integer not less than x. We will show that addition of the edges A_2 to T makes T bridge-connected. We need a little notation and two lemmas. If (v, w) is an edge of T', we write $v \to w$ in T'. If w is reachable from v in T', we write $v \overset{*}{\to} w$. In this case v is an *ancestor* of w and w is a *descendant* of v.

LEMMA 5. *The descendants of any vertex have consecutive numbers in any preorder numbering.*

Proof. See [13].

LEMMA 6. *Let (v, w) be an edge of $G' = (\mathscr{V}', \mathscr{E}'_0 \cup A_1 \cup A_2)$ with* number$(v) <$ number(w). *Then (v, w) is a bridge of G' if and only if $v \to w$ and there is no edge (x, y) such that $w \overset{*}{\to} x$ in T' and $\neg w \overset{*}{\to} y$ in T'.*

Proof. See [13].

LEMMA 7. *$G' = (\mathscr{V}', \mathscr{E}'_0 \cup A_1 \cup A_2)$ is bridge-connected.*

Proof. We must show that G' has no bridges. Let (v, w) be any edge of G' such that $v \to w$. Suppose the pendants of T which are descendants of w are contained in $\{v(1), \cdots, v(\lceil p'/2 \rceil)\}$. Then there is an edge $(x, y) \in A_2$ such that $w \overset{*}{\to} x$ and $\neg w \overset{*}{\to} y$, so (v, w) is not a bridge by Lemma 5. Similarly, if the pendants of T which are descendants of w are contained in $\{v(\lfloor p'/2 \rfloor + 1), \cdots, v(p')\}$ then (v, w) cannot be a bridge.

The last case occurs when some pendant $v(i)$ with $1 \leqq i \leqq \lceil p'/2 \rceil$ and some pendant $v(j)$ with $\lfloor p'/2 \rfloor + 1 \leqq j \leqq p'$ are descendants of w. Then, by Lemma 5, $v(\lceil p'/2 \rceil)$ and $v(\lfloor p'/2 \rfloor + 1)$ are descendants of w. Since the root of T' has degree

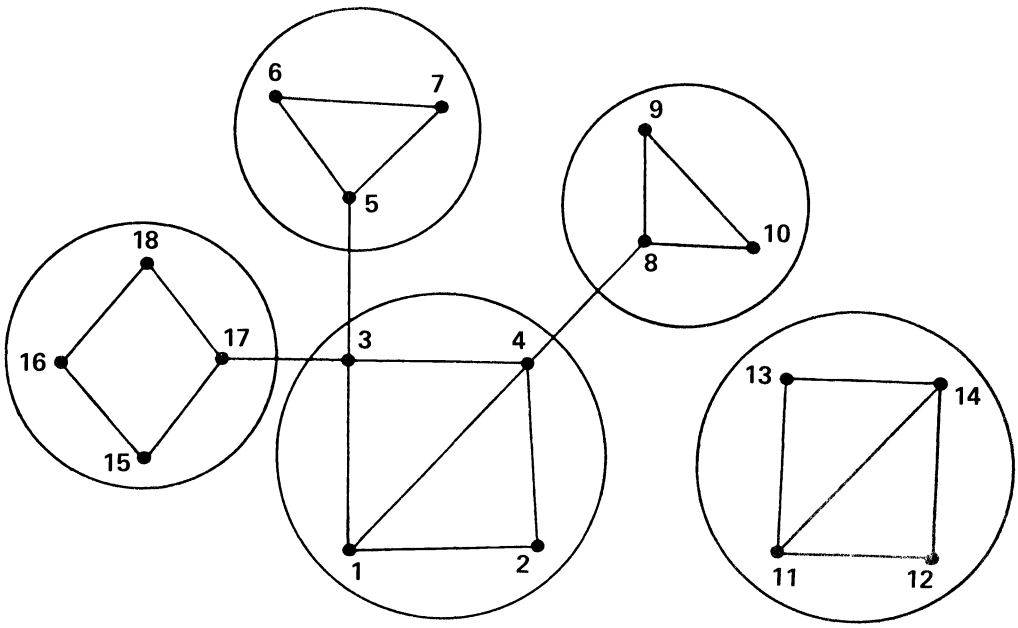


FIG. 3. *Undirected graph we wish to make bridge-connected. Bridge components are circled.*

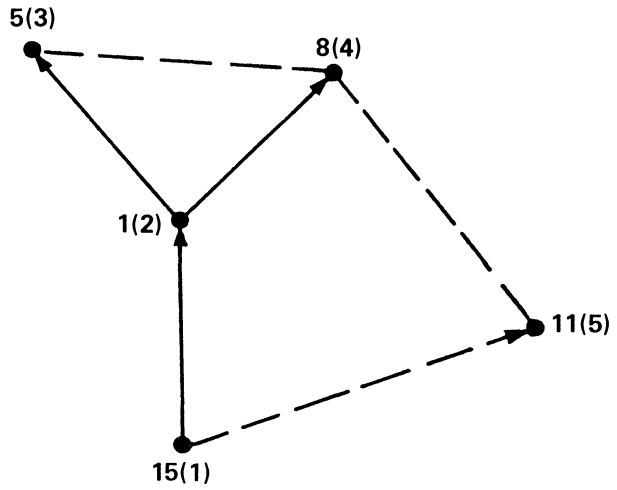


FIG. 4. *Set of trees corresponding to graph in Fig. 3. Graph has three pendants and one isolated vertex. First numbers on vertices are those of arbitrary vertices in the corresponding bridge components. Numbers in parentheses give preorder numbering. Edge* (15, 11) *is in* A_1; (5, 8) *and* (8, 11) *are in* A_2.

two or more, either $v(1)$ or $v(p')$ is not a descendant of w, and there is an edge $(x, y) \in A_2$ such that $w \overset{*}{\rightarrow} x$ and $\neg w \overset{*}{\rightarrow} y$, so (v, w) is not a bridge. □

Since A_2 contains $\lceil p'/2 \rceil$ edges, A_1 and A_2 together contain $\lceil p/2 \rceil + q$ edges and form a minimum augmenting set for G_0'. Combining the steps discussed above, we get the following bridge-connectivity augmentation algorithm.

algorithm BRCONNECT; **begin**
BR1: find the bridge-connected components of G_0;
BR2: condense G_0 into an acyclic graph G_0' by collapsing each component subgraph of G_0 into a single vertex;
BR3: construct a set A_1 of edges to connect G_0' into a tree T;
BR4: convert T into an arborescence, number its vertices in preorder, and order the pendants of T by preorder number;
BR5: construct A_2 to bridge-connect T;
BR6: convert $A = A_1 \cup A_2$ into a minimum set of edges which bridge-connect G_0 by using Lemma 3;
end BRCONNECT;

It is clear that BRCONNECT requires $O(V + E)$ time and space. Figures 3–4 show the application of this algorithm to an example graph.

3.2. Biconnectivity. Let G_0 be a graph we want to make biconnected by adding a minimum number of edges. In this section we characterize the number of edges which must be added. G_0 has a number of biconnected components (or *blocks*). We call a block of G_0 *isolated* if it contains no cutnodes, and *pendant* if it contains exactly one cutnode. Let p be the number of pendant blocks and let q be the number of isolated blocks of G_0.

Let v be any node of G_0. We can divide the edges of G_0 into equivalence classes E_i such that two edges (x, y), (w, z) are in the same equivalence class if and only if

there is a path $(v_1, v_2), \cdots, (v_{k-1}, v_k)$ containing (x, y) and (w, z) but such that $v_i \neq v$ for $2 \leqq i \leqq k-1$. Each equivalence class E_i defines a subgraph $G_i = (V_i, E_i)$ of G_0, where $V_i = \{x \in V \mid (x, y) \in E_i\}$. We call the subgraphs G_i the *v-blocks* of G_0. Let $d(v)$ be the number of v-blocks of G_0. Then $d(v) > 1$ if and only if v is a cutnode. If G contains n connected components and v is contained in b blocks, then $d(v) = b + n - 1$. Let $d = \max \{d(v) \mid v \in V\}$.

Suppose G_0 is connected and consider two pendant blocks of G_0 which lie in different v-blocks. Suppose we connect these two pendant blocks by an edge not connecting cutnodes. Consider the resulting graph G_0'. G_0' has the same v-blocks as G_0 with the exception that the two v-blocks containing the joined pendant blocks are coalesced into one new v-block. G_0' has one pendant block in the new v-block for each pendant block in the old v-blocks except for the two pendant blocks newly joined, unless the old v-blocks both contain only one pendant block. In this case, the new v-block has exactly one pendant block representing the entire v-block. This observation forms the basis for Theorem 7 below.

THEOREM 6. *At least* $\max(d - 1, \lceil p/2 \rceil + q)$ *edges are needed to make* G_0 *biconnected if* $p + q > 1$.

Proof. Since any biconnected graph is bridge-connected, Theorem 5 implies that at least $\lceil p/2 \rceil + q$ edges are necessary to biconnect G_0. Let v be a cutnode of G_0 with $d(v) = d$. Deletion of v from G_0 leaves d connected components. At least $d - 1$ edges are necessary to connect these components. □

The bound in Theorem 6 is attainable, as we show in the next theorem.

THEOREM 7. *We can biconnect* G_0 *by adding* $\max(d - 1, \lceil p/2 \rceil + q)$ *edges.*

Proof. The proof is by induction on $i = \max(d - 1, \lceil p/2 \rceil + q)$. If G_0 is biconnected, there is nothing to prove. If G_0 is not biconnected, $i \geqq 1$. It is easy to verify the theorem for $i = 1$. Suppose the theorem is true if $\max(d - 1, \lceil p/2 \rceil + q) = i \geqq 1$. Let $\max(d - 1, \lceil p/2 \rceil + q) = i + 1$. We must consider several cases.

(i) G_0 is not connected. Pick two connected components of G_0. Each is either an isolated block or contains a pendant block. Join an isolated or pendant block in one component to an isolated or pendant block in the other component by a single edge not connecting cutnodes. The resulting graph G_0' has $\max(d' - 1, \lceil p'/2 \rceil + q') = i$. By the induction hypothesis we need only i more edges to biconnect G_0.

(ii) G_0 is connected. Then $q = 0$. There are three subcases.

(a) $d - 1 > \lceil p/2 \rceil$. There can only be one cutnode v with $d(v) = d$. Pick any two pendant blocks of G_0 which lie in different v-blocks in G_0 and join them by an edge not connecting cutnodes. The resulting graph G_0' has $\max(d' - 1, \lceil p'/2 \rceil) = i$. By the induction hypothesis we can biconnect G_0 with i more edges.

(b) $d - 1 = \lceil p/2 \rceil$. There can be at most two cutnodes v with $d(v) = d$. Choose such a cutnode v. Since $d - 1 = \lceil p/2 \rceil = i + 1 \geqq 2$, there must be some v-block of G_0 which contains two or more pendant blocks of G_0. Pick such a v-block G_1. If some other cutnode w satisfies $d(w) = d$, then w must be in G_1. Furthermore, each pendant block of G_0 must be the unique pendant block in either a v-block or a w-block.

Connect any pendant block in G_1 to any pendant block in some other v-block, using a single edge not joining cutnodes. This reduces the number of pendant

blocks by two. It also reduces the number of v-blocks by one. Finally, if another cutnode w satisfies $d(w) = d$, addition of the edge reduces the number of w-blocks by one. Thus the resulting graph G_0' has $\max(d' - 1, \lceil p'/2 \rceil) = i$, and by the induction hypothesis we can biconnect G_0 with i more edges.

(c) $d - 1 < \lceil p/2 \rceil$. There must be a cutnode v in G_0 having a v-block which contains two or more pendant blocks of G_0. Connect one of the pendant blocks of G_0 in such a v-block to a pendant block in some other v-block, using an edge not joining cutnodes. The resulting graph G_0' has two less pendant blocks than G_0, and by the induction hypothesis we can biconnect G_0 with i more edges. □

The proof of Theorem 7 leads to an algorithm for finding a minimum augmenting set of edges to biconnect a graph. This algorithm will run in $O(V + E)$ time if properly implemented [11], but the program is fairly complicated because some involved list-processing must be used. We leave open the problem of determining a simple $O(V + E)$ algorithm for finding a minimum augmenting set of edges to biconnect a graph.

REFERENCES

[1] S. Cook *The complexity of theorem-proving procedures*, Proc. 3rd Ann. ACM Symp. on the Theory of Computing, Shaker Heights, Ohio, 1971, pp. 151–158.

[2] J. Edmonds, *Optimum branchings*, J. Res. Nat. Bur. Standards Sect. B, 71 (1967), pp. 233–240.

[3] K. Eswaran, *Representation of graphs and minimally augmented Eulerian graphs with applications in data base management*, Rep. RJ 1305, IBM Research, Yorktown Heights, N.Y., 1973.

[4] H. Frank and W. Chou, *Connectivity considerations in the design of survivable networks*, IEEE Trans. Circuit Theory, CT-17 (1970), pp. 486–490.

[5] S. E. Goodman and S. T. Hedetniemi, *On the Hamiltonian completion problem*, presented at the Capitol Conf. on Graph Theory and Combinatorics, Washington, D.C., 1973.

[6] J. Hopcroft and R. Tarjan, *Efficient algorithms for graph manipulation*, Comm. ACM, 16 (1973), pp. 372–378.

[7] D. B. Johnson, *Algorithms for shortest paths*, Rep. TR 73-169, Dept. of Computer Sci., Cornell Univ., Ithaca, N.Y., 1973.

[8] R. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[9] A. Kershenbaum and R. Van Slyke, *Computing minimum spanning trees efficiently*, Proc. 25th Ann. Conf. of the ACM, 1972, pp. 518–527.

[10] D. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[11] R. Pecherer and A. Rosenthal, private communication, Univ. of Calif., Berkeley, March 1973.

[12] R. Tarjan, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[13] ———, *A note on finding the bridges of a graph*, Information Processing Lett., 2 (1974), pp. 160–161.

[14] ———, *Testing graph connectivity*, Proc. 6th Ann. ACM Symp. on the Theory of Computing, Seattle, Washington, May 1974, pp. 185–193.

[15] ———, *Finding optimum branchings*, Networks, to appear (1975).

[16] ———, *Finding minimum spanning trees*, Electronics Research Laboratory Memo no. M-501, Univ. of California, Berkeley, 1975.

[17] A. Yao, *An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees*, Information Processing Lett., 4 (1975), pp. 21–23.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

SIMPLE PROGRAMS REALIZE EXACTLY PRESBERGER FORMULAS*

JOHN C. CHERNIAVSKY†

Abstract. A class of programs, L_+, is introduced which lies between L_1 and L_2 in the loop program hierarchy. As a tool for analysis of this class, we use formulas in Presberger's arithmetic and we show that L_+ realizes these formulas both relationally and functionally. Moreover, the realization is exact in the sense that every program in L_+ realizes some Presberger formula. Using known decidability and complexity results for Presberger's arithmetic, we are able to obtain a decision procedure for equivalence of L_+ programs and to obtain upper bounds for the complexity of this procedure.

Key words. logic, computational complexity, program semantics, flow chart schemata

Introduction. A key problem facing computer scientists is the formulation of meaningful and mathematically precise semantics for computer programs. This problem runs a spectrum of difficulty ranging from the development of semantics for regular languages to the development of semantics for high level languages such as ALGOL 68 and PL/1. We propose to study the semantics of a simple programming language L_+.

The semantics of L_+ will be related, in a very precise manner, to functions and relations definable in Presberger's theory of addition, P_+. The motivation for the study was twofold. First, we were interested in classes of programming languages with decidable equivalence problems. Rather than attempt to use the usual techniques for their analysis, we decided to try and use a first order decidable logical theory, for the logical theory provides a ready-made formalism for describing the semantics of the programming language. Second, we wanted to develop a purely computational interpretation of P_+, that is, a programming language which "realized" formulas in P_+. We were able to show that the pair L_+, P_+ satisfied both criteria; that is, P_+ acts as a formalism to describe the semantics of L_+ and L_+ acts as a mechanistic description of P_+. We call such a realization of a logic by a class of programs an *exact* realization.

The notion of realization was originally introduced by Kleene [7] to differentiate between classical and intuitionistic provability in number theory. Starting from a sentence in intuitionistic number theory (INT), a Gödel number for the sentence can be effectively calculated from a proof in INT of the sentence. The Gödel number refers to a program that duplicates the intuitionistic proof in the sense that constructivity is interpreted as computability. In essence, an interpretation of the provable formulas in a logic is provided using programs (this is inverse to the usual procedure where interpretations for programs are obtained as formulas, to be proven in a logic). We will introduce a notion of realizability motivated both by Kleene's considerations and semantic considerations.

Let T be a logical theory; let $F(T)$ be the first order formulas constructed from function and predicate symbols occurring in T; let $I = \langle D, \{f_i\}_{i\in I}, \{p_j\}_{j\in J}\rangle$

* Received by the editors December 2, 1974, and in revised form September 23, 1975.

† Department of Computer Science, SUNY at Stony Brook, Stony Brook, New York 11794. This research was supported by the National Science Foundation under Grant GJ-43186.

be a model for T; let L be a set of programs computing over D. We say that L *realizes* T *relative to* I if for every formula $Q(x_1, \cdots, x_n)$ in $F(T)$ there is a program π in L with input variables $x_1, \cdots, x_n$ such that for $d_1, \cdots, d_n$ in D, $Q(d_1, \cdots, d_n)$ is true in I iff π halts on inputs $d_1, \cdots, d_n$. The realization is *exact* if for every program π in L with inputs $x_1, \cdots, x_n$ there is a formula $Q_\pi(x_1, \cdots, x_n)$ in $F(T)$ such that π realizes Q_π.

L is said to *functionally realize* T *relative to* I if for every function f in $D^n \to D$ represented by a formula $Q_f(x_1, \cdots, x_n, z)$ in $F(T)$ (i.e., $Q_f(d_1, \cdots, d_n, c)$ is true in I iff $f(d_1, \cdots, d_n) = c$), there is a program π_Q with inputs $x_1, \cdots, x_n$ and output z such that $\pi_Q(d_1, \cdots, d_n) = c$ iff $Q_f(d_1, \cdots, d_n, c)$ is true in I. A functional realization is *exact* if for every program π with inputs $x_1, \cdots, x_n$ and output z, there is a formula $Q_\pi(x_1, \cdots, x_n, z)$ in $F(T)$ such that π functionally realizes Q_π.

In this paper we shall be investigating the semantics of a language L_+ that essentially lies between Meyer's and Ritchie's [9] L_1 and L_2. Following their notation, the functions computed by programs in L_+ will be denoted by $\mathscr{L}_+$. We will show that L_+ has an exact realization in the theory P_+. Since properties of this theory are quite well known (and the theory is decidable), it provides a good semantics for L_+. In addition, we will have extended the decidability result of Tsichritzis [11] from L_1 to L_+ and thus further refined the boundary between decidable and undecidable equivalence problems for subrecursive classes. As an added benefit, using known complexity results for equivalence in P_+, we are able to obtain complexity bounds for equivalence in L_+ and in L_1. Finally, we will have obtained yet another characterization of P_+, thus further refining our notions of the power of this theory.

1. The theory P_+. P_+ is the theory for the structure $\langle \mathbb{N}, 0, +1, +, <, = \rangle$ and is a conservative extension of the original Presberger theory (for axioms, see [8]). Historically this theory is important because the solution to its decision problem was interpreted by many as a positive step in Hilbert's program. Today it remains as one of the stronger logical theories with a decidable equivalence problem. In addition, it has been of recent interest for problems associated with its complexity, as evidenced by the research of Oppen [10] and Fischer and Rabin [6]. It should be remarked that the decision procedure for L_+ has some striking resemblances to that developed for L_1. However, L_1 is not a realization for P_+, as will be shown below.

Because of its key role in one of the embeddings, we feel that a brief presentation of some elements of the quantifier elimination procedure for P_+ is in order (see [5] for more details). P_+ is somewhat peculiar because the quantifier elimination cannot be done in P_+ but must be done in a conservative extension of P_+, P_+^*. The language for P_+^* includes all the symbols of P_+ with the addition of new symbols $\equiv_2, \equiv_3, \cdots$ which are to represent binary relations corresponding to equivalence modulo 2, modulo 3, etc. That the theory resulting is a conservative extension of P_+ can easily be seen by translating "$t_1 \equiv_m t_2$" to "$\exists b_1 \exists b_2((t_1 = mb_1 \,\&\, t_2 = mb_2) \vee (t_1 = mb_1 + 1 \,\&\, t_2 = mb_2 + 1) \vee \cdots \vee (t_1 = mb_1 + m - 1 \,\&\, t_2 = mb_2 + m - 1))$", where abbreviations have been freely used and t_1, t_2 are terms.

Beginning with a formula $Q(x_1, \cdots, x_n)$ in $F(P_+)$ and just before elimination of variable z, a disjunction of formulas of the following form is obtained:

$$\exists z[\bigwedge_{i<l} r_i < z + s_i \,\&\, \bigwedge_{j<k} z + u_j < t_j \,\&\, \bigwedge_{i<n} x + w_i \equiv_{m_i} v_i]$$
$$\&\ [\text{Boolean combination of atomic formulas not involving } z]$$

where $r_i, s_i, u_j, t_j, w_i, v_i$ are terms in P_+. After all quantifiers have been eliminated in Q, the result is a Boolean combination of atomic formulas of the form $t_i = t_j$, $t_i < t_j$, $t_i \equiv_m t_j$, where the terms have free variables from $\{x_1, \cdots, x_n\}$.

2. The programming language L_+. The language L_+ is essentially a language in which no assignment statement is embedded in more than one loop. In order to make this more precise, we must define "loop" more precisely.

DEFINITION. A *cycle* in a labeled directed graph is a sequence of nodes $a_1, \cdots, a_n$, such that there is an edge leading from a_i to a_{i+1}, $1 \leqq i < n$, and $a_1 = a_n$.

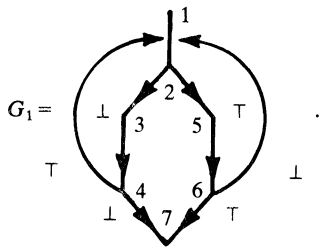
DEFINITION. A *subcycle* of a cycle $a_1, \cdots, a_n$ is a sequence of nodes $b_1, \cdots, b_k$ such that for some a_l, $a_l = b_1$, $a_{l+1} = b_2, \cdots, a_{l+k-1} = b_k$ and $b_1, \cdots, b_k$ is a cycle.

DEFINITION. A *canonical cycle* is a cycle with no identical subcycles.

A P_{NIL} *graph* G is defined as a connected labeled directed graph G with a distinguished source node of outdegree 1 (the start node), distinguished sink nodes (halt and loop nodes), nodes of outdegree 1 (operational nodes), and nodes of outdegree 2 (test nodes) for which one branch is labeled "$\perp$" and the other "$\top$". P_{NIL} graphs satisfy:

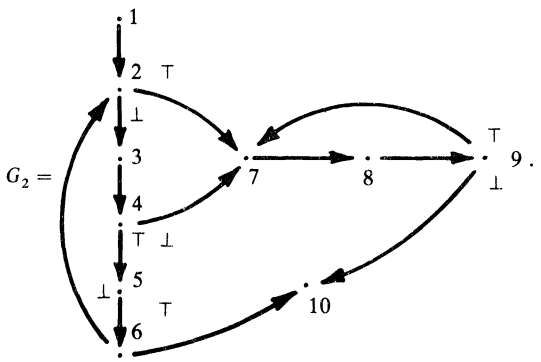
(A) No operational node is contained in two or more distinct canonical cycles of G.

Example 1. Let



The start node of G_1 is 1; the sink node is 7; the operational nodes are 3, 5 and the test nodes 2, 4, 6. G_1 is not a P_{NIL} graph for 3 is in the canonical cycles 2342 and 2342562.

Example 2. Let



The start node is 1; the sink node is 10; the operational nodes are 3, 5, 7, 8 and the test nodes are 2, 4, 6, 9. G_2 is a P_{NIL} graph for the only canonical cycles are 234562 and 7897 and they have no operational nodes in common.

An L_+ *program* is then defined as a P_{NIL} graph with source node START; sink nodes HALT and LOOP; operational nodes $x \leftarrow y + 1$, $x \leftarrow y \dot{-} 1$, $x \leftarrow y$ and $x \leftarrow 0$; and test nodes $x = 0$, where variables are drawn from a countable set of names and $y \dot{-} 1$ is defined as $y - 1$ if $y > 0$; 0 otherwise. The other operations have their standard interpretations.

The set of input variables to an L_+ program is a set of variables, included in the program's variables, that include those variables that first occur on the right-hand side of an assignment statement within some path through the program, or which first occur in a test node. The set of output variables may be defined arbitrarily as a set of variables included in the program's variables; however, in order to ensure that the only undefined computations are those that encounter a loop node or those that never halt, we assume that the set of output variables is contained within the set of input variables (and hence they always have a value). Execution of an L_+ program is defined in the natural manner, where flow of control is determined by the connections of the underlying P_{NIL} graph. The interpretations of the operational and test statements are again natural. The statements in L_+ can be weakened somewhat by noticing that $x \leftarrow y + 1$ can be replaced by $x \leftarrow x + 1$ and $x \leftarrow y \dot{-} 1$ can be replaced by $x \leftarrow x \dot{-} 1$, the former statements being simulated by the introduction of extra variables.

The total functions in $\mathscr{L}_+$ lie between $\mathscr{L}_1$ and $\mathscr{L}_2$, where $\mathscr{L}_1$ and $\mathscr{L}_2$ are the functions computed by programs defined in a program hierarchy due to Meyer and Ritchie [9]. The statement types allowed in programs in this hierarchy consisted of the following:

(i) $x \leftarrow 0$,
(ii) $x \leftarrow x + 1$,
(iii) $x \leftarrow y$,
(iv) LOOP x,
(v) END,

where every LOOP x is paired with an END.

L_0 is defined as the set of programs consisting of statement types from (i)–(iii). L_n is the set of programs containing L_{n-1} in which every LOOP x, END pair encloses an L_{n-1} program. Given a value $\bar{x}$ for x, the interpretation of LOOP x α END, for α a program, is that α is executed exactly $\bar{x}$ times. The interesting facts, for our purposes, about these classes is that the equivalence problem for L_1 programs is decidable [11], but is undecidable for L_2 programs [9].

It is easy to see that L_+ programs contain L_1 programs. The statement types (i)–(iii) are in L_+ and the following code simulates LOOP x α END, for α in L_0:

```
        count ← x;
L2:     if count = 0 then go to L1;
        α
        count ← count ∸ 1;
        go to L2;
L1:
```

Since α is in L_0, no other looping (and hence no other cycles) can occur. Hence no canonical cycles intersect and the resultant program is in L_+.

What makes L_+ interesting is that it computes functions not computable in L_1. This is because of two features:

(i) $\mathscr{L}_+$ contains partial functions ($\mathscr{L}_1$ functions are total).

(ii) $x \dot{-} y$ is in $\mathscr{L}_+$ but not in $\mathscr{L}_1$.

Feature (i) tells us that $\mathscr{L}_+$ is not really a subrecursive class (though trivially so). Feature (ii) says that $\mathscr{L}_+$ is interesting for its total functions properly contain $\mathscr{L}_1$.

LEMMA 1. *$x \dot{-} y$ is not in $\mathscr{L}_1$.*

Proof. As a consequence of Theorem 5 in Tsichritzis [11], $x \dot{-} y$ is not simple and hence not in $\mathscr{L}_1$. □

All that remains to be shown is that $\mathscr{L}_+$ is properly contained in $\mathscr{L}_2$ (for total $\mathscr{L}_+$ functions). This result depends upon Theorems 4 and 5 below and their proofs. The theorems state that L_+ is an exact functional realization of P_+. To prove that $\mathscr{L}_+ \subseteq \mathscr{L}_2$, all that needs to be done is to replace the L_+ programs by L_2 programs in the proof of Theorem 5. The easiest way to do this is to use a result of Constable and Borodin [3] and to replace L_+ programs by SR_1 programs; we leave it to the reader to fill in the details. To show the containment is proper, all that is necessary is to exhibit an $\mathscr{L}_2$ function that is not $\mathscr{L}_+$ computable. By Theorems 4 and 5, this is equivalent to showing that the $\mathscr{L}_2$ function is not P_+ expressible. One such function is $x * y = z$, for there is no formula $Q(x, y, z)$ in P_+ such that $Q(x, y, z)$ is true exactly when $x * y = z$ [5].

3. L_+ exactly realizes P_+. As indicated in the Introduction, there are two notions of realizability: a relational notion and a functional notion. L_+ realizes P_+ in both senses.

DEFINITION. A program α in L_+ with inputs $x_1, \cdots, x_n$ *realizes* a relation on $\mathbb{N}$ described by $Q(x_1, \cdots, x_n)$ in $F(P_+)$ iff $\forall a_1, \cdots, a_n$ in $\mathbb{N}$ ($Q(a_1, \cdots, a_n)$ is valid $\Leftrightarrow \pi\ (a_1, \cdots, a_n)$ halts).

Our problem is to prove that every program in L_+ realizes some P_+ describable relation and that every relation described by a formula in P_+ is realized by a program in L_+.

THEOREM 1. *Every program in L_+ realizes a relation described by a formula in $F(P_+)$. In addition, if n is the length of the program, the length of the formula is at most $O(2^{2^{p(n)}})$, where p is a polynomial.*

Proof. The proof of the first part of the theorem is in [2]. For the purpose of calculating complexity bounds, we will present an outline of the proof.

The proof is a two step process. First we use a normal form reduction from P_{NIL} flowchart schemata (i.e., schemata defined on P_{NIL} graphs) to formulas in a two-sorted logic. The functions and predicate symbols of these formulas are then interpreted in accordance with the interpretations of functions and predicates in L_+. In the second step, the resulting formulas are translated to P_+ formulas.

In more detail, the first step of the process consists of the following substeps:

(a) A translation from a P_{NIL} graph to its Engeler normal form.

(b) An expansion of the Engeler normal form graph by identification of loops and discrimination of periodic outputs of these loops.

(c) For each path in the resulting tree, the construction of a conjunction, each element arising from a node in the tree (these elements are the iterated expressions described below). A disjunction is then taken over all such conjunctions.

Given a program of length n, step (a) results in a program of length $O(2^n)$. In step (b), the identification of a loop as a node along with the discrimination of outputs can result in the spawning of $O(n^n)$ subtrees below the identified node (shown in [2]). Thus the size of the tree produced in step (b) is $O(2^{2^{p(n)}})$, where p is a polynomial. Each branch in the tree still has length at most n. In step (c), each element of the conjunction is linearly related to n in length; thus the resultant formula has size $O(2^{2^{p(n)}})$.

After interpreting the function and predicate symbols according to those in L_+, our atomic formulas have the form "$t_i < t_j$", "$t_i = t_j$" and "$\alpha_1^{i_1} \circ \cdots \circ \alpha_k^{i_k} x = 0$", where t_i and t_j are terms constructed from $+1, 0$ and variables ranging over $\mathbb{N}$; $x, i_1, \cdots, i_k$ are variables ranging over $\mathbb{N}$; and α_i is a composition of the functions $+1$, $\dot{-}1$ and $\bar{0}$ (where $\bar{0}(x) = 0$ for all x). Given a value $\bar{\imath}$ for i, the interpretation of $\alpha^i x$ is the $\bar{\imath}$-fold composition of α applied to x.

The final step of the construction is to convert atomic formulae of the form $\alpha_1^{i_1} \circ \cdots \circ \alpha_n^{i_n} = 0$ to P_+^* formulas. This conversion is a straightforward (although lengthy) recursive argument and results in a P_+^* formula of length $O(2^n)$, where n is the length of the atomic formula. This results in a multiplicative expansion of $O(2^n)$ on the formula resulting from the first step of the process. Thus the representing formula has length $O(2^{2^{p(n)}})$, where n is the length of the original program. Since P_+^* is a conservative extension of P_+, there is a P_+ formula representing the same relation. □

We must next show that every P_+ formula is realized by an L_+ program. The idea is to perform an elimination of quantifier procedure on the P_+ formula, obtaining a formula in P_+^*. Next we put this formula in disjunctive normal form and then construct the L_+ program. The final result is obtained after the proof of a sequence of lemmas.

LEMMA 2. *All P_+ terms are L_+ computable.*

Proof. We proceed inductively on the number of function occurrences in the P_+ term. We let z represent the output variable.

Case 0. The term is either "0" or "x_i", and respectively "$z \leftarrow 0$" or "$z \leftarrow x_i$" suffices.

Case n. All terms with n or less occurrences of functions are L_+ computable.

Case $n + 1$. Let $f \circ \alpha x_i$ be a term of P_+, where α has n occurrences of P_+ functions and $f = +0, +1$ or $+x_j$. Let $\bar{z}$ be the output of the program to compute αx_i and let $\bar{x}_j$ be a new variable; then:

(a) αx_i program
$z \leftarrow \bar{z}$;
HALT

(b) αx_i program
$z \leftarrow \bar{z}$;
$z \leftarrow z + 1$;
HALT

(c) αx_i program
$z \leftarrow \bar{z}$;
$\bar{x}_j \leftarrow x_j$;
L: if $\bar{x}_j = 0$ then go to L'
$z \leftarrow z + 1$;
$\bar{x}_j \leftarrow \bar{x}_j \dot{-} 1$;
go to L;
L': HALT

suffice to cover the respective cases. □

LEMMA 3. *The formula* $\wedge_{i<l} r_i < s_i$ *is realized by a program in* L_+ (r_i, s_i *are terms*).

Proof. Let $\underline{r_i}$ and $\underline{s_i}$ be the outputs of programs to compute r_i and s_i. The following program, α_i, realizes one of the conjuncts:

1. e_i^1: compute $\underline{r_i}$ and $\underline{s_i}$
2. L_i^1: if $\underline{s_i} = 0$ then LOOP;
3. if $\underline{r_i} = 0$ then go to e_{i+1}^1;
4. $\underline{r_i} \leftarrow \underline{r_i} \dot{-} 1$;
5. $\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
6. go to L_i^1

The entire conjunction is realized by

$$\alpha_0; \alpha_1; \cdots \alpha_{l-1}; e_l^1 : \text{HALT}$$ □

LEMMA 4. *The formula* $\wedge_{i<k} r_i = s_i$ *is* L_+ *realizable.*

Proof. The proof is the same as for Lemma 3 except that the labels e_i^1, L_i^1 are replaced by e_i^2, L_i^2 and lines 2 and 3 are replaced by:

L_i^2: if $\underline{s_i} = 0$ then if $\underline{r_i} = 0$ then go to e_{i+1}^2 else LOOP;
if $\underline{r_i} = 0$ then LOOP; □

LEMMA 5. *The formula* $\wedge_{i<n} r_i \equiv_{m_i} s_i$ *is* L_+ *realizable.*

Proof. Let r_i and s_i be as in Lemma 3. The program for each conjunct consists of two parts. The first is a "decision table" which checks what $\underline{r_i}$ is congruent to among $\{0, 1, \cdots, m_i - 1\}$. If it is congruent to k, a branch is made to b_k which checks if $\underline{s_i}$ is congruent to k also. For each conjunct, construct a program β_i as follows:

e_i^3: compute $\underline{r_i}$ and $\underline{s_i}$
L_i^3: if $\underline{r_i} = 0$ then go to b_0^i;
$\underline{r_i} \leftarrow \underline{r_i} \dot{-} 1$;
⋮
if $\underline{r_i} = 0$ then go to $b_{m_i-1}^i$;
$\underline{r_i} \leftarrow \underline{r_i} \dot{-} 1$;
go to L_i^3;
b_0^i: if $\underline{s_i} = 0$ then go to e_{i+1}^3;
$\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
if $\underline{s_i} = 0$ then LOOP;
$\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
} $m_i - 1$ repetitions
go to b_0^i;
⋮
b_k^i: if $\underline{s_i} = 0$ then LOOP;
$\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
} k repetitions
if $\underline{s_i} = 0$ then go to e_{i+1}^3;
$\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
if $\underline{s_i} = 0$ then LOOP
$\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
} $m_i - k - 1$ repetitions
go to b_k^i;
⋮
go to $b_{m_i-1}^i$;

The program for the entire conjunction then consists of

$$\beta_0 ; \beta_1 ; \cdots \beta_{n-1} ; e_n^3 : \text{HALT}. \qquad \square$$

THEOREM 2. *Every formula $Q(x_1, \cdots, x_n)$ in P_+ is realized by a program in L_+.*

Proof. From $Q(x_1, \cdots, x_n)$ in P_+, apply the elimination of quantifier procedure to obtain an equivalent formula $Q'(x_1, \cdots, x_n)$ in P_+^*. Put Q' in dnf form, with negations of atomic formulas eliminated (as in the quantifier elimination procedure [5]), and obtain a disjunction of m conjunctions, each of which has the form

$$\text{“}(\bigwedge_{i<l} r_i < s_i \,\&\, \bigwedge_{j<k} t_j = u_j \,\&\, \bigwedge_{i<n} v_i \equiv_{m_i} w_i)\text{”}.$$

For each such conjunction i, $1 \leqq i \leqq m$, use Lemmas 3, 4 and 5 to obtain programs γ_i, δ_i and μ_i which realize

$$\bigwedge_{i<l} r_i < s_i, \quad \bigwedge_{j<k} t_j = u_j \quad \text{and} \quad \bigwedge_{i<n} v_i \equiv_{m_i} w_i,$$

respectively. In γ_i and δ_i, change the statements labeled by e_l^1 and e_k^2 respectively from “HALT” to “go to e_0^2” and “go to e_0^3”. Call each of these programs ξ_i, $1 \leqq i \leqq m$. To the programs ξ_i, $2 \leqq i \leqq m$, add a label e_i^4 to their first statements. Finally, to the programs ξ_i, $1 \leqq i \leqq m-1$, change all “LOOP” statements to “go to e_{i+1}^4”. The resulting program realizes Q'. $\square$

Theorems 1 and 2 now allow us to state:

THEOREM 3. *L_+ is an exact realization of P_+.*

The proof for functional realization is somewhat different. Theorem 4, which will correspond to Theorem 1, will be proven by a reduction to Theorem 1. Theorem 5, which will correspond to Theorem 2, will be proven by a construction driven by an elimination of quantifiers.

THEOREM 4. *Every program π with inputs $x_1, \cdots, x_n$ and output x_{out} functionally realizes some formula in $F(P_+)$.*

Proof. Modify the program π so it contains a new input variable z. In place of each HALT statement in π, insert the following code:

L: if $x_{\text{out}} = 0$ then if $z = 0$ then HALT else LOOP;
if $z = 0$ then LOOP;
$x_{\text{out}} \leftarrow x_{\text{out}} \dot{-} 1$;
$z \leftarrow z \dot{-} 1$;
go to L;

The modified program, π', has input variables $x_1, \cdots, x_n, z$ and for values $a_1, \cdots, a_n, b$ in $\mathbb{N}$, $\pi'(a_1, \cdots, a_n, b)$ halts $\Leftrightarrow \pi(a_1, \cdots, a_n) = b$. Use Theorem 1 to obtain from π' a P_+^* formula which realizes π' and which also, by the above construction, is functionally realized by π. Since P_+^* is a conservative extension of P_+, there is a P_+ formula which is functionally realized by π. $\square$

THEOREM 5. *Given any formula $Q(x_1, \cdots, x_n, z)$ in $F(P_+)$ describing some function over $\mathbb{N}$, there is a program π with inputs $x_1, \cdots, x_n$ and output z such that π functionally realizes Q.*

Proof. At the beginning, we proceed much as in Theorem 2. We obtain from Q a formula Q' in P_+^* in which all quantifiers have been eliminated; Q' has free variables $x_1, \cdots, x_n, z$, and we want to express the functional dependence of z

on $x_1, \cdots, x_n$. We designate one of the input variables as an output variable and call it x_{out}. From § 1 we see that the functional dependence is expressed via disjunctions of sets of equations of the form

$$\left(\bigwedge_{i<l} r_i < z' + s_i \,\&\, \bigwedge_{j<k} z' + u_j < t_j \,\&\, \bigwedge_{i<n} z' + w_i \equiv_{m_i} v_i\right)$$

& conjunction of atomic formula not involving z'.

The variable z' is a constant multiple of z, $k \cdot z$, for which the constant k and the change of variable name arises from the collection of terms involving z. Any solution z' to the above equations is guaranteed to be divisible by k, for the congruence $z' \equiv_k 0$ is introduced during the collection process.

Letting $M = \prod_{i<n} m_i$, our procedure is quite simple. From the terms r_i and s_i, we calculate the minimum z' satisfying $\wedge_{i<l} r_i < z' + s_i$. We then check $z', z' + 1, \cdots, z' + M - 1$ and find the minimum value which satisfies

$$\text{(i)} \bigwedge_{j<k} z' + u_j < t_j \quad \text{and} \quad \text{(ii)} \bigwedge_{i<n} z' + w_i \equiv_{m_i} v_i.$$

Since Q described a function and because of the Chinese remainder theorem, if this conjunction is satisfiable it will be satisfied by exactly one of $z', z' + 1, \cdots, z' + M - 1$. The next step is to ensure that the conjunction of formulas not involving z' is satisfied (this construction is the same as in Theorem 2), finally the calculated z' is divided by k to yield the value assigned to x_{out}. If failure occurs at any step, we LOOP. The final program is obtained by connecting the programs constructed for each conjunction as in Theorem 2 to express the full disjunction.

From the equations it is clear that:

(i) $z' > \max\{r_i \dot{-} s_i\}, i < l$, and

(ii) $z' < \min\{t_j \dot{-} u_j\}, j < k$.

We let $p_1 = \max\{r_i \dot{-} s_i\}$ and $p_2 = \min\{t_j \dot{-} u_j\}$, and we show how to calculate p_1 and p_2. We compute $r_i \dot{-} s_i$ using the following program:

e_i: compute $\underline{r_i}$;
compute $\underline{s_i}$;
$n_i \leftarrow 0$;
L_i: if $\underline{r_i} = 0$ then go to L_{i+1};
if $\underline{s_i} = 0$ then $n_i \leftarrow \underline{r_i}$; go to L_{i+1};
$\underline{r_i} \leftarrow \underline{r_i} \dot{-} 1$;
$\underline{s_i} \leftarrow \underline{s_i} \dot{-} 1$;
go to L_i;

Calling the above program α_i, $\{r_i \dot{-} s_i\}$ is computed and stored in $\{n_i\}$ by $\alpha_1; \cdots; \alpha_{l-1}$; e_l: HALT. To find maximum $\{n_i\}$ we construct programs $\beta_{i,j}$, $i < j < l$ as follows:

$e_{i,j}$: $n_{\max} \leftarrow n_i$;
count $\leftarrow n_j$;
$L_{i,j}$: if count $= 0$ then go to $e_{i,j+1}$;
if $n_{\max} = 0$ then go to $e_{j,j+1}$;
count $\leftarrow$ count $\dot{-} 1$;
$n_{\max} \leftarrow n_{\max} \dot{-} 1$;
go to $L_{i,j}$;

and programs $\beta_{i,l}$, $i < l$, as follows:

$e_{i,l}$: $n_{\max} \leftarrow n_i$;
go to $e_{l,l}$;

Then the complete program to compute maximum $\{t_i\}$ and store it in $n_{\max}$ is:

$$\beta_{1,2}; \cdots; \beta_{l-2,l-1}; \beta_{1,l}; \cdots; \beta_{l-1,l}; e_{l,l}: \text{HALT}.$$

A similar construction will calculate the minimum $\{t_j \dot{-} u_j\}$ and store the result in $n_{\min}$.

To check that $n_{\max}$ is less than $n_{\min}$, the following program suffices:

$x \leftarrow n_{\max}$;
$y \leftarrow n_{\min}$;
L: if $y = 0$ then LOOP;
if $x = 0$ then HALT;
$x \leftarrow x \dot{-} 1$;
$y \leftarrow y \dot{-} 1$;
go to L;

We next need to check congruences. We first set z' to $n_{\max} + 1$, then we check that z' is less than $n_{\min}$, then we repeat the congruence construction of Lemma 5. We change all LOOP nodes from that construction (i.e., failure) to paths leading to a copy of the above construction (i.e., set z' to $z' + 1$, check that z' is less than $n_{\min}$, etc.). We do this M times.

Upon successful termination we test that the atomic formulas not involving z' are satisfied, and upon successful termination of these tests we divide z' by k, using the following program:

$x_{\text{out}} \leftarrow 0$;
L: if $z' = 0$ then HALT;
$x_{\text{out}} \leftarrow x_{\text{out}} + 1$;
$z' \leftarrow z' \dot{-} 1$
$\vdots$ } k repetitions
$z' \leftarrow z' \dot{-} 1$
go to L;

This completes the construction and the resulting program, connected as indicated, computes the function described by Q. □

4. Applications and conclusions. The next natural question to ask is, "Is the result for $L_+ - P_+$ realizability unique?" That is, are there other examples of exact realizability? In one sense, the question is trivial, for we can, given any class of programs and the relations they represent, define a logical theory and an interpretation such that the formulas in that theory are exactly realized (i.e., for each program π, introduce a relation symbol R_π and define R_π to be true exactly when π halts).

For more natural logical theories we can say a bit more. Both the theory of successor and the theory of "less than" [5] can be realized by natural loopless programming languages. The astute reader will have also noticed that we need not have restricted ourselves to the natural numbers and Presberger's arithmetic

in the above proofs (we did this primarily to obtain a comparison with L_1), but we could have been computing over the domain of integers. The appropriate logical theory then would have been that of a commutative group with a discrete total ordering. The proofs presented above remain virtually the same except that $\dot{-}1$ is replaced by -1 (and 0 is no longer an endpoint).

Mention should be made here of similar concepts due to Cook [4] arising out of the study of interpretive semantics for verification theories. Cook defines the notion of assertive in obtaining relative completeness proofs. A logical theory L_1 is *assertive relative to itself, an interpretation and a logic*, L_2, for the programming language entities, if given any precondition from L_1 and any statement from the programming language, there is a strongest post condition from L_1. Taking our programming language logic to be the formula from the theory of the structure $\langle \mathbb{N}, +1, \dot{-}1, 0, = \rangle$, we have that for L_+ programs P_+ is assertive.

At this point an interesting metamathematical question can be formulated. The techniques used to prove exact realizations require an explicit elimination of quantifiers to some closed form. Many decidable theories do not fit this pattern in that they use algebraic and/or model theoretic techniques to prove that their elementary theories are decidable (the theory of finite fields [1] is an example). The question is, are there decidable theories with no exact realizations?

Finally, we can present a complexity result for the difficulty of deciding equivalence of programs in L_+, and hence by reduction, programs in L_1. The proof is based on known results for P_+^*, thus fulfilling one of our earlier goals.

Define $g(m, n)$ by

1. $g(0, n) = n$,
2. $g(m, n) = 2^{g(m-1,n)}$ for $m > 0$.

THEOREM 6. *An upper bound for deciding equivalence of L_+ programs lies between nondeterministic $g(3, p(n))$ time and deterministic $g(4, p(n))$ time, where p is a polynomial and n the sum of the sizes of the L_+ programs.*

Proof. Programs π_1 and π_2 are equivalent iff their representing formulas are equivalent. Let $Q_{\pi_1}(x_1, \cdots, x_n)$ and $Q_{\pi_2}(x_1, \cdots, x_n)$ be the representing formulas (in P_+^*); then the desired equivalence statement is

$$\forall x_1 \cdots \forall x_n [Q_{\pi_1}(x_1, \cdots, x_n) \Leftrightarrow Q_{\pi_2}(x_1, \cdots, x_n)].$$

Letting n be the sum of the length of π_1 and π_2, this formula, by Theorem 1, is of length $O(2^{2^{p(n)}})$. The results of Oppen [10] and Fischer and Rabin [6] then yield the desired bounds. □

Acknowledgment. Appreciation is expressed for the referees' comments, which led to clarifications and improvements in several of the theorems.

REFERENCES

[1] J. AX, *The elementary theory of finite fields*, Ann. of Math., 88 (1968), pp. 239–271.

[2] J. C. CHERNIAVSKY, *Function iteration logics and flowchart schemata*, Computing, 14 (1975), pp. 285–312.

[3] R. L. CONSTABLE AND A. B. BORODIN, *Subrecursive programming languages. Part I: Efficiency and program structure*, J. Assoc. Comput. Mach., 19 (1972), pp. 526–568.

[4] S. A. COOK, *Axiomatic and interpretive semantics for an ALGOL fragment*, Tech. Rep. 79, Dept. of Computer Sci., Univ. of Toronto, 1975.

[5] H. B. ENDERTON, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[6] M. J. FISCHER AND M. O. RABIN, *Super-exponential complexity of Presburger arithmetic*, Complexity of Computation, R. M. Karp, ed., American Mathematical Society, Providence, R.I., 1974, pp. 27–41.
[7] S. C. KLEENE, *Introduction to Metamathematics*, Van Nostrand, Princeton, N.J., 1964.
[8] E. MENDELSON, *Introduction to Mathematical Logic*, Van Nostrand, Princeton, N.J., 1966.
[9] A. R. MEYER AND D. M. RITCHIE, *The complexity of loop programs*, Proc. 22nd Nat. Conf. ACM, Thompson Book Co., Washington, D.C., 1967, pp. 465–469.
[10] D. C. OPPEN, *Elementary bounds for Presberger Arithmetic*, Proc. 5th SIGACT, 1973, pp. 34–37.
[11] D. TSICHRITZIS, *The equivalence problem of simple programs*, Assoc. Comput. Mach., 17 (1970), pp. 729–738.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

ON THE PARALLEL EVALUATION OF BOOLEAN EXPRESSIONS*

AMNON BARAK AND ELIAHU SHAMIR†

Abstract. A bound for the number of steps that are required to evaluate Boolean expressions is obtained. It is shown that any Boolean expression of n distinct variables may be evaluated in $2 \log_2 n - 1$ steps if sufficiently many processors are available.

Key words. parallel algorithms, Boolean expressions

1. Introduction. There has been growing interest in parallel computation in recent years. The importance of the evaluation of arithmetic expressions led to the development of several efficient algorithms for parallel computation.

Brent, Kuck and Maruyama [1] have investigated the parallel evaluation of general arithmetic expressions without division and showed that any such expression can be evaluated in $3 \log_2 n + O(1)$ steps using unlimited parallelism. They further improved their result and reduced $3 \log_2 n$ to $2.465 \log_2 n$ steps. In the conclusions of their paper they noted that one application of their work could be in the area of logic design. For further reading the reader is referred to [1], [2], [3], [4].

In this paper we study the parallel evaluation of Boolean expressions. We assume that sufficiently many independent processors are available and that each processor can perform basic Boolean operations in one unit of time or step. The time required for communication between processors is ignored.

In § 2 we present the notations used in the paper. We also prove a lemma which establishes relations between trees and subtrees of Boolean expressions. In § 3 we derive a bound for the time required to evaluate Boolean expression of $n \leqq 2^k$ distinct elements called *atoms*, connected by the two operators $\wedge$ (AND) and $\vee$ (OR). We prove that such expressions may be evaluated in parallel in at most $2k - 1$ steps.

The operator of negation is also included since it can be pushed down to the atoms level without changing the number of atoms.

Finally, we note that our proof is valid for Boolean expressions (while it is not valid for arithmetic expressions) just because in addition to associativity and commutativity, the operator $\wedge$ distributes over $\vee$ and vice versa. Therefore, our results can be generalized to any pair of operators which satisfy these requirements.

2. Preliminaries. Define a *literal* as a Boolean variable or a constant. An *atom* is a literal, or the negation of a literal. Atoms are denoted by lower case letters. Boolean expressions are denoted by E. The number of atoms in E is denoted by $|E|$. An *atom* is a Boolean expression E, where $|E| = 1$. If E and F are expressions, then $(E \wedge F), (E \vee F)$ are Boolean expressions, where $|E \wedge F| = |E \vee F| = |E| + |F|$. By $E(r)$ we denote an expression with $\leqq r$ atoms (r may be real).

* Received by the editors March 25, 1975.

† Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel.

Two Boolean expressions are equal if one can be obtained from the other by applying the commutative, associative and distributive laws.

An expression E may be conveniently described by a unique binary tree T_E whose nodes are labeled with the operation symbols $\wedge$ and $\vee$ and terminal nodes labeled with atoms. Let g be an atom of E. Let T_i be the sequence of subexpressions corresponding to the subtrees with roots θ_i, $1 \leqq i \leqq r$, where θ are the nodes on the branch leading from g to the root of T_E. Then θ_i joins the expressions T_{i-1} and L_i (cf. Fig. 1), where for simplicity all L_i are drawn to the left.

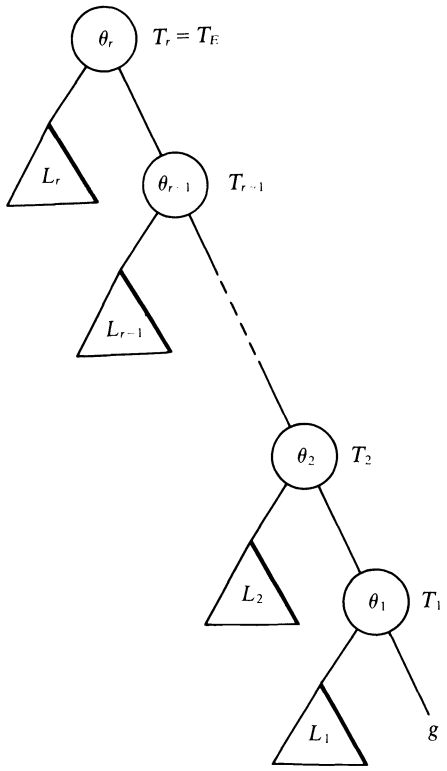


FIG. 1. *Binary tree for* T_E

The next lemma is a Boolean version of Lemma 1 in [1].

LEMMA 1. *Let E be a Boolean expression. Let g be an atom in E. There exists a pair of subexpressions A and B such that*

$$E = (A \wedge g) \vee B,$$

where

$$A = \wedge L_i | \wedge \text{ taken over all } i,\ 1 \leqq i \leqq r, \text{ for which } \theta_i = \wedge,$$

and B corresponds to the subtree obtained from T_E by deleting the subtree T_j, where $j + 1$ is the smallest index for which $\theta_{j+1} = \vee$. (This corresponds to a substitution $g = 0$ in T_E.)

A dual result holds if $\wedge$ and $\vee$ are interchanged:

$$E = (A' \vee g) \wedge B'.$$

3. Monotonic Boolean expressions. In this section we prove the main result of this paper.

THEOREM 1. *Any monotonic Boolean expression* $E(2^{k/2})$, $k \geqq 1$, *can be evaluated in* $k - 1$ *steps if unlimited number of processors are used.*

Proof. Let T_E be any tree for $E(2^{k/2})$. Without loss of generality, we rearrange the tree in such a way that if T_L and T_R are the left and right subtrees which are joined by θ, $\theta \in \{\vee, \wedge\}$, then for all the nodes of T_E

$$|T_L| \leqq |T_R|. \tag{3.1}$$

The proof is by induction on k. By inspection, the theorem is true for $k \leqq 4$. Suppose that for some $k \geqq 5$ any expression $E(2^{k/2})$ may be evaluated in $k - 1$ steps. Now take an expression $E(2^{(k+1)/2})$. Define the rightmost branch of $E(2^{(k+1)/2})$ as in (3.1), with T_i (the expressions of) the subtrees rooted at θ_i and L_i the left arguments joined at θ_i, $1 \leqq i \leqq r$ (cf. Fig. 1). For convenience we denote $2^{k/2} = N$. There is a node θ_p such that

(i) $|T_{p-1}| \leqq N/2$ (hence also $|L_p| \leqq N/2$),

(ii) $|T_p| > (\sqrt{2} - 1)N$.

Without loss of generality, we assume $\theta_p = \wedge$. We note that similar proof holds if $\theta_p = \vee$. We denote the expression of T_{p-1} by g_1 and of L_p by g_2. Then $G = g_2 \wedge g_1$ is the expression of T_p. Let E' be the expression obtained by replacing G by an atom g. Because of (ii), $|T_{E'}| \leqq N$. Using Lemma 1, we obtain

$$E' = (A \wedge g) \vee B,$$

so that

$$E = (A \wedge g_2 \wedge g_1) \vee B,$$

where $|A| \leqq N$, $|B| \leqq N$, and A is a product ($A = \wedge L_i$) of some factor L_i. By the induction hypothesis B can be computed in $k - 1$ steps. It suffices to prove that the product $A \wedge g_2 \wedge g_1$ can also be computed in $k - 1$ steps.

Case 1. Two factors in the product $(\wedge L_i) \wedge g_2 \wedge g_1$ have more than $N \cdot 2^{-3/2}$ atoms (but $\leqq N/2$ atoms). Then the remaining factors together have less than $N/\sqrt{2}$ atoms. The product decomposes into 3 factors which can be computed in $k - 3$, $k - 3$, $k - 2$ steps. Thus $A \wedge g_2 \wedge g_1$ can be computed in $k - 1$ steps.

Case 2. One factor of A, say A_1, has more than $N/2$ atoms (since it is an L_i by (3.1) it must have $\leqq N/\sqrt{2}$ atoms). The remaining factors of A are combined and will be denoted A_2. Since we are not in case 1, two factors in $A_1 \wedge A_2 \wedge g_2 \wedge g_1$ $(= A \wedge G)$ have $\leqq N \cdot 2^{-3/2}$ atoms each, and can be computed in $k - 4$ steps. Thus the four factors are computed in $k - 2$, $k - 4$, $k - 4$, $k - 3$ steps, and so $A \wedge g_2 \wedge g_1$ can be computed in $k - 1$ steps. Since at most one factor of A (and no factor of G) can have more than $N/2$ atoms, we are left with Case 3.

Case 3. At most one of the factors of A or of G has more than $N \cdot 2^{-3/2}$ atoms, all the others (at least 3) have $\leqq N \cdot 2^{-3/2}$ atoms. We can clearly group several of

those to have more than $N \cdot 2^{-3/2}$ atoms, but $\leqq N/\sqrt{2}$ (since the difference between these limits is again $N \cdot 2^{-3/2}$). Hence we have reduced the situation to Case 1 or 2, and the proof is concluded.

Note added in proof. Using dual distributivity F. P. Preparata and D. E. Muller [4] improved our bound to $1.81 \log_2 n$.

REFERENCES

[1] R. BRENT, D. J. KUCK AND K. MARUYAMA, *The parallel evaluation of arithmetic expression without division*, IEEE Trans. Computers, C-22 (1973), pp. 532–534.

[2] R. BRENT, *The parallel evaluation of arithmetic expressions in logarithmic time, complexity of sequential and parallel numerical algorithms*, J. F. Traub, ed., Academic Press, New York, 1973, pp. 83–102.

[3] D. J. KUCK AND K. MARUYAMA, *Time bounds on the parallel evaluation of arithmetic expressions*, this Journal, 4 (1975), pp. 147–162, 1975.

[4] F. P. PREPARATA AND D. E. MULLER, *Efficient parallel evaluation of Boolean expressions*, CSL Tech. Memo., Univ. of Illinois at Urbana-Champaign, Urbana, April 1975.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

A GENERALIZED ASYMPTOTIC UPPER BOUND ON FAST POLYNOMIAL EVALUATION AND INTERPOLATION*

FRANCIS Y. CHIN†

Abstract. It is shown in this paper that the evaluation and interpolation problems corresponding to a set of points, $\{x_i\}_{i=0}^{n-1}$, with $(c_i - 1)$ higher derivatives at each x_i such that $\sum_{i=1}^{n-1} c_i = N$, can be solved in $O([N \log N][(\log n) + 1])$ steps.[1] This upper bound matches perfectly with the known upper bounds of the two extreme cases, which are $O(N \log^2 N)$ and $O(N \log N)$ steps when $n = N$ and $n = 1$, respectively.

Key words. polynomial evaluation, polynomial interpolation, asymptotic upper bounds

1. Introduction. It has been shown by Moenck and Borodin [6] and Kung [5] that the evaluation problem of an $(N - 1)$st-degree polynomial at N points and the interpolation problem at N points by an $(N - 1)$st-degree polynomial can be solved in $O(N \log^2 N)$ steps. Aho, Steiglitz and Ullman [2] and Vari [9] have independently investigated the evaluation problem of an $(N - 1)$st-degree polynomial and all its derivatives at a single point and the interpolation problem by Taylor series of N terms, and they have shown that these two problems can be done in $O(N \log N)$ steps.

In this paper, we extend these results to a set of n points, $\{x_i\}_{i=0}^{n-1}$, with arbitrary numbers of higher derivatives at each point. Let $(c_i - 1)$ be the number of higher derivatives at the corresponding point x_i and $N = \sum_{i=1}^{n-1} c_i$; then we can show that these generalized evaluation and interpolation problems can be done in $O(N \log N(\log n + 1))$ steps, which fits exactly with the known upper bounds of the two extreme cases, $O(N \log^2 N)$ and $O(N \log N)$ steps when $n = N$ and $n = 1$, respectively.

In summary, we have Table 1.

TABLE 1

	Evaluation	Interpolation
(a) N points	$O(N \log^2 N)$[6], [5]	$O(N \log^2 N)$ [6], [5]
(b) n points, $\{x_i\}_{i=0}^{n-1}$ and their corresponding $(c_i - 1)$ derivatives such that $\sum_{i=0}^{n-1} c_i = N$	$O([N \log N][(\log n) + 1])$ (Theorem 1)	$O([N \log N][(\log n) + 1])$ (Theorem 2)
(c) Single point and all its derivatives	$O(N \log N)$ [2], [9]	$O(N \log N)$ [2], [9]

* Received by the editors February 6, 1975, and in revised form October 24, 1975.

† Department of Mathematics, University of Maryland, Baltimore County, Maryland. Now at Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2G1. This research is part of the author's Ph.D. dissertation, Princeton University, Princeton, New Jersey. This work was supported by the National Science Foundation under Grant GK-42048 and by the U.S. Army Research Office—Durham under Contract DAHCO4-69-C0012.

[1] In this paper, the number of arithmetic operations is being considered. In the paper by Shaw and Traub, *On the number of multiplications for the evaluation of a polynomial and some of its derivatives*, appearing in J. Assoc. Comput. Mach., 1974, only the number of multiplications is counted.

2. Polynomial evaluation. Moenck and Borodin [6] and Kung [5] have shown that the problem of evaluating an Nth-degree polynomial is reducible to the problem of dividing the polynomial and proved that a polynomial of degree $N-1$ can be evaluated at N points in $O(N \log^2 N)$ steps. Aho, Steiglitz and Ullman [2] and Vari [9] have independently discovered an $O(N \log N)$ algorithm for evaluating an $(N-1)$st degree polynomial and all its derivatives at a single point. Now we are going to consider the generalized evaluation problem corresponding to different numbers of higher derivatives at an arbitrary set of points. In this paper, we say that $P(x)$ is *computed* if the coefficients t_i in $P(x) = \sum_{i=0}^{n-1} t_i x^i$ are known. (Presumably $P(x)$ was initially represented in a different manner, such as a Taylor expansion about a point other than 0.)

LEMMA 1. *$I_i(x) = (x - x_i)^{c_i}$ can be computed in $O(c_i)$ steps.*

Proof. As

$$I_i(x) = \sum_{j=0}^{c_i} a_{c_i - j} x^j = \sum_{j=0}^{c_i} \binom{c_i}{c_i - j} x_i^{c_i - j} x^j,$$

so

$$a_0 = 1, \qquad a_k = \binom{c_i}{k} x_i^k = \left[\frac{(c_i - k + 1)x_i}{k}\right] a_{k-1}$$

for $k = 1, 2, \cdots, c_i$.

It is obvious that all the a_k's can be found in $O(c_i)$ steps. □

Let $J_{e,f} = \prod_{i=e}^{f} I_i(x)$; then we have

LEMMA 2. *$\{J_{2^i(j-1),2^ij-1} | j = 1, 2, \cdots, 2^{r-i}; i = 1, \cdots, r\}$ can be obtained in $O([N \log N][\log n])$, where $N = \sum_{i=0}^{n-1} c_i$ and n is the number of distinct points.*

Proof. We shall give a constructive proof schematically as in Fig. 1, assuming $n = 2^r$.

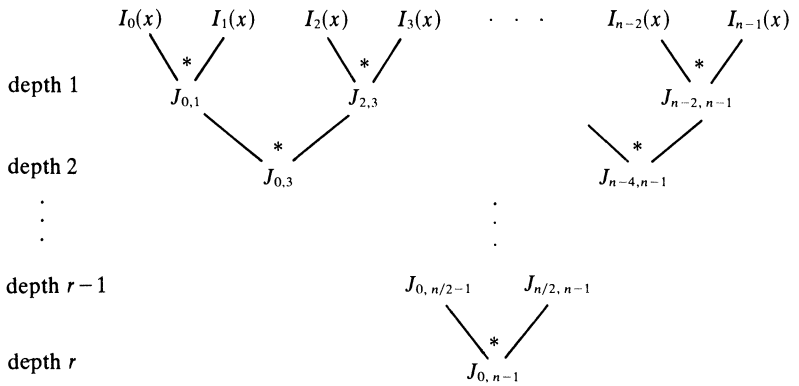


FIG. 1

Let Z_i be the number of steps required to compute all the terms at depth i from the terms computed at depth $i-1$ and $z_{i,j}$ be the degree of the jth terms at depth i.

Since there are 2^{r-i} terms at depth i and multiplication of two kth-degree polynomial takes $O(k \log k)$ steps, so we have $Z_i \leqq \sum_{j=1}^{2^{r-i}} O(z_{ij} \log z_{ij}) \leqq O(N \log N)$, as $\sum_{j=1}^{2^{r-i}} z_{ij} = N$ for all $i = 1, \cdots, r$. Hence it will take no more than $O(rN \log N) = O([N \log N][\log n])$ steps to build the whole tree. □

Define $P(x)$ as a general polynomial of degree $(N-1)$, $P(x) = \sum_{i=0}^{N-1} a_i x^i$; then we have

LEMMA 3. *Let* $P(x) = Q_i(x)I_i(x) + R_i(x)$, *where* $\deg(R_i(x)) < \deg(I_i(x))$ *or* $R_i(x) = P(x) \bmod I_i(x)$. *Then* $\{R_i(x)\}_{i=0}^{n-1}$ *can be computed in* $O([N \log N][\log n])$ *steps.*

Proof. The constructive proof is again illustrated schematically in Fig. 2.

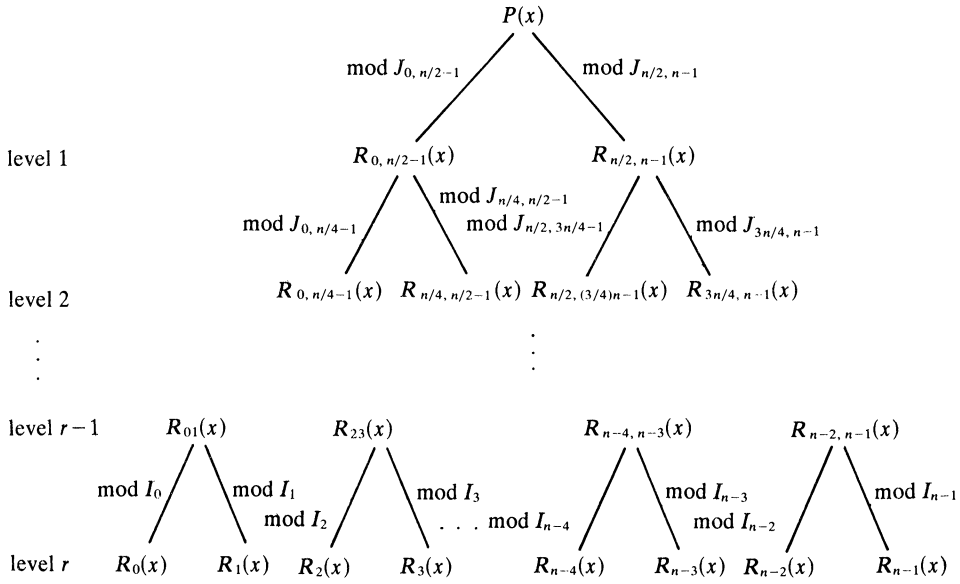


FIG. 2

The argument is analogous to that in Lemma 2, except that polynomial division is used instead of polynomial multiplication. Consider any internal node in the tree, say $R'(x) = \sum_{i=0}^{u-1} r'_i x^i$, and let $R''(x) = \sum_{i=0}^{v-1} r_i x^i$ be a son; then we have

$$R''(x) = R'(x) \bmod J'(x) \quad \text{or} \quad R'(x) = Q(x)J'(x) + R''(x),$$

with $\deg(R''(x)) < \deg(J'(x))$, where $J'(x) = \sum_{i=0}^{v} s_i x^i$; $Q(x) = \sum_{i=0}^{u-v-1} q_i x^i$.

If $v > u/2$, then by assuming that the higher coefficients of $R'(x)$ are zeros, we can treat $R'(x)$ as a polynomial of degree $2v$ and find $R''(x)$ in $O(v \log v) < O(u \log u)$ steps [1, pp. 286–289], [5], [6].

If $v \leqq u/2$, by letting $Q(x)J'(x) = \sum_{i=0}^{u-1} t_i x^i$, it is clear that $r'_i = t_i$ for $i = v, \cdots, u-1$. Therefore we have $M\bar{q} = \bar{r}$, where

$$M = \begin{bmatrix} s_v & s_{v-1} & s_{v-2} & \cdots & s_0 & & & \\ & s_v & s_{v-1} & \cdots & s_1 & s_0 & & 0 \\ & & s_v & & & \ddots & \ddots & \\ & & & \ddots & & & \ddots & s_0 \\ & & & & \ddots & & & s_1 \\ & & 0 & & & \ddots & & \vdots \\ & & & & & & s_v & s_{v-1} \\ & & & & & & & s_v \end{bmatrix},$$

$$\bar{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{u-v-2} \\ q_{u-v-1} \end{bmatrix} \quad \text{and} \quad \bar{r} = \begin{bmatrix} r'_v \\ r'_{v+1} \\ r'_{v+2} \\ \vdots \\ r'_{u-2} \\ r'_{u-1} \end{bmatrix},$$

and by [1, pp. 286–289], [5], [6] again, M^{-1}, which is an upper Toeplitz matrix, can be obtained in at most $O(u \log u)$ steps, and then $Q(x)$ can be obtained with another convolution. Since $R''(x) = R'(x) - Q(x)J'(x)$, $R''(x)$ can be computed with another polynomial multiplication and subtraction which take no more than another $O(u \log u)$ steps. Thus the tree is very similar to the tree in Lemma 2, and the same kind of argument shows that all the terms at each level can be found from the terms computed at the previous level in $O(N \log N)$ steps, and hence it will take no more than $O([N \log N][\log n])$ steps to compute the whole set of polynomials $\{R_i(x)\}_{i=0}^{n-1}$. □

So far, we have shown that it will take no more than $O([N \log N][\log n])$ steps to find all the $R_i(x)$, where

$$P(x) = Q_i(x)I_i(x) + R_i(x),$$

and if we denote the kth derivative of $P(x)$ by $P^{(k)}(x)$, then for $k < c_i$,

$$P^{(k)}(x) = \sum_{j=0}^{k} \binom{k}{j} Q_i^{(j)}(x) I_i^{(k-j)}(x) + R_i^{(k)}(x)$$

and

$$P^{(k)}(x)|_{x=x_i} = R_i^{(k)}(x)|_{x=x_i}.$$

Thus we have the following theorem.

THEOREM 1. *Given an* $(N-1)$*-st-degree polynomial,* $P(x)$*, the problem of evaluating* $\{P^{(j)}(x)|_{x=x_i} | j = 0, \cdots, c_i - 1; i = 0, \cdots, n-1\}$*, where* $N = \sum_{i=0}^{n-1} c_i$*, can be done in* $O([N \log N][(\log n) + 1])$ *steps.*

Proof. By Lemma 3, all the $R_i(x)$ can be obtained in $O([N \log N][\log n])$ steps, and from the above discussion, we know that the problem of evaluating

$P^{(j)}(x)|_{x=x_i}$ is the same as evaluating $R_i^{(j)}(x)|_{x=x_i}$ for all $0 \leqq i < n$ and $j < c_i$. Since $R_i(x)$ is a $(c_i - 1)$st-degree polynomial, the problem of evaluating $R_i(x)$ and all its derivatives at x_i will take $O(c_i \log c_i)$ steps [2], [9]. Hence, with another $\sum_{i=0}^{n-1} O(c_i \log c_i) \leqq O(N \log N)$ steps, we can evaluate $P^{(j)}(x)|_{x=x_i}$ for $j = 0, \cdots, c_i - 1$ and $i = 0, \cdots, n - 1$. Thus total work will take $O([N \log N][(\log n) + 1])$ steps. □

3. Polynomial interpolation. Interpolation of an Nth-degree polynomial by means of the Lagrangian interpolation formula has been studied by Moenck and Borodin [6] and Kung [5], and they have proved that given the function values at N points, the interpolation by an $(N - 1)$st-degree polynomial can be performed in $O(N \log^2 N)$ steps. Aho, Steiglitz and Ullman [2] have also shown that given the values of a polynomial and all its derivatives at a point, the interpolation by Taylor series expansion can be done in $O(N \log N)$ steps. Now we are going to show a more general result on interpolation. Lemma 4 gives the Hermite interpolation formula which appears in a different form in most numerical analysis texts [7].

LEMMA 4. *Given* $\{f_i^{(j)} | j = 0, \cdots, c_i - 1; i = 0, \cdots, n - 1\}$ *and* $\{x_i\}_{i=0}^{n-1}$, *the polynomial* $P(x)$ *can be written as*

$$P(x) = \sum_{i=0}^{n-1} Q_i(x),$$

with

$$Q_i(x) = \frac{L_i(x)}{L_i(x_i)} \sum_{j=0}^{c_i-1} \frac{f_i^{(j)}}{j!} \sum_{k=0}^{c_i-j-1} G_{k,i}(x - x_i)^{j+k},$$

where

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (x - x_j)^{c_j}$$

and $G_{j,i}$ *are defined as follows*:

$$G_{0,i} = 1, \qquad G_{k,i} = -\sum_{m=1}^{k} \frac{1}{m!} G_{k-m,i} \frac{d^m}{dx^m}\left(\frac{L_i(x)}{L_i(x_i)}\right)\Bigg|_{x=x_i},$$

such that $P^{(j)}(x_i) = f_i^{(j)}$ *for* $i = 0, \cdots, n - 1$ *and* $j = 0, \cdots, c_i - 1$.

Proof. Since $N = \sum_{i=0}^{n-1} c_i$, it can be shown easily that the degree of $P(x) = \sum_{i=0}^{n-1} Q_i(x)$ is indeed $N - 1$. Without loss generality, we shall show that $P^{(r)}(x_i) = f_i^{(r)}$ for some i and $r < c_i$. It is obvious that $Q_k^{(r)}(x_i) = 0$ for $k \neq i$, so we have

$$\begin{aligned}
P^{(r)}(x_i) &= \frac{d^r}{dx^r} Q_i(x)\Big|_{x=x_i} \\
&= \frac{d^r}{dx^r}\left(\sum_{j=0}^{c_i-1} \frac{f_i^{(j)}}{j!} \sum_{k=0}^{c_i-j-1} G_{k,i}(x - x_i)^{j+k} \frac{L_i(x)}{L_i(x_i)}\right)\Bigg|_{x=x_i} \\
&= \sum_{j=0}^{c_i-1} \frac{f_i^{(j)}}{j!} \sum_{k=0}^{c_i-j-1} G_{k,i} \sum_{s=0}^{r} \left[\binom{r}{s} \frac{d^s}{dx^s}(x - x_i)^{j+k} \frac{d^{r-s}}{dx^{r-s}}\left(\frac{L_i(x)}{L_i(x_i)}\right)\right]\Bigg|_{x=x_i}
\end{aligned}$$

since

$$\frac{d^s}{dx^s}(x - x_i)^{j+k}|_{x=x_i} \neq 0 \quad \text{iff} \quad j + k = s \leqq r,$$

so we have

$$P^{(r)}(x_i) = \sum_{j=0}^{r} \frac{f_i^{(j)}}{j!} \sum_{k=0}^{r-j} \left[G_{k,i} \binom{r}{j+k} (j+k)! \frac{d^{r-j-k}}{dx^{r-j-k}} \left(\frac{L_i(x)}{L_i(x_i)} \right) \right] \Bigg|_{x=x_i}.$$

By separating the term $j = r$ and letting $m = r - j - k$, we have

$$\begin{aligned} P^{(r)}(x_i) &= f_i^{(r)} + \sum_{j=0}^{r-1} \frac{f_i^{(j)}}{j!} \sum_{m=0}^{r-j} \left[G_{r-j-m,i} \binom{r}{r-m} (r-m)! \frac{d^m}{dx^m} \left(\frac{L_i(x)}{L_i(x_i)} \right) \right] \Bigg|_{x=x_i} \\ &= f_i^{(r)} + \sum_{j=0}^{r-1} \frac{f_i^{(j)}}{j!} r! \left[G_{r-j,i} + \sum_{m=1}^{r-j} G^{r-j-m,i} \quad \frac{1}{m!} \frac{d^m}{dx^m} \left(\frac{L_i(x)}{L_i(x_i)} \right) \right] \Bigg|_{x=x_i} \\ &= f_i^{(r)}. \end{aligned}$$

□

Before getting into the main theorem, we shall prove the following lemmas.

LEMMA 5. *Given* $\{a_i\}_{i=0}^{N-1}$ *and* $\{b_i\}_{i=0}^{N-1}$, *if there exists a* $\{d_i\}_{i=0}^{N-1}$ *which has the property that* $b_j = \sum_{i=0}^{j} d_i a_{j-i}$, *then the sequence* $\{d_i\}_{i=0}^{N-1}$ *can be found in* $O(N \log N)$ *steps.*

Proof. Let

$$A(x) = \sum_{i=0}^{N-1} a_i' x^i, \quad B(x) = \sum_{i=0}^{2N-2} b_i' x^i, \quad D(x) = \sum_{i=0}^{N-1} d_i' x^i,$$

where

$$\begin{aligned} a_i' &= a_{N-i-1} \quad \text{for } i = 0, \cdots, N-1, \\ b_i' &= b_{2N-i-2} \quad \text{for } i = N-1, \cdots, 2N-2, \\ d_i' &= d_{N-i-1} \quad \text{for } i = 0, \cdots, N-1. \end{aligned}$$

By [5], [6], [1, pp. 286–289], given $A(x)$ and $B(x)$, we can find $D(x)$ such that

$$B(x) = A(x)D(x) + R(x), \quad \text{where } \deg(R(x)) < N - 1,$$

in $O(N \log N)$ steps: By equating the coefficients of x^{2N-2-j} in $B(x)$ and $A(x)D(x)$, where $N - 1 \geqq i \geqq 0$, we have

$$\begin{aligned} b_{2N-2-j}' &= \sum_{i=N-1-j}^{N-1} d_i' a_{2N-2-j-i}', \\ b_j &= \sum_{i=0}^{j} d_{N-1-i}' a_{N-1-j+i}' = \sum_{i=0}^{j} d_i a_{j-1}. \end{aligned}$$

In other words, the set $\{d_i\}_{i=0}^{N-1}$ can be found in $O(N \log N)$ steps. □

LEMMA 6. *The set of values,*

$$\left\{ \frac{d^m}{dx^m} \left(\frac{L_i(x)}{L_i(x_i)} \right) \Bigg|_{x=x_i} | m = 0, \cdots, c_i - 1, i = 0, \cdots, n-1 \right\},$$

where $N = \sum_{i=0}^{n-1} c_i$, *can be found in* $O([N \log N][(\log n) + 1])$ *steps.*

Proof. By Lemma 2, $J_{0,n-1}(x)$ can be computed in $O([N \log N][\log n])$ steps. Since

$$L_i(x) = \frac{J_{0,n-1}(x)}{(x - x_i)^{c_i}},$$

$$\frac{d^k J_{0,n-1}(x)}{dx^k} = \sum_{j=0}^{k} \binom{k}{j} \frac{c_i!}{(c_i + j)!} (x - x_i)^{c_i - j} \frac{d^{k-j}}{dx^{k-j}} L_i(x)$$

and

$$\left.\frac{d^{c_i+m} J_{0,n-1}(x)}{dx^{c_i+m}}\right|_{x=x_i} = \binom{c_i + m}{c_i} c_i! \left.\frac{d^m}{dx^m} L_i(x)\right|_{x=x_i},$$

so

$$\left.\frac{d^m}{dx^m} L_i(x)\right|_{x=x_i} = \frac{m!}{(c_i + m)!} \left.\left(\frac{d^{c_i+m}}{dx^{c_i+m}} J_{0,n-1}(x)\right)\right|_{x=x_i}.$$

Thus the problem of finding

$$\left\{ \frac{d^m}{dx^m} L_i(x)|_{x=x_i} \mid m = 0, \cdots, c_i - 1; i = 0, \cdots, n - 1 \right\}$$

reduces to the problem of finding

$$\left\{ \frac{d^m}{dx^m} J_{0,n-1}(x)|_{x=x_i} \mid m = c_i, \cdots, 2c_i - 1; i = 0, \cdots, n - 1 \right\},$$

which has been shown in Theorem 1 to require no more than $O([N \log N][(\log n) + 1])$ steps (Every c_i in Lemmas 1–3 and Theorem 1 is changed to $2c_i$, but it is still the same order of magnitude.) In particular,

$$L_i(x_i) = \frac{1}{c_i!} \frac{d^{c_i}}{dx^{c_i}} J_{0,n-1}(x)|_{x=x_i}$$

and

$$\left.\frac{d^m}{dx^m}\left(\frac{L_i(x)}{L_i(x_i)}\right)\right|_{x=x_i} = \frac{1}{L_i(x_i)} \left.\left(\frac{d^m}{dx^m} L_i(x)\right)\right|_{x=x_i},$$

so with another N divisions, we can compute all the

$$\left.\frac{d^m}{dx^m}\left(\frac{L_i(x)}{L_i(x_i)}\right)\right|_{x=x_i}$$

for $0 \leqq m < c_i$ and $0 \leqq i < n$. □

LEMMA 7. $\{\sum_{j=0}^{c_i-1} f_i^{(j)}/j! \sum_{k=0}^{c_i-j-1} G_{k,i}(x - x_i)^{j+k} | i = 0, 1, \cdots, n - 1\}$ *can be computed in* $O([N \log N][(\log n) + 1])$ *steps.*

Proof. In Lemma 6, we have shown that

$$\left\{ \left.\frac{d^m}{dx^m}\left(\frac{L_i(x)}{L_i(x_i)}\right)\right|_{x=x_i} | m = 0, \cdots, c_i - 1; i = 0, \cdots, n - 1 \right\}$$

can be obtained in $O([N \log N][(\log n) + 1])$ steps. If we let

$$a_k = \frac{1}{k!}\frac{d^k}{dx^k}\left(\frac{L_i(x)}{L_i(x_i)}\right)\Bigg|_{x=x_i} \quad \text{for } k = 0, 1, \cdots, c_i - 1,$$

$$b_k = \begin{cases} 1, & k = 0, \\ 0, & 1 \leqq k \leqq c_i - 1, \end{cases}$$

$$d_k = G_{k,i} \quad \text{for } k = 0, 1, \cdots, c_i - 1,$$

then the problem of obtaining $\{G_{j,i}\}_{j=0}^{c_i-1}$ which is defined by the recursive formula given in Lemma 4, reduces to the problem of finding $\{d_j\}_{j=0}^{c_i-1}$ such that $b_k = \sum_{j=0}^{k} d_j a_{k-j}$. Since a_k and b_k can be computed in $O(c_i)$ steps, by Lemma 5, the d_k's (or $G_{k,i}$) can be found in $O(c_i \log c_i)$ steps. Furthermore, if we let $H_i(x) = \sum_{j=0}^{c_i-1} f_i^{(j)}/j! \sum_{k=0}^{c_i-j-1} G_{k,i}(x - x_i)^{j+k}$, we can write

$$H_i(x) = \sum_{j=0}^{c_i-1} \frac{f_i^{(j)}}{j!} \sum_{k=j}^{c_i-1} G_{k-j,i}(x - x_i)^k = \sum_{k=0}^{c_i-1} \sum_{j=0}^{k} \frac{f_i^{(j)}}{j!} G_{k-j,i}(x - x_i)^k.$$

As the inner summation in the above equation is a convolution, it can be computed in $O(c_i \log c_i)$ steps. If we let $h_k = \sum_{j=0}^{k} f_i^{(j)}/j!\, G_{k-j,i}$ for $k = 0, \cdots, c_i - 1$, then we have

$$H_i = \sum_{k=0}^{c_i-1} h_k(x - x_i)^k.$$

It has also been shown in [2] that the $H_i(x)$'s can be computed in $O(c_i \log c_i)$ steps. Hence total work in finding the set of polynomials $\{H_i(x)\}_{i=0}^{n-1}$ will take no more than $\sum_{i=0}^{n-1} O(c_i \log c_i) + O([N \log N][(\log N) + 1]) \leqq O([N \log N][(\log n) + 1])$ steps. □

After we have established the above results, we can go to the main theorem about the general interpolation problem.

THEOREM 2. *Given* $\{f_i^{(j)} | j = 0, \cdots, c_i - 1 ; i = 0, \cdots, n - 1\}$ *and* $\{x_i\}_{i=0}^{n-1}$, *the interpolation polynomial* $P(x)$ *given in Lemma* 4 *can be computed in* $O([N \log N] \cdot [(\log n) + 1])$ *steps.*

Proof. In Lemma 7, we have shown that $\{H_i(x)\}_{i=0}^{n-1}$ can be obtained in $O([N \log N][(\log n) + 1])$ steps. By using a technique similar to [3], we get

$$\begin{aligned} P(x) &= \sum_{i=0}^{n-1} \frac{L_i(x)}{L_i(x_i)} H_i(x) \\ &= \sum_{i=0}^{n-1} \frac{H_i(x)}{L_i(x_i)} \prod_{\substack{j=0 \\ j \neq i}}^{n-1} I_j(x) \\ &= J_{n/2,n-1} \sum_{i=0}^{n/2-1} H_i'(x) \prod_{\substack{j=0 \\ j \neq i}}^{n/2-1} I_j(x) + J_{0,n/2-1} \sum_{i=n/2}^{n/2-1} H_i'(x) \prod_{\substack{j=n/2 \\ j \neq i}}^{n-1} I_j(x), \end{aligned}$$

where $H_i'(x) = H_i(x)/L_i(x_i)$. Let

$$T_{e,f} = \sum_{i=e}^{f} H_i'(x) \prod_{\substack{j=e \\ j \neq i}}^{f} I_j(x):$$

then we have Fig. 3.

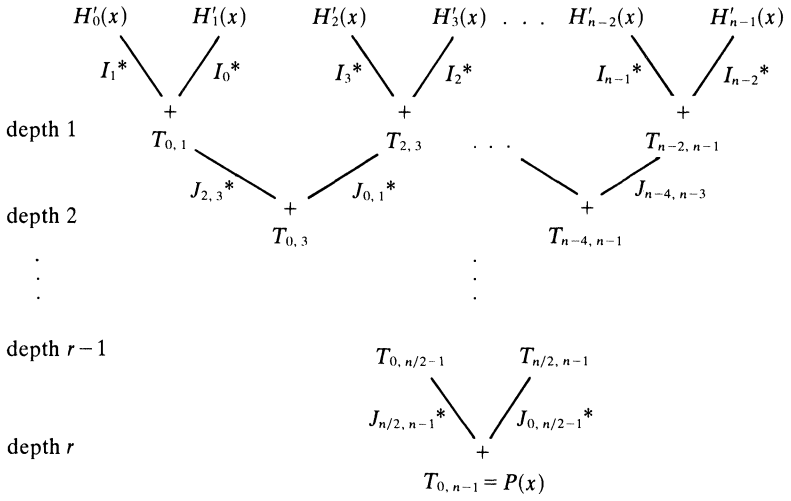


FIG. 3

It can be shown in a way similar to the proof given in Lemma 2 that the number of steps for computing all the terms at each depth is $O(N \log N)$. Since the depth of this binary tree is $r = \log n$, $T_{0,n-1}$ can be obtained in $O([N \log N][\log n])$ steps. Thus total work for obtaining $P(x)$ takes no more than $O([N \log N][(\log n) + 1])$ steps. □

Acknowledgments. The author is grateful to Professor Kenneth Steiglitz for his helpful discussions and guidance. The author also wishes to thank the referees for their comments.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] A. V. AHO, K. STEIGLITZ AND J. D. ULLMAN, *Evaluation polynomials at a fixed set of points*, this Journal, 4 (1975), pp. 533–539.

[3] F. Y. CHIN, *Asymptotic complexity of the inverse partial fraction expansion and Vandermonde matrix*, Tech. Rep. 171, Dept. of Electrical Engrg., Princeton Univ., Princeton, N.J., 1975.

[4] H. T. KUNG, *Fast evaluation and interpolation*, Computer Sci. Tech. Rep., Carnegie-Mellon Univ., Pittsburgh, 1973.

[5] ———, *On computing reciprocals of power series*, Computer Sci. Tech. Rep., Carnegie-Mellon Univ., Pittsburgh, 1973.

[6] R. MOENCK AND A. BORODIN, *Fast modular transforms via division*, Proc. IEEE 13th Ann. Symposium on Switching and Automata theory, Oct. 72, pp. 90–96.

[7] A. C. R. NEWBERY, *Interpolation by algebraic and trigonometric polynomials*, Math. Comp., 20 (1966), pp. 597–599.

[8] M. SIEVEKING, *An algorithm for division of power series*, Computing, 10 (1972), pp. 153–156.

[9] T. M. VARI, *Some complexity results for a class of Toeplitz matrices*, Tech. Rep., Dept. of Computer Sci. and Math., York Univ., Toronto, 1974.

[10] M. SHAW AND J. F. TRAUB, *On the number of multiplications for the evaluation of a polynomial and some of its derivatives*, J. Assoc. Comput. Mach., 21 (1974), no. 1.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

ON THE COMPLEXITY OF TIMETABLE AND MULTICOMMODITY FLOW PROBLEMS*

S. EVEN†, A. ITAI† AND A. SHAMIR‡

Abstract. A very primitive version of Gotlieb's timetable problem is shown to be NP-complete, and therefore all the common timetable problems are NP-complete. A polynomial time algorithm, in case all teachers are binary, is shown. The theorem that a meeting function always exists if all teachers and classes have no time constraints is proved. The multicommodity integral flow problem is shown to be NP-complete even if the number of commodities is two. This is true both in the directed and undirected cases.

1. The timetable problem is NP-complete. The *timetable problem* (TT), which we shall discuss here, is a mathematical model of the problem of scheduling the teaching program of a school. In fact, it is a rather naive model since it ignores several factors which definitely play a role in practice [1]. However, we shall show that even a further restriction of the problem still leads to an NP-complete problem [2], [3].

DEFINITION (TT). Given the following data:

1. a finite set H (of hours in the week);
2. a collection $\{T_1, T_2, \cdots, T_n\}$, where $T_i \subseteq H$; (there are n teachers and T_i is the set of hours during which the ith teacher is available for teaching);
3. a collection $\{C_1, C_2, \cdots, C_m\}$, where $C_j \subseteq H$; (there are m classes and C_j is the set of hours during which the jth class is available for studying);
4. an $n \times m$ matrix R of nonnegative integers; (R_{ij} is the number of hours which the ith teacher is required to teach the jth class).

The problem is to determine whether there exists a meeting function

$$f(i,j,h): \{1, \cdots, n\} \times \{1, \cdots, m\} \times H \to \{0,1\}$$

(where $f(i,j,h) = 1$ if and only if teacher i teaches class j during hour h) such that:

(a) $f(i,j,h) = 1 \Rightarrow h \in T_i \cap C_j$;

(b) $\sum_{h \in H} f(i,j,h) = R_{ij}$ for all $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$;

(c) $\sum_{i=1}^{n} f(i,j,h) \leqq 1$ for all $1 \leqq j \leqq m$ and $h \in H$;

(d) $\sum_{j=1}^{m} f(i,j,h) \leqq 1$ for all $1 \leqq i \leqq n$ and $h \in H$.

(a) assures that a meet takes place only when both the teacher and the class are available. (b) assures that the number of meets during the week between teacher i and class j is the required number R_{ij}. (c) assures that no class has more than one

* Received by the editors February 20, 1975, and in revised form October 29, 1975. Part of this work was done in the Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

† Department of Computer Science, Technion—Israel Institute of Technology, Haifa, Israel.
‡ Department of Computer Science, University of Warwick, Coventry, England

teacher at a time, and (d) assures that no teacher is teaching two classes simultaneously.

A teacher i is called a *k-teacher* if $|T_i| = k$; he is called *tight* if

$$|T_i| = \sum_{j=1}^{m} R_{ij},$$

that is, he must teach whenever he is available.

DEFINITION (RTT). RTT (the restricted timetable problem) is a TT problem with the following restrictions:

1. $|H| = 3$;
2. $C_j = H$ for all $1 \leqq j \leqq m$ (the classes are always available);
3. each teacher is either a tight 2-teacher or a tight 3-teacher;
4. $R_{ij} = 0$ or 1 for every $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$.

Clearly both the TT and the RTT problem are in the NP class. We want to show that RTT is NP-complete. In that case TT is trivially NP-complete too. We recall that 3-SAT (satisfiability of a conjunctive normal form with 3 literals per clause) is NP-complete where 3-SAT is defined as follows: Given the data

1. a set of *literals* $X = \{x_1, x_2, \cdots, x_l, \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_l\}$,
2. a family of *clauses* $D_1, D_2, \cdots, D_k$ such that for every $1 \leqq j \leqq k$, $|D_j| = 3$ and $D_j \subseteq X$,

the problem is to determine whether there exists an assignment of values "true" and "false" to the literals, such that

(a) exactly one of x_i and $\bar{x}_i$ is assigned "true" while the other is assigned "false",

(b) in each clause D_j there is at least one literal assigned "true".

THEOREM 1. 3-SAT $\propto$ RTT.

Proof. The proof is by displaying a polynomially bounded reduction of the 3-SAT to RTT. In our construction, certain classes play the role of occurrences of literals x_i or $\bar{x}_i$ in the clauses; the order in which some 2-teachers teach these classes indicates the truth value of the literals. All other classes and teachers are used in order to guarantee that this assignment of truth values satisfies conditions (a) and (b) above, and that all occurrences of a literal are assigned the same truth value.

Let p_i be the number of times the variable x_i appears in the clauses, i.e.,

$$p_i = \sum_{j=1}^{k} |D_j \cap \{x_i, \bar{x}_i\}|.$$

For each x_i we construct a set of $5 \cdot p_i$ classes which will be denoted by $C_{ab}^{(i)}$, where $1 \leqq a \leqq p_i$ and $1 \leqq b \leqq 5$ (we omit the superscript i whenever all classes used in the construction refer to the same i). In order to simplify the exposition, we shall use a graphic representation of the classes and teachers (see Fig. 1 for the structure corresponding to a single i). In our graphic representation the vertices denote class-hour combinations, where the rows signify the hours and the columns signify the classes. The hours are h_1, h_2 and h_3. Now a 2-teacher who is available during hours h_1 and h_2, and is supposed to meet once with $C_{a_1b_1}$ and once with $C_{a_2b_2}$ will be represented as shown in Fig. 2. The two diagonals show the only two

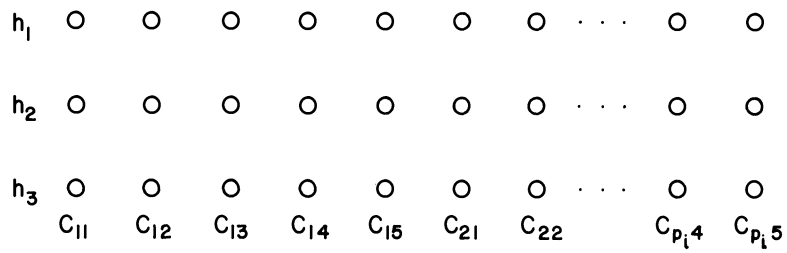


FIG. 1

ways possible to schedule this teacher. A 3-teacher who has to teach $C_{a_1b_1}$, $C_{a_2b_2}$ and $C_{a_3b_3}$ is denoted by a line with three arrows in the columns corresponding to these classes, as shown in Fig. 3. For every $1 \leqq q \leqq p_i$, we add two new classes, C'_{q1} and C''_{q1} with the structure shown in Fig. 4. There are three teachers described in the structure; two are 2-teachers and one 3-teacher. Since all these 3 teachers must teach during h_1, the top 3 vertices, (h_1, C_{q1}), $(h_1 C'_{q1})$ and (h_1, C''_{q1}) must be utilized.

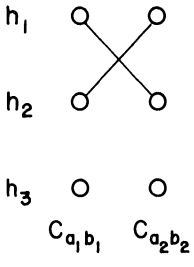


FIG. 2

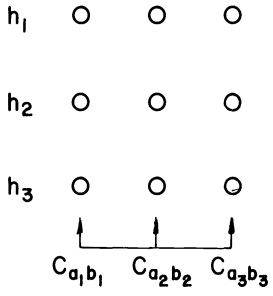


FIG. 3

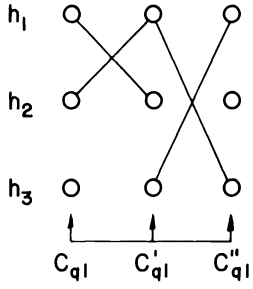


FIG. 4

However, we have a choice of utilizing exactly one of the vertices (h_2, C_{q1}) and (h_3, C_{q1}), while leaving the other available; there are several ways to do this, as the reader may verify by himself. As far as the rest of our structure is concerned, the effect of this substructure is as follows: (h_1, C_{q1}) is taken and one of (h_2, C_{q1}) and (h_3, C_{q1}) is taken. Thus we shall delete (h_1, C_{q1}) from our diagrams.

Consider now the structure of teachers described in Fig. 5; it is intended to consistently assign truth values to all occurrences of x_i and $\bar{x}_i$ in the clauses.

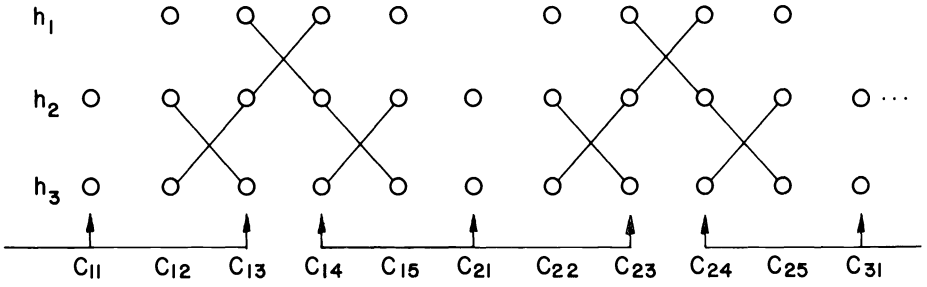


FIG. 5

Clearly, there is a 3-teacher assigned to classes C_{p_i4}, C_{11} and C_{13}; thus the structure described in Fig. 5 is circular. Consider now the p_i 2-teachers who are available during h_1 and h_2, where the qth such teacher is assigned to classes C_{q3} and C_{q4}. We claim that all these teachers must be scheduled in the same manner; that is, either all of them teach the C_{q3} classes during h_1 and the C_{q4} classes during h_2, or all of them teach the C_{q3} classes during h_2 and C_{q4} classes during h_1. Assume we have a schedule which does not satisfy this consistency condition. Then there must be a q such that the qth teacher teaches C_{q3} during h_2 and C_{q4} during h_1, while the $(q+1)$st teacher[1] teaches the $C_{(q+1),3}$ during h_1 and $C_{(q+1),4}$ during h_2. In this case, the 3-teacher who must teach C_{q4}, $C_{(q+1),1}$ and $C_{(q+1),3}$ cannot be scheduled during h_1—a contradiction.

We thus obtain, independently for each i, a uniform scheduling of all the 2-teachers who are available during h_1 and h_2. The order in which these teachers teach C_{q3} and C_{q4} in the ith structure will be interpreted as the truth value of the variable x_i in the original 3-SAT problem.

We now add a few more 3-teachers, connecting the various i-structures, in order to guarantee that in each clause D_j, at least one literal gets the value "true". For every clause $D_j = \{\xi_1, \xi_2, \xi_3\}$, we assign a 3-teacher in the following way. He is assigned to one class for each of the three literals. If $\xi_1 = x_i$ and this is the qth appearance of this variable, then the corresponding class is $C^{(i)}_{q2}$, while if $\xi_1 = \bar{x}_i$, the corresponding class is $C^{(i)}_{q3}$. The classes corresponding to ξ_2 and ξ_3 are defined analogously.

This completes the definition of the RTT problem. The total number of classes defined is $21 \cdot k$, and the total number of teachers is $22 \cdot k$ ($15 \cdot k$ 2-teachers and $7 \cdot k$ 3-teachers). We claim that the given 3-SAT problem has a positive answer if and only if the RTT problem constructed above has a positive answer.

First, assume the 3-SAT problem has a positive answer. We use, now, the values of the literals in such an assignment to display a schedule for the constructed RTT problem—to prove that its answer is positive, too.

If x_i is assigned "true", then for every $1 \leqq q \leqq p_i$, the qth 2-teacher is scheduled to teach $C^{(i)}_{q3}$ during h_1 and to teach $C^{(i)}_{q4}$ during h_2. Conversely, if x_i is assigned "false", then for every $1 \leqq q \leqq p_i$, the qth 2-teacher is scheduled to teach $C^{(i)}_{q3}$ during h_2 and $C^{(i)}_{q4}$ during h_1.

In every clause D_j there is at least one literal assigned "true"; assume it is ξ. If $\xi = x_i$ and this is the qth appearance of this variable, then the 2-teacher who is supposed to teach C_{q2} and C_{q3} is scheduled to teach C_{q2} during h_3 and C_{q3} during h_2.

(In our Fig. 6, the schedule assigned to each of the 2-teachers discussed so far is shown by a heavy solid line, and the choice we avoided is shown by a dashed line. A light solid line indicates that no choice has been made yet.) The 3-teacher of $C_{(q-1),4}$, C_{q1} and C_{q3} uses h_1 to teach $C_{(q-1),4}$, h_2 to teach C_{q1} and h_3 to teach C_{q3}. (His meets are indicated by the circled vertices.) Finally, the 3-teacher corresponding to D_j uses h_2 to teach C_{q2}. It remains to be shown that he can use h_1 and h_3 to teach the other two classes he is assigned to teach. Clearly, h_1 is never occupied

[1] Here $q+1$ should be computed conventionally, except that $p_i + 1 = 1$, to fit the circular structure.

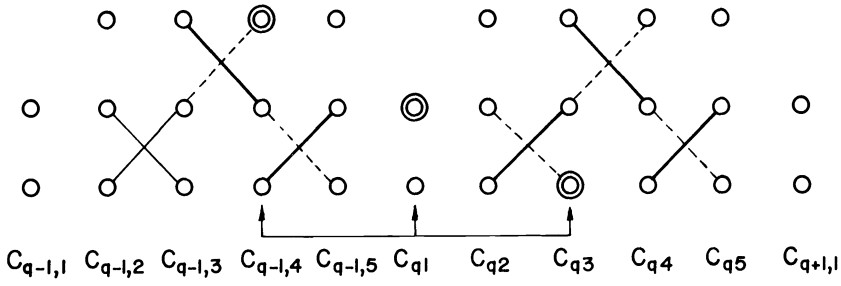


FIG. 6

by any other teacher in classes of types C_{a2} and C_{a5}. If $\xi' = x_r$ is another literal in D_j and it is "false", then the corresponding C_{a2} class must be taught during h_2 by the 2-teacher, and h_3 remains available. Also if $\xi' = \bar{x}_r$ and it is "false", then C_{a5} must be taught during h_2 by the 2-teacher and again h_3 remains available. Finally, if both remaining literals in D_j are "true", then for one of them, we do not follow the scheme used for ξ. For example, if $\xi' = x_r$, it is "true" and this is the ath appearance of this variable, then the 2-teacher teaches C_{a2} during h_2 and C_{a3} during h_3. The 3-teacher teaches $C_{(a-1),4}$ during h_1, C_{a1} during h_3 and C_{a3} during h_2 (as shown in Fig. 7). Thus h_3 remains available to teach C_{a2}, and the scheduling of the 3-teacher corresponding to D_j is now easy. The other cases are similar, and the reader may check them out for himself.

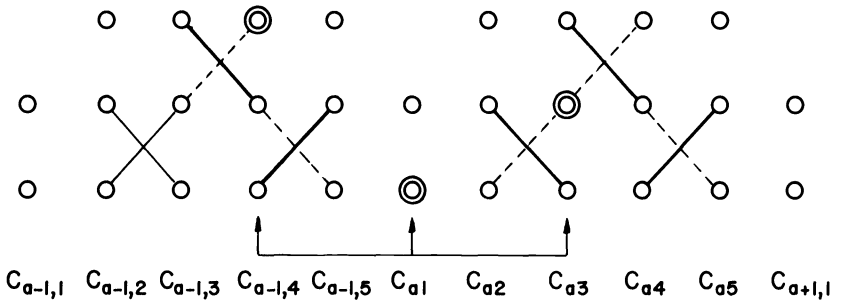


FIG. 7

Second, assume the answer to the constructed RTT problem is positive, and assume we have a legal scheduling. If in the structure of x_i the 2-teachers assigned to teach C_{q3} and C_{q4} teach C_{q3} during h_1 and C_{q4} during h_2, then x_i is given the value "true", and if they teach C_{q3} during h_2 and C_{q4} during h_1, then x_i is given the value "false". It remains to be shown that each clause $D_j = \{\xi_1, \xi_2, \xi_3\}$ contains at least one literal which is "true". If $\xi \in D_j$ and it is "false", then h_2 is used for teaching the corresponding class (a C_{a2} if $\xi = x_i$, and a C_{a5} if $\xi = \bar{x}_i$) by the 2-teacher that teaches it and the adjacent class (C_{a3} if $\xi = x_i$ and C_{a4} if $\xi = \bar{x}_i$). Thus, if all three literals are "false", the 3-teacher corresponding to D_j cannot have an assignment to teach its three classes, since he cannot use h_2. Q.E.D.

2. The timetable problem with binary teachers is polynomially solvable. Consider the TT problem with the restriction that all teachers are 2-teachers. (A 1-teacher is of no interest.) We shall show that a simple branching procedure solves the problem in polynomial time, since the branching depth is limited.

Our algorithm will determine schedules for the teachers progressively. At a given stage, when part of the teachers have been scheduled we say that a teacher is *impossible* if he cannot be scheduled consistently; we say that he is *implied* if there is only one possible way to schedule him consistently with the schedules established so far.

ALGORITHM.

1. Set PHASE to 2.
2. If all teachers have been scheduled, halt with a positive answer.
3. If there is an unscheduled teacher who is impossible, go to step 7.
4. If there is no unscheduled implied teacher, go to step 6.
5. Let T_i be an unscheduled implied teacher. Temporarily schedule T_i as necessary and go to step 2.
6. Make all temporary schedules permanent. Let T_i be any unscheduled teacher. Arbitrarily choose a schedule for him and record this decision. Set PHASE to 1 and go to step 2.
7. If PHASE = 2, halt with a negative answer.
8. Reverse the schedule of the recorded teacher and undo all the temporary schedules. Set PHASE to 2 and go to step 3.

This algorithm clearly returns a positive answer only if a possible meeting function is constructed. It uses a limited backtracking since only one decision is ever recorded and possibly changed. It is less obvious that this limited backtracking is sufficient to discover a meeting function, if one exists.

Let a component of the evaluation be a set of teachers whose schedules gained permanency simultaneously (in step 6). The components may depend on arbitrary choices and on the order in which the teachers are considered. They are numbered consecutively according to their order of occurrence. For completeness, the set of teachers who are not scheduled or whose schedule had not been made permanent at the time the algorithm terminated is considered the last component.

LEMMA 1. *If T_i is a teacher of the last component, then none of the class-hours he may use is occupied by a teacher of a previous component.*

Proof. New components are started by entering step 6; but this occurs only when no teacher is implied. Since all teachers are binary, the lemma follows. Q.E.D.

The lemma implies that whenever the algorithm terminates with a negative answer, after trying both possible schedules for a certain teacher and all the schedules implied by it and failing, we can be sure that all the permanent schedules made before could not have hindered the situation, and thus the negative answer is conclusive.

It is worth noting here, that the technique of limited branching is applicable in other similar situations, such as the 2-SAT problem (i.e., the satisfiability problem for conjunctive normal forms with at most two literals per clause). Using appropriate data structures in order to find the implications of any decision made, and trying both decisions in step 6 in parallel (so that the quicker success stops the evaluation of the other possibility), it can be shown that the algorithm has

time complexity $O(n)$. Other known algorithms for the 2-SAT problem, such as the Davis and Putham [4] algorithm (pointed out by Cook [2]) or an algorithm which follows from Quine's work [5] on the concensus (star) operation, have time complexity $O(n^2)$.

3. There is always a meeting function if all teachers and classes have no time constraints. The purpose of this section is to document a theorem which follows from the classical theory of matching in bipartite graphs [6].

We say that a given TT problem has no *time constraints* if for all $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$ $T_i = C_j = H$; we say that it is *apparently feasible* if neither the teachers nor the classes are overloaded, i.e.:

$$\text{(i)} \quad \text{for all } 1 \leqq i \leqq n, \quad \sum_{j=1}^{m} R_{ij} \leqq |H|,$$

$$\text{(ii)} \quad \text{for all } 1 \leqq j \leqq m, \quad \sum_{i=1}^{n} R_{ij} \leqq |H|.$$

Clearly the condition that a TT problem be apparently feasible is necessary for the existence of a meeting function, but is not sufficient.

Our purpose is to prove the following theorem:

THEOREM 2. *If a* TT *problem is apparently feasible and has no time constraints, then it has a meeting function.*

Proof. First let us define the following quantities:

$$r = \sum_{i=1}^{n} \sum_{j=1}^{m} R_{ij}, \qquad h = |H|,$$

$$\nu = m - \left\lfloor \frac{r}{h} \right\rfloor, \qquad \mu = n - \left\lfloor \frac{r}{h} \right\rfloor.$$

Now, define a bipartite multi-graph $G(X, Y, E)$ in the following way:

$$X = \{x_1, x_2, \cdots, x_n\} \cup \{\xi_1, \xi_2, \cdots, \xi_\nu\},$$

$$Y = \{y_1, y_2, \cdots, y_m\} \cup \{\eta_1, \eta_2, \cdots, \eta_\mu\}.$$

E is a set of edges connecting between vertices of X and vertices of Y constructed as follows. For every $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$, we put R_{ij} parallel edges between x_i and y_j. Next, for each $1 \leqq i \leqq n$, we complete the degree[2] of x_i to be exactly h by putting $h - \sum_{j=1}^{m} R_{ij}$ edges between x_i and vertices of $\{\eta_1, \eta_2, \cdots, \eta_\mu\}$; it does not matter to which of these vertices these edges are connected provided the degree of each η_k never exceeds h. Also, for each $1 \leqq j \leqq m$, we complete the degree of y_j to be exactly h by putting $h - \sum_{i=1}^{n} R_{ij}$ edges between y_j and vertices of $\{\xi_1, \xi_2, \cdots, \xi_\nu\}$, again taking care that the degree of each ξ_l never exceeds h. Finally, we complete the degree of the vertices in $\{\xi_1, \xi_2, \cdots, \xi_\nu\}$ and $\{\eta_1, \eta_2, \cdots, \eta_\mu\}$ to be exactly h, too, by putting edges from any ξ_l to any η_k which both have a lower degree.

It remains to show that this definition is proper in the sense that all the conditions it implies are easily met.

[2] The degree of a vertex is the number of edges incident to it.

The number of edges we construct in the completion of the degrees of $x_1, x_2, \cdots, x_n$ is $n \cdot h - r$. Thus we can do this if $\mu \cdot h \geqq n \cdot h - r$, and μ satisfies this inequality. Similarly, ν satisfies the condition for the possibility of the completion of the degrees of $y_1, y_2, \cdots, y_m$. Finally, the number of edges required to complete the degrees of $\xi_1, \xi_2, \cdots, \xi_\nu$ is $\nu \cdot h - (m \cdot h - r)$, which is equal to $r - \lfloor r/h \rfloor \cdot h$. (This is the remainder of r upon division by h.) Similarly, the number of edges required to complete the degrees of $\{\eta_1, \eta_2, \cdots, \eta_\mu\}$ is the same. Thus, the last part of the construction raises no difficulties, either.

Next, let $\Gamma(A)$, where $A \subseteq X$, be the set of vertices $B \subseteq Y$ such that there is an edge $a \to b \in E$, where $a \in A$ and $b \in B$.

LEMMA 2. *For every* $A \subseteq X$, $|\Gamma(A)| \geqq |A|$.

Proof. There are $h \cdot |\Gamma(A)|$ edges incident to $\Gamma(A)$ in G. This includes all the edges which are incident to A. Thus

$$h \cdot |\Gamma(A)| \geqq h \cdot |A|. \qquad \text{Q.E.D.}$$

Lemma 2 assures that Hall's condition holds, and thus, by Hall's theorem [6], there is a set of $n + \nu (= m + \mu)$ edges, no two of which have a common endpoint. We now use this set of edges M (which is commonly called a complete match of X to Y) to define the meeting function for the first hour $h_1 \in H$; if $x_i \to y_j \in M$, then $f(i, j, h_1) = 1$; otherwise $f(i, j, h_1) = 0$. Clearly conditions (c) and (d) hold for h_1. Next we remove M from E. The new graph has degree $h - 1$ for all its vertices, and as in Lemma 2, Hall's condition holds again. This assures the existence of another complete match M' of X to Y, and we can use it to define $f(i, j, h_2)$ for all $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$. We repeat this until, by the hth application, all E's edges have been used. This assures that condition (b) holds. Thus the proof of Theorem 2 is complete. Q.E.D.

The technique used here is an easy generalization of the one classically used to prove the school dance theorem. (See, for example, [7, Example 2, p. 92].) Since the proof is constructive and a complete match of X to Y can be obtained in polynomial time (Hopcroft and Karp [8]), this technique can be used in order to find an appropriate scheduling in polynomial time rather than just proving its existence.

4. The two-commodity integral flow problem is NP-complete. Knuth (see [9]) has shown that the multicommodity integral flow problem is NP-complete. His reduction, from the satisfiability problem, uses as many commodities as there are clauses.

We present a reduction of the satisfiability problem to the two-commodity integral flow in directed graphs (D2CIF), and in turn, a reduction of the D2CIF problem to the U2CIF (the undirected version).

DEFINITION (D2CIF). Given the following data:

1. $G(V, E)$ a directed finite graph; a directed edge from u to v is denoted $u \to v$;
2. a capacity function $c : E \to N$, where N is the set of nonnegative integers;
3. vertices s_1 and s_2 (not necessarily distinct) which are called the *sources*;
4. vertices t_1 and t_2 (not necessarily distinct) which are called the *terminals*;
5. two nonnegative integers R_1 and R_2 which are called the *requirements*.

The problem is to determine whether there exist two flow functions f_1 and f_2, both $E \to N$, such that

(a) for every $u \to v \in E$, $f_1(u \to v) \geqq 0$, $f_2(u \to v) \geqq 0$ and $f_1(u \to v) + f_2(u \to v) \leqq c(u \to v)$;
intuitively, the commodities flow along the directed edge $u \to v$ from u to v. The total flow along an edge is bounded from above by the capacity of the edge;

(b) for each commodity $i \in \{1, 2\}$ and each vertex $v \in V - \{s_i, t_i\}$

$$\sum_{u \to v \in E} f_i(u \to v) = \sum_{v \to w \in E} f_i(v \to w);$$

this is the conservation rule which states that for each commodity the amount of flow which enters a vertex equals the flow which emanates from it;

(c) for each commodity $i \in \{1, 2\}$, let the total flow be

$$F_i = \sum_{s_i \to v \in E} f_i(s_i \to v) - \sum_{v \to s_i \in E} f_i(v \to s_i);$$

then it is required that

$$F_i \geqq R_i.$$

A flow problem is *simple* if the capacities of all edges are equal to one.

THEOREM 3. *Simple* D2CIF *is* NP-*complete.*

Proof. It suffices to show: satisfiability $\propto$ simple D2CIF.

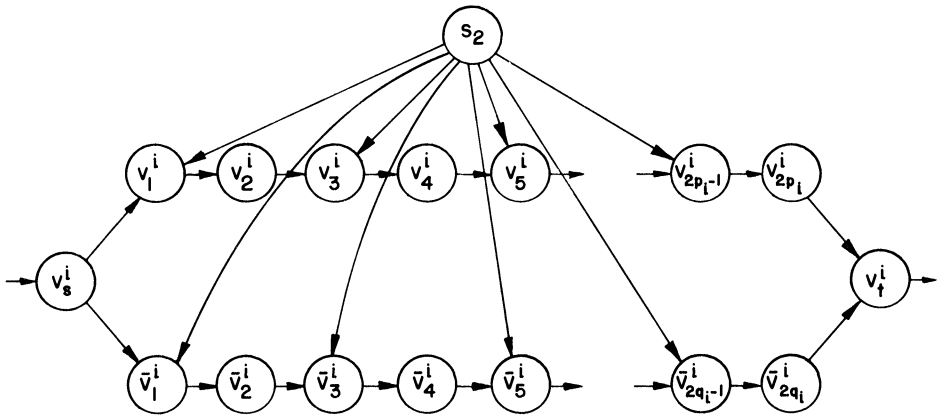


FIG. 8

Let the clauses of the satisfiability problem be $D_1, \cdots, D_k$ and $x_1, \cdots, x_l$, $\bar{x}_1, \cdots, \bar{x}_l$ be the literals. For each variable x_i, we construct a lobe as shown in Fig. 8. (Here p_i is the number of occurrences of x_i in the clauses and q_i is the number of occurrences of $\bar{x}_i$.) The capacity of all edges is 1. The lobes are connected to one another in series: v_t^i is connected to v_s^{i+1}, s_1 is connected to v_s^1 and v_t^l to t_1. s_2 is connected to all the vertices v_j^i and $\bar{v}_j^i$ where j is odd. In addition, there are

vertices $D_1, D_2, \cdots, D_k$ and an edge from each to t_2. For the jth occurrence of x_i ($\bar{x}_i$), there is an edge from v^i_{2j} ($\bar{v}^i_{2j}$) to the D_r in which it occurs. The requirements are $R_1 = 1$ and $R_2 = k$.

(a) Assume that there exist flow functions f_1 and f_2 which satisfy the requirements. Clearly, $F_1 = 1$ and $F_2 = k$. The unit of the first commodity flow must pass through all lobes. Define x_i to be "true" if and only if the first commodity flow passes through the lower path of the ith lobe. In this case, flow of the second commodity may pass through the upper part of the lobe to all the clauses which contain x_i. Since $F_2 = k$, through each vertex D_j there is a unit flow of the second commodity. Assume that this unit of flow comes from the ith lobe. If it comes from the upper part of the lobe, then $x_i \in D_j$ and the first commodity must flow through the lower part of the lobe. Thus x_i is "true" and D_j is satisfied.

If the flow comes from the lower part of the lobe, a similar argument holds. This completes the proof that the expression is satisfiable.

(b) If the expression is satisfiable, we send the first commodity flow through the lower path of the ith lobe if and only if x_i is "true". Since each clause D_j contains at least one literal x_i or $\bar{x}_i$ which is "true", the second commodity passes through the upper or lower path depending on whether x_i or $\bar{x}_i$ is "true".

Thus both requirements are met. Q.E.D.

Next, we show that U2CIF [10] is NP-complete, too. The definition of U2CIF is similar to that of D2CIF except that the graph is undirected. Denoting an undirected edge between u and v as $u - v$, its capacity is $c(u - v)$. However, the flow has a direction. If the flow is from u to v $f_i(u - v)$ is positive and $f_i(v - u)$ is its negation. (Note that $c(u - v) = c(v - u) \geqq 0$.) Condition (a) changes into

$$|f_1(u - v)| + |f_2(u - v)| \leqq c(u - v), \qquad u - v \in E,$$

implying that the total flow in both directions is less than the capacity.

As before, condition (b) assures that for each $v \in V - \{s_i, t_i\}$ the total flow of commodity i entering v is equal to the total flow of commodity i emanating from v, i.e.,

$$\sum_{u-v \in E} f_i(u - v) = 0.$$

Let the total ith commodity flow be $F_i = \sum_{s_i - u \in E} f_i(s_i - u)$. Condition (c) states that $F_i \geqq R_i$.

THEOREM 4. *Simple* U2CIF *is* NP-*complete.*

Proof. It suffices to show:

$$\text{simple D2CIF} \propto \text{simple U2CIF}.$$

First we change the directed graph $G(V, E)$ as follows: we add four new vertices $\bar{s}_1, \bar{s}_2, \bar{t}_1$ and $\bar{t}_2$ to serve as the two new sources and terminals, respectively. We connect $\bar{s}_1$ to s_1 via R_1 parallel edges and t_1 to $\bar{t}_1$ via R_1 parallel edges. Similarly, $\bar{s}_2$ is connected to s_2 and t_2 to $\bar{t}_2$ via R_2 parallel edges in each case. Vertices s_1, s_2, t_1 and t_2 are now subject to the conservation rule and the requirements are the same. Clearly, the requirements can be met in the new graph $G'(E', V')$ if and only if they can be met in the original one. Also, without loss of generality, we may assume that $R_1 + R_2 \leqq |E|$, or obviously the requirements cannot be met.

Thus these changes can only expand the data describing the problem linearly. Now we proceed to construct the undirected network from the new directed network.

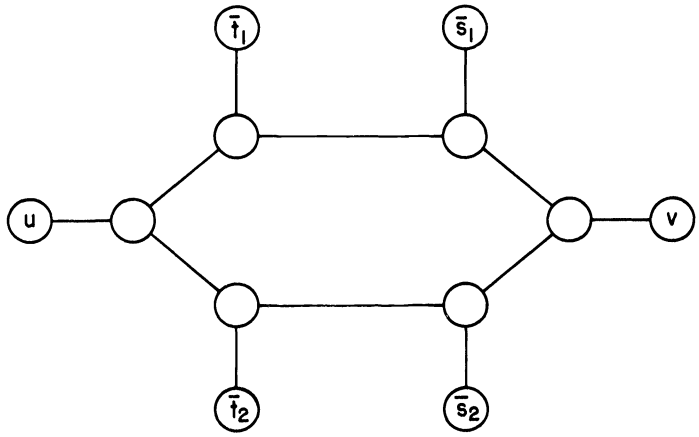


FIG. 9

Each edge $u \to v$ of G' is replaced by the construct shown in Fig. 9. (u or v may be one of the sources or terminals.) Only the unlabeled vertices of the construct are new and do not appear elsewhere in the graph.

It remains to be shown that the requirements can be met in the directed network if and only if the requirements $R_1' = R_1 + |E'|$ and $R_2' = R_2 + |E'|$ can be met in the undirected network.

First assume that the requirements of the directed network can be met. Initially, flow one unit of each commodity through each one of the edge-constructs, as shown in Fig. 10. This yields $F_1 = |E'|$ and $F_2 = |E'|$. Next, if $u \to v$ is used in G' to flow one unit of the first commodity, then we change the flows in the edge-construct as shown in Fig. 11. The case of the second commodity flowing through

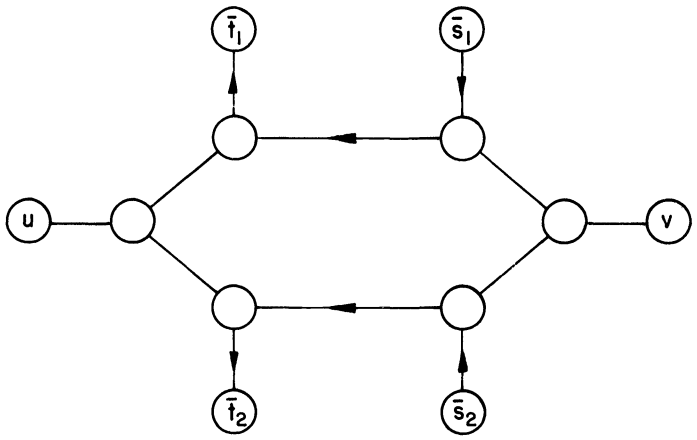


FIG. 10

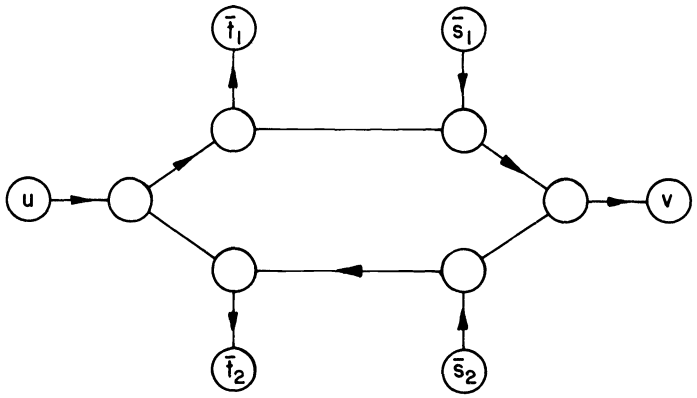


FIG. 11

$u \to v$ in G' is handled similarly. It is easy to see that R'_1 and R'_2 are now met in the undirected graph.

Now assume we have a flow in the undirected graph satisfying the requirements R'_1 and R'_2. Since the number of edges incident to $\bar{s}_i$ ($\bar{t}_i$) is R'_i, all these edges are used to emanate (inject) ith commodity flow from (into) $\bar{s}_i$ ($\bar{t}_i$). The flow through each edge-construct must therefore be in one of the following patterns:

1. as in Fig. 10;
2. as in Fig. 11;
3. as in Fig. 11, for the second commodity.

We can now use the following flow through $u \to v$ in G': If the $u \to v$ construct is of pattern 1, then $f_1(u \to v) = f_2(u \to v) = 0$. If it is of pattern 2, then $f_1(u \to v) = 1$ and $f_2(u \to v) = 0$, etc. Clearly this defines a legal flow for G' which meets the requirements. Q.E.D.

It is easy to see that the multicommodity integral flow problems, as we have defined them, are easily reducible to the version in which we have only one (total) requirement, i.e., $F_1 + F_2 \geqq R$. Thus, the latter versions are NP-complete, too. Also, the completeness of the above problems imply the completeness of the two-commodity integral flow problems with arbitrary capacities for both the directed and the undirected case. Also, the completeness of $m \geqq 2$-commodity integral flow problems follows.

REFERENCES

[1] C. C. GOTLIEB, *The construction of class-teacher time-tables*, Proc. IFIP Congr. 62, 1963, pp. 73–77.

[2] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. of the 3rd Ann. ACM Symp. on Theory of Computing, 1971, pp. 151–158.

[3] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computation, R. N. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[4] M. DAVIS AND H. PUTNAM, *A computing procedure for quantification theory*, J. Assoc. Comput. Mach., (1960), pp. 201–215.

[5] W. V. QUINE, *On cores and prime implicants of truth functions*, Amer. Math. Monthly, 66 (1959), pp. 755–760.

[6] P. HALL, *On representations of subsets*, J. London Math. Soc., 10 (1935), pp. 26–30.
[7] C. BERGE, *The Theory of Graphs and its Applications*, John Wiley, New York, 1962.
[8] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, this Journal, 2 (1973), pp. 225–231.
[9] R. M. KARP, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.
[10] T. C. HU, *Multi-Commodity network flows*, J. Operations Res. Soc. Amer., 11 (1963), pp. 344–360.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

THE PLANAR HAMILTONIAN CIRCUIT PROBLEM IS NP-COMPLETE*

M. R. GAREY†, D. S. JOHNSON† AND R. ENDRE TARJAN‡

Abstract. We consider the problem of determining whether a planar, cubic, triply-connected graph G has a Hamiltonian circuit. We show that this problem is NP-complete. Hence the Hamiltonian circuit problem for this class of graphs, or any larger class containing all such graphs, is probably computationally intractable.

Key words. algorithms, computational complexity, graph theory, Hamiltonian circuit, NP-completeness

1. Introduction. A *Hamiltonian circuit* in a graph[1] is a path which passes through every vertex exactly once and returns to its starting point. Many attempts have been made to characterize the graphs which contain Hamiltonian circuits (see [2, Chap. 10] for a survey). While providing characterizations in various special cases, none of these results has led to an efficient algorithm for identifying such graphs in general. In fact, recent results [5] showing this problem to be "NP-complete" indicate that no simple, computationally-oriented characterization is possible. For this reason, attention has shifted to special cases with more restricted structure for which such a characterization may still be possible. One special case of particular interest is that of planar graphs. In 1880 Tait made a famous conjecture [8] that every cubic, triply-connected, planar graph contains a Hamiltonian circuit. Though this conjecture received considerable attention (if true it would have resolved the "four color conjecture"), it was not until 1946 that Tutte constructed the first counterexample [9]. We shall show that, not only do these highly-restricted planar graphs occasionally fail to contain a Hamiltonian circuit, but it is probably impossible to give an efficient algorithm which distinguishes those that do from those that do not.

2. Proof of result. Our proof of this result is based on the recently developed theory of "NP-complete problems". This class of problems possesses the following important properties:

(A) There is no known polynomial-time algorithm that solves any single problem in the class.

(B) The existence of a polynomial-time algorithm for solving any *particular* problem in the class would imply that *every* problem in the class can be solved with a polynomial-time algorithm.

It is widely believed that no NP-complete problem can be solved with a polynomial-time algorithm and hence that all such problems are inherently

* Received by the editors October 23, 1975.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by a Miller Research Fellowship at the University of California, Berkeley, National Science Foundation Grant GJ-36473X at Stanford University and Bell Laboratories.

[1] See [2] for any undefined terminology in graph theory.

computationally intractable. Formal introductions to the notion of NP-completeness can be found in [1], [5], [6]. Karp [5] first demonstrated that many well-known combinatorial problems were NP-complete. Others have added to a long and growing list of such problems (see [6] for a recent survey).

In [5] a construction due to Lawler was presented which showed that the Hamiltonian circuit problem for arbitrary graphs is NP-complete. Garey, Johnson and Stockmeyer [4] proved that the Hamiltonian line problem for directed planar graphs is NP-complete. (A Hamiltonian line in a directed graph is a directed path which passes through each vertex exactly once, but need not return to its starting point.) We shall show that the Hamiltonian circuit problem is NP-complete even for graphs G satisfying

(i) G is planar,
(ii) G is cubic (each vertex has degree 3),
(iii) G is triply-connected (deletion of any two vertices leaves the graph connected).

Thus the Hamiltonian circuit problem for these highly restricted graphs seems to be essentially as difficult as that for arbitrary graphs.

The formal technical requirements for a proof of NP-completeness are adequately described in [1, Chap. 10], [5], [6]. For our purposes, the only nontrivial requirement is that we show how a known NP-complete problem can be "transformed" in polynomial time into this restricted Hamiltonian circuit problem. This "known" NP-complete problem will be the satisfiability problem of propositional calculus [3], [5].

Let F be any well-formed formula containing atomic variables and the connectives $\wedge$ (and), $\vee$ (or) and $-$ (not). F is *satisfiable* if there exists some assignment of the values *true* and *false* to the variables which makes F *true* under the standard interpretation of the connectives. We shall show how to construct, in polynomial time, a graph G satisfying (i)–(iii) such that F is satisfiable if and only if G contains a Hamiltonian circuit. By results in [3], [5], it suffices to consider only formulas F in conjunctive normal form with three literals per clause. That is, we may assume that F has the form

$$(p_{11}\vee p_{12}\vee p_{13})\wedge(p_{21}\vee p_{22}\vee p_{23})\wedge\cdots\wedge(p_{m1}\vee p_{m2}\vee p_{m3}),$$

where each $(p_{i1}\vee p_{i2}\vee p_{i3})$ is called a *clause* and each p_{ij}, called a *literal*, is either an atomic variable or the negation of an atomic variable. We assume that F contains n atomic variables, denoted $x_1, x_2, \cdots, x_n$.

A number of special graph configurations will be used in our construction and are illustrated in Figs. 1–7. Consider the graph, due to Tutte [9], shown in Fig. 1(a). Any Hamiltonian circuit in a graph G that contains this graph as a vertex-induced subgraph must appear locally as one of the states shown in Fig. 1(b) and thus must use the edge marked A. That is, this subgraph acts like a single degree-3 vertex which has one "specified" edge that is required to be used in any Hamiltonian circuit of G.

We use the graph in Fig. 1 to construct the "exclusive-or" graph shown in Fig. 2(a). Any Hamiltonian circuit in a graph G which contains this graph as a vertex-induced subgraph must appear locally in one of the two states shown in Fig. 2(b). Thus this subgraph acts like two separate edges, one joining v to v' and the

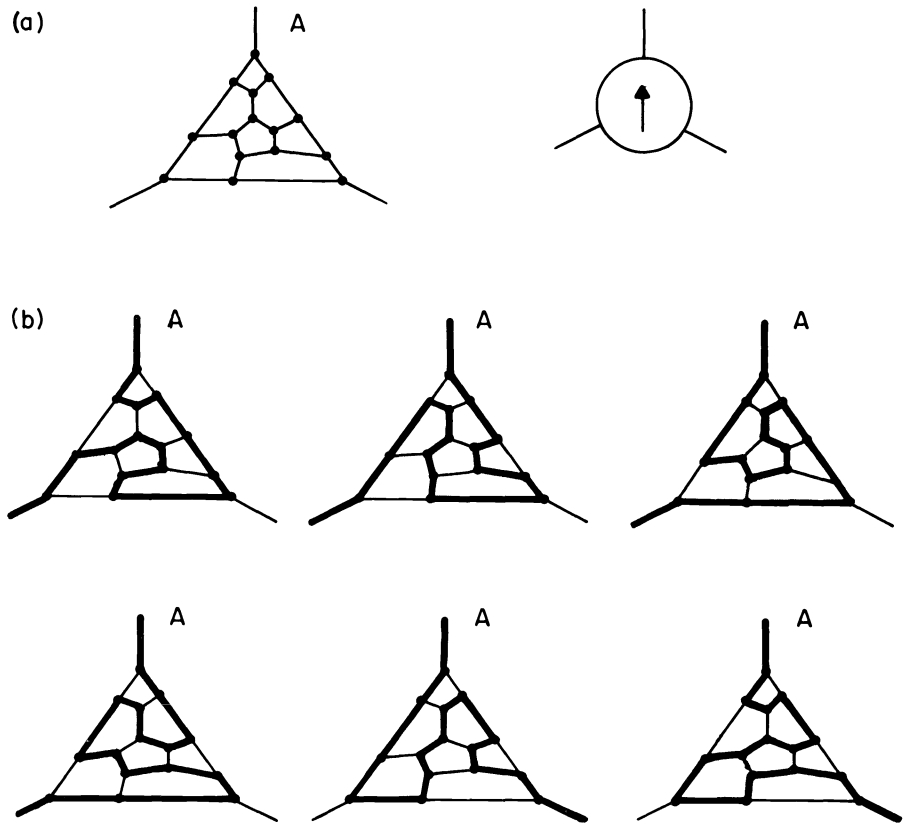


FIG. 1. *Required-edge graph*
(a) *Graph and abbreviation*
(b) *Possible local states*

other joining u to u', with the constraint that exactly one of these two edges must occur in any Hamiltonian circuit of G. In this case, we say that the edges $\{u, u'\}$ and $\{v, v'\}$ have been "joined" by an exclusive-or. Schematically, this will be represented by the abbreviation shown in Fig. 2(a), which we shall call an "exclusive-or line".

The exclusive-or construction is crucial to the planarity of the graph G which will correspond to the formula F. The key observation is that two "exclusive-or lines" joining different pairs of edges may cross each other without destroying the planarity of G. The property which permits this is that "exclusive-or lines" can be connected in series, as shown in Fig. 3, to cross over an edge of G, when that edge is required to occur in any Hamiltonian circuit. The sequence of two exclusive-or's pictured there act like a single "exclusive-or line" joining the two outermost edges (B and D) while permitting the required edge (C) to pass between them. In particular, since all 4 vertical edges in an exclusive-or graph must occur in any Hamiltonian circuit, we can use this property to allow two "exclusive-or lines" to

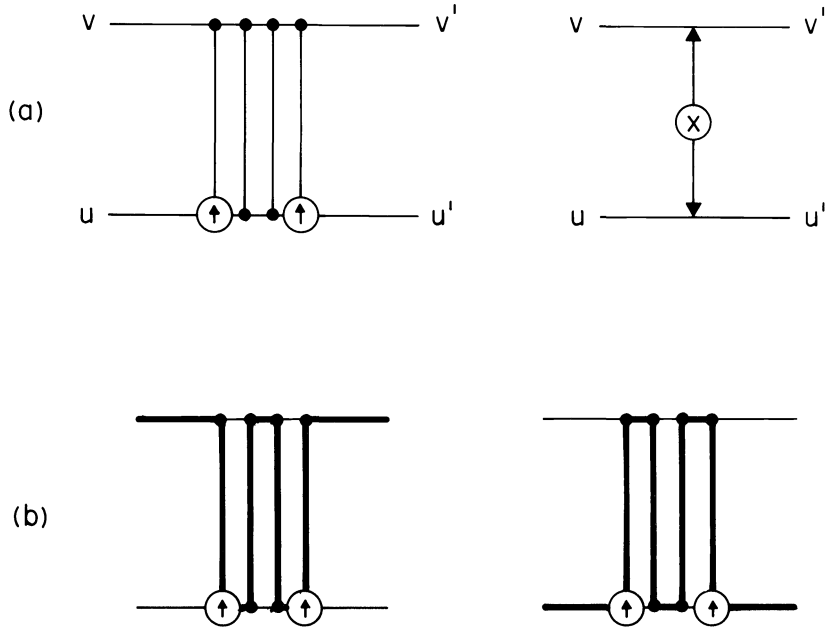


FIG. 2. *Exclusive-or*
(a) *Graph and abbreviation*
(b) *Possible local states*

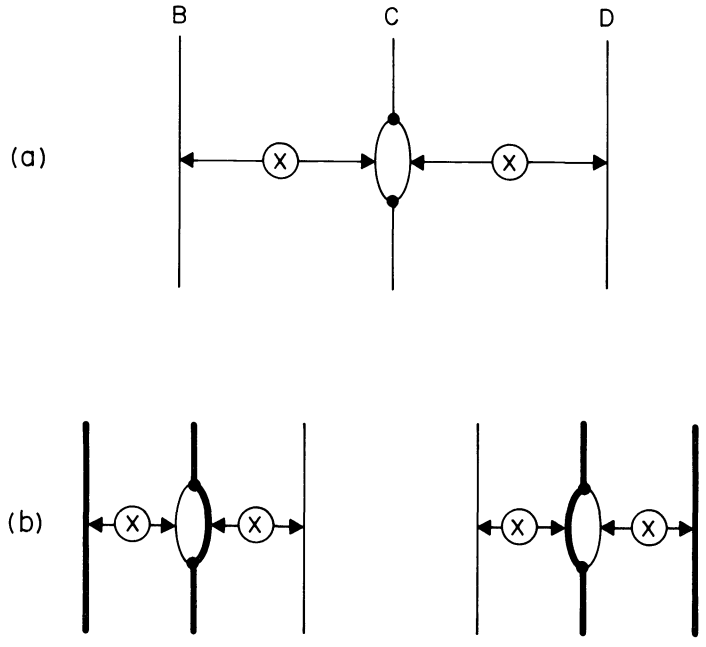


FIG. 3. *Exclusive-or's in series*
(a) *Schematic of graph*
(b) *Possible local states*

cross each other. Figure 4 shows schematically how this can be done and illustrates the possible states that can occur in a Hamiltonian circuit.

In addition to the exclusive-or, we will also use the two-input "or" graph of Fig. 5(a). Any Hamiltonian circuit in a graph G which contains this graph as a vertex-induced subgraph must appear locally in one of the states shown in Fig. 5(b). Thus this subgraph acts like two separate edges, one joining v to v' and the other joining u to u', with the constraint that at least one of these two edges must occur in any Hamiltonian circuit of G.

Finally we use the graphs in Figs. 1, 2 and 5 to construct the three-input "or" shown in Fig. 6. This subgraph acts like three separate edges, one joining v to v', one joining u to u', and one joining w to w', with the constraint that at least one of these three edges must occur in any Hamiltonian circuit of G.

With these components we can undertake the construction. For each of the variables x_i, $1 \leqq i \leqq n$, we construct four vertices v_{i1}, v_{i2}, v_{i3} and v_{i4}, and for each

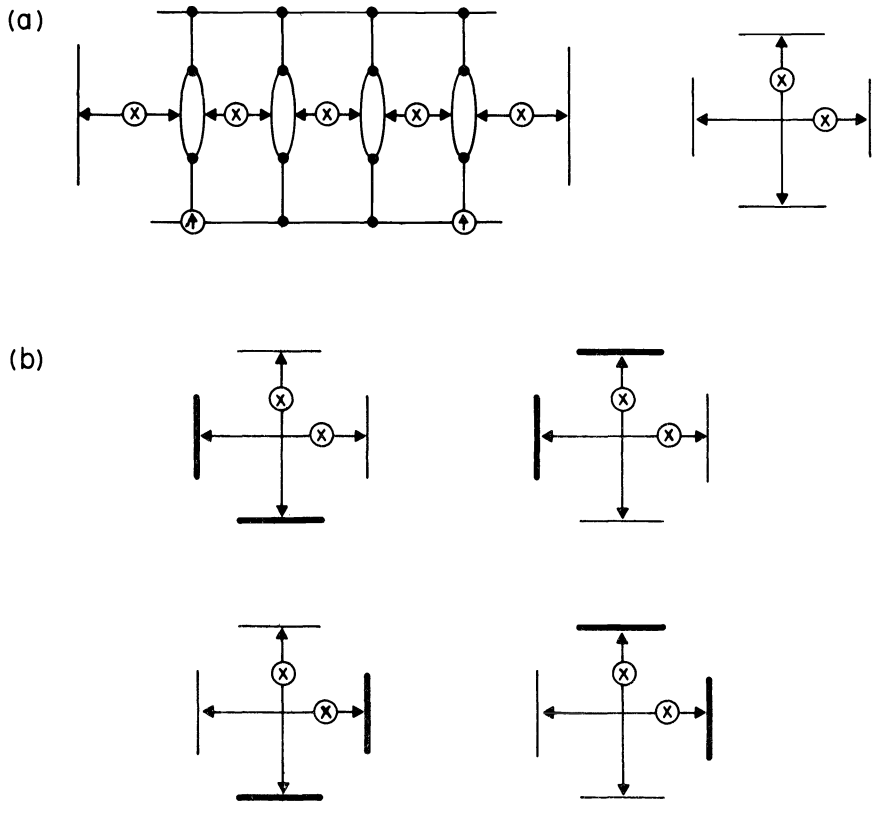


FIG. 4. *Crossing of exclusive-or's*
(a) *Graph and abbreviation*
(b) *Possible local states*

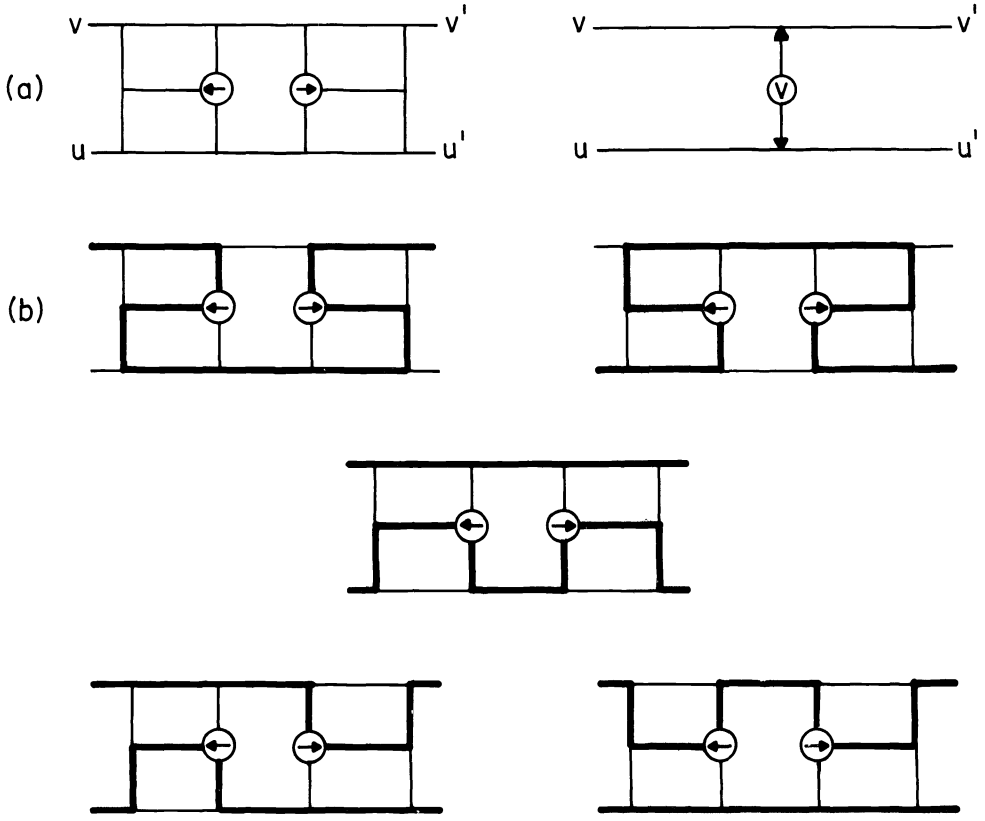


FIG. 5. *2-input or*
(a) *Graph and abbreviation*
(b) *Possible local states*

clause C_j, $1 \leqq j \leqq m$, we construct six vertices w_{j1}, w_{j2}, w_{j3}, w_{j4}, w_{j5}, w_{j6}. We start with the following skeletal edges:

(a) two copies each of $\{v_{i1}, v_{i2}\}$ and $\{v_{i3}, v_{i4}\}$, $1 \leqq i \leqq n$;
(b) $\{v_{i2}, v_{i3}\}$, $1 \leqq i \leqq n$;
(c) $\{v_{i4}, v_{i+1,1}\}$, $1 \leqq i \leqq n-1$;
(d) $\{v_{n4}, w_{m6}\}$;
(e) $\{v_{11}, w_{11}\}$;
(f) two copies each of $\{w_{j1}, w_{j2}\}$, $\{w_{j3}, w_{j4}\}$ and $\{w_{j5}, w_{j6}\}$, $1 \leqq j \leqq m$;
(g) $\{w_{j2}, w_{j3}\}$, $\{w_{j4}, w_{j5}\}$, $1 \leqq j \leqq m$;
(h) $\{w_{j6}, w_{j+1,1}\}$, $1 \leqq j \leqq m-1$

For each i, we join one copy of $\{v_{i1}, v_{i2}\}$ to one copy of $\{v_{i3}, v_{i4}\}$ with an "exclusive-or". For each j, we connect one copy each of $\{w_{j1}, w_{j2}\}$, $\{w_{j3}, w_{j4}\}$, and $\{w_{j5}, w_{j6}\}$ with a three-input "or" Observe that so far the construction is planar and depends only on the numbers m and n, rather than any more specific details about F.

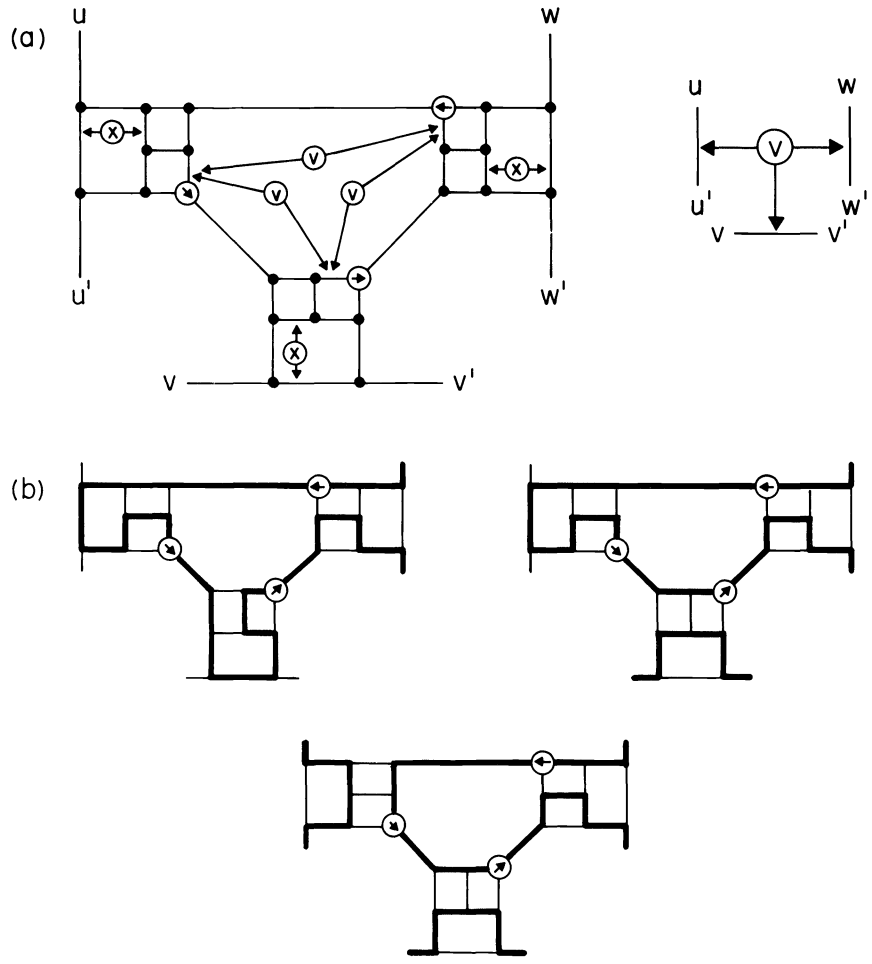


FIG. 6. *3-input or*
(a) *Graph and abbreviation*
(b) *Possible local states (symmetric cases not shown)*

Now let us consider each literal p_{jk} in F. If $p_{jk} = x_i$, we use an "exclusive-or" to join the copy of $\{w_{j,2k-1}, w_{j,2k}\}$ not connected to a three-input "or" to the copy of $\{v_{i1}, v_{i2}\}$ which is not joined to $\{v_{i3}, v_{i4}\}$ by an "exclusive-or". If $p_{jk} = \bar{x}_i$, we use an "exclusive-or" to join that copy of $\{w_{j,2k-1}, w_{j,2k}\}$ to a copy of $\{v_{i3}, v_{i4}\}$ which is not joined to $\{v_{i1}, v_{i2}\}$ by an "exclusive-or". Finally, we connect $\{v_{11}, w_{11}\}$ and $\{v_{n4}, w_{m6}\}$ to a two-input "or". (This is only used to ensure triple-connectedness; *both* edges in fact must be used in any Hamiltonian circuit of G.) See Fig. 7 for a schematic of this construction for $F = (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee w) \wedge (y \vee \bar{z} \vee \bar{w})$.

Our constructed graph is planar except for crossing "exclusive-or lines", which are made planar as in Fig. 4. Two "exclusive-or's" need only cross once so

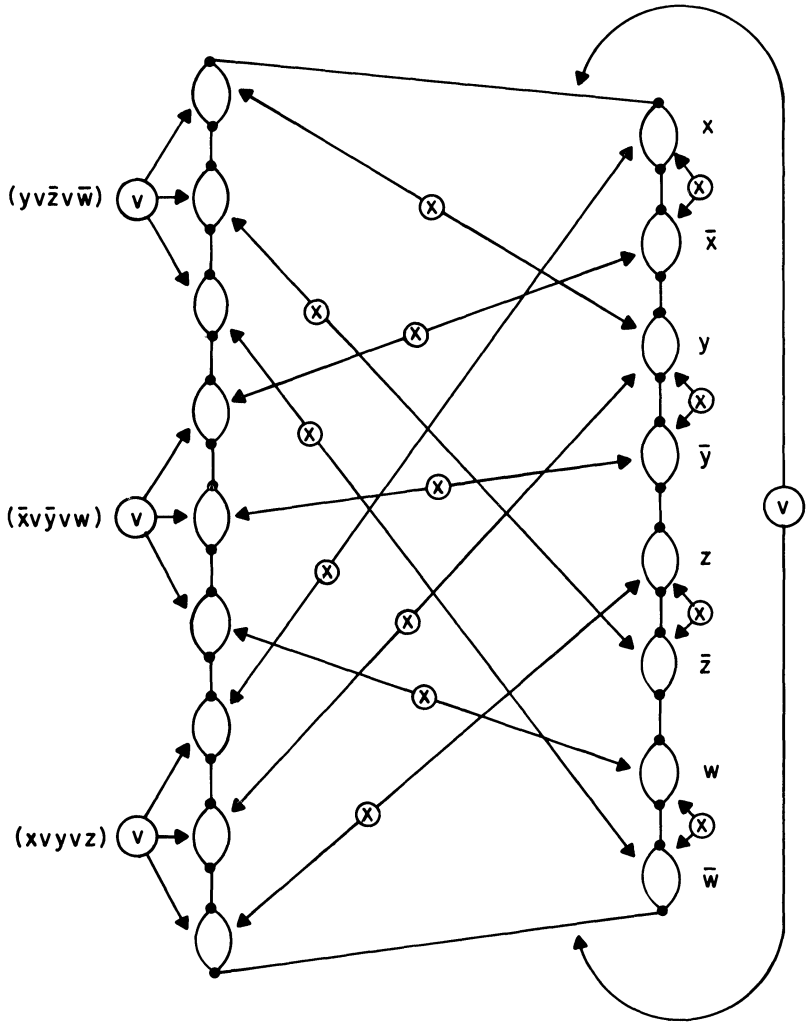


FIG. 7. *Complete construction for* $F = (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee w) \wedge (y \vee \bar{z} \vee \bar{w})$

the constructed graph has $O(m^2)$ vertices and edges and is easily constructed in polynomial time.

We leave to the reader the straightforward but tedious verification that the graph is cubic and triply connected. Basically, all one need do is verify that our special subgraphs have these properties (or would if their external edges were connected by external paths) and that the overall superstructure does also (while in addition providing the required "external paths"). Let us now see why the graph as it stands has a Hamiltonian circuit if and only if F is satisfiable.

Consider any Hamiltonian circuit in G. Of each pair of edges $\{v_{i1}, v_{i2}\}$, $\{v_{i3}, v_{i4}\}$ joined by an "exclusive-or", the circuit must use exactly one. If the circuit uses $\{v_{i1}, v_{i2}\}$ from this pair, we assign x_i the value *false*; otherwise x_i is assigned *true*.

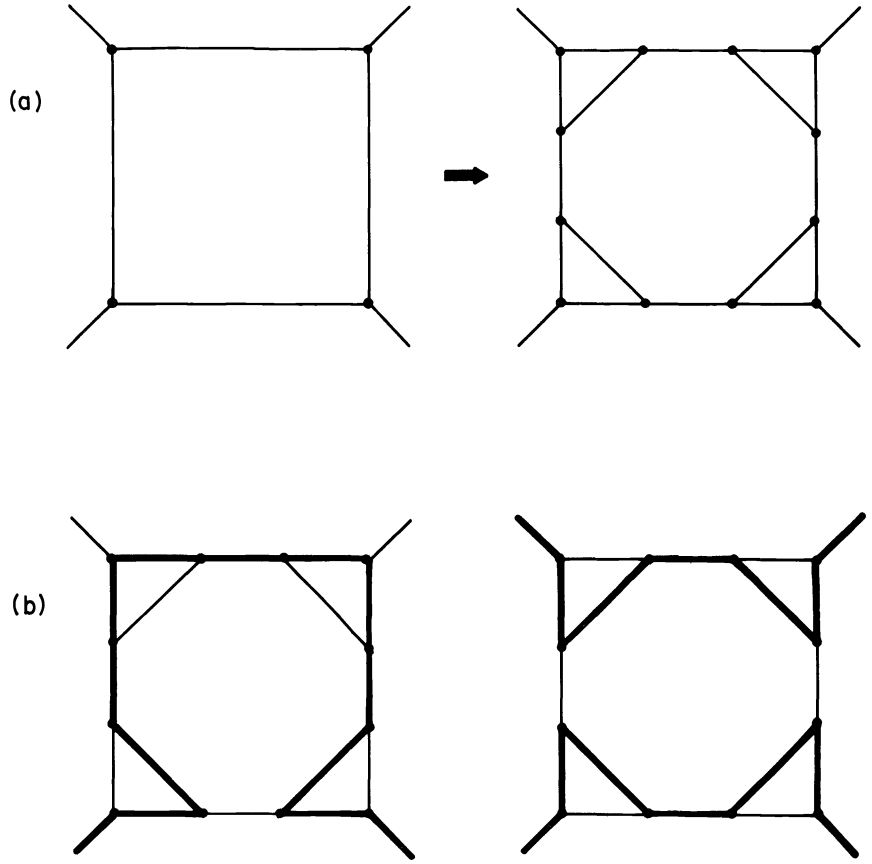


FIG. 8. *Elimination of four-sided face*
(a) *Substitution graph*
(b) *Possible local states (symmetric cases not shown)*

The "exclusive-or lines" connecting edges for variables to edges for clauses prevent the clause edges participating in the three-input "or" for that clause from being used unless the corresponding variable makes the clause *true*. Since at least one of those edges *must* be used in any Hamiltonian circuit, it follows that this truth setting must make each clause, and hence F itself, *true*. Similar reasoning shows that any truth assignment satisfying F can be used to construct a Hamiltonian circuit for G. Thus the Hamiltonian circuit problem for graphs satisfying (i)–(iii) is NP-complete.

From our construction, we can also conclude that the three general planar Hamiltonian problems that were left open in [4] are all NP-complete. The undirected planar Hamiltonian circuit problem is NP-complete because it contains our problem as a special case. The directed planar Hamiltonian circuit problem is NP-complete because we can replace every edge $\{u, v\}$ in our construction with two directed edges (u, v) and (v, u) and thus get a directed graph

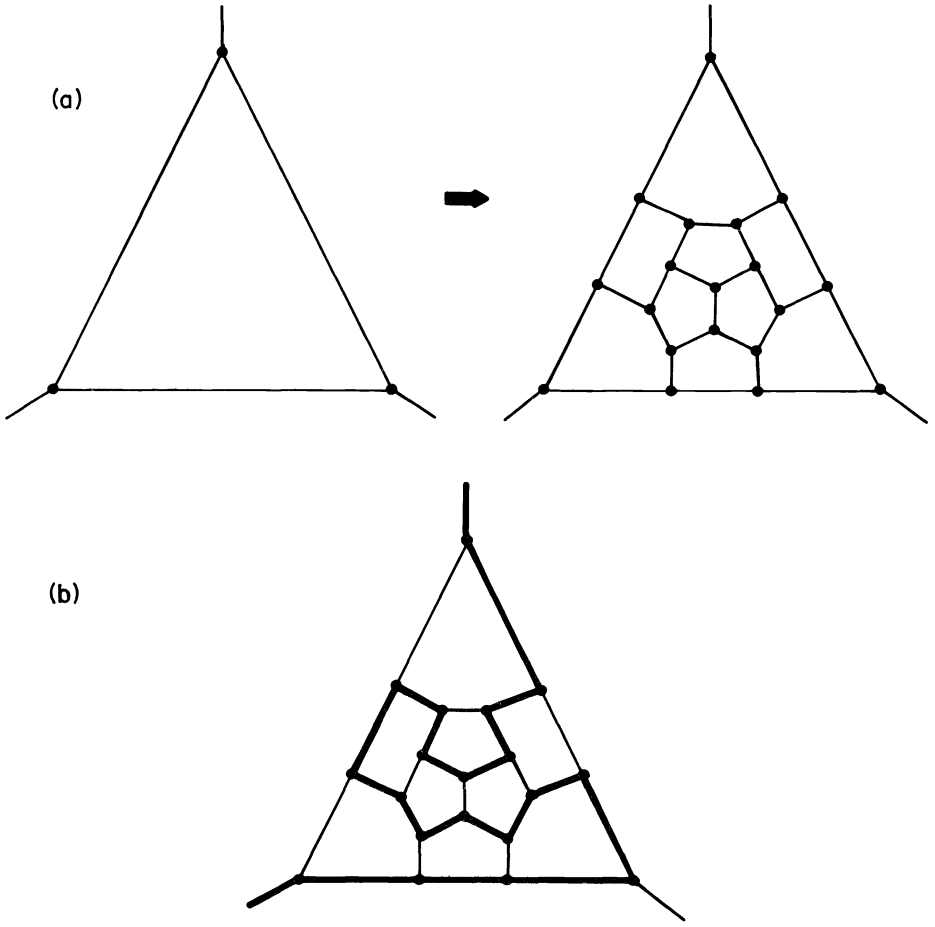


FIG. 9. *Elimination of triangle*
(a) *Substitution graph*
(b) *Possible local state (alternate and symmetric states not shown)*

which has a Hamiltonian circuit if and only if our original undirected graph had one. Finally, the undirected planar Hamiltonian line problem is NP-complete: convert the "or" linking edges $\{v_{11}, w_{11}\}$ and $\{v_{n4}, w_{m6}\}$ into an "exclusive or". A Hamiltonian line must either start at v_{11} and finish at w_{11}, or start at v_{n4} and finish at w_{m6}. Such a line will exist if and only if the original graph had a Hamiltonian circuit. Note that the construction preserves triple connectivity and degree threeness, as well as planarity.

We can also obtain a more specialized result that may be of interest. It has been shown (see [7, Thm. 8.4.1]) that the four color conjecture depends only on graphs in which each face has at least five boundary edges. We can show that the Hamiltonian circuit problem remains NP-complete, even if we restrict ourselves to graphs which have this property and obey (i)–(iii). We do this by taking our

original graph and eliminating its faces with four or fewer sides by making the following substitutions:

If G contains a four-sided face, introduce triangles into each of its four corners as shown in Fig. 8(a). Figure 8(b) illustrates the fact that all the ways that a Hamiltonian circuit might pass through the original face can be mimicked by its replacement, and clearly no new possibilities are introduced. Also, since *two* vertices are introduced into every edge of the original face, no external faces can be made into four-sided faces by this substitution. We further note that the graph remains planar, cubic and three-connected. Repeat the substitution until there are no more four-sided faces.

Triangles are now eliminated in an analogous step by step manner, with the substitution shown in Fig. 9. The result is a graph which is planar, cubic, three connected, has no face with fewer than five sides, and has a Hamiltonian circuit if and only if our original graph did. Since the replacement can clearly be accomplished in time proportional to the number of faces in the original graph, this means that the more restricted problem is also NP-complete.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] C. BERGE, *Graphs and Hypergraphs*, American Elsevier, New York, 1973.

[3] S. A. COOK, *The complexity of theorem proving procedures*, Proceedings of Third Annual ACM Symposium, on Theory of Computing, Association for Computing Machinery, New York, 1971, 151–158.

[4] M. R. GAREY, D. S. JOHNSON AND L. J. STOCKMEYER, *Some simplified NP-complete graph problems*, Theoretical Computer Science, 1 (1976), pp. 237–267.

[5] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972 pp. 85–104.

[6] ————, *On the computational complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

[7] O. ORE, *The Four-Color Problem*, Academic Press, New York, 1967.

[8] P. G. TAIT, *Remarks on the colouring of maps*, Proc. Royal Soc. Edinburgh, 10 (1880), p. 729.

[9] W. T. TUTTE, *On Hamilton circuits*, J. London Math. Soc. 21 (1946), pp. 98–101.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

AN ACCOUNT OF SELF-ORGANIZING SYSTEMS*

W. J. HENDRICKS†

Abstract. An account is given of existing results for two types of self-organizing schemes, and more direct proofs are provided. Several new results are included, and a variety of open problems of theoretical and practical interest are stated. The work has particular relevance to stochastic problems which arise in computing.

Key words. self-organizing systems, library-type Markov chains

1. Introduction. We consider the following scheme. A finite or countably infinite set of items $I_1, I_2, \cdots, I_N$ (N a positive integer or $+\infty$, assumed finite unless otherwise stated) is placed in linear array; that is, according to one of the $N!$ possible orderings on a line. One can imagine the items as books on a shelf with position numbered $1, 2, \cdots, N$ from left to right or records in a vertical stack numbered from top to bottom to give the problem a more colorful setting. At successive integral times $t_1, t_2, \cdots$, one of the items is called for. If the item called occupies position j on the shelf, $1 \leqq j \leqq N$, then a permutation τ_j is performed upon the existing order. If $\tau_1, \tau_2, \cdots, \tau_N$ are specified so that from a given ordering we can reach (not necessarily in one step) any other specified ordering by successive application of some or all of the τ_i, repetitions of the same τ_i being permitted, then we call the resulting $\tau = \{\tau_j\}_{j=1}^N$ a self-organizing scheme. Assume that each item I_i has a probability p_i of being selected and that the p_i satisfy:

(i) $p_i > 0$;
(ii) $\sum_{i=1}^N p_i = 1$;
(iii) the p_i remain fixed for times $t_1, t_2, \cdots$, and successive demands are independent (in the probabalistic sense of independence) of each other.

Particular examples of τ in which we shall be interested are the move-to-front scheme τ^* and the move-back-one scheme τ' defined by ($\tau_i(j)$ denotes the position, after application of τ_i, of the item lying in position j):

$$\tau^* = \{\tau_i\}_{i=1}^N, \quad \text{where } \tau_i(j) = \begin{cases} j+1 & \text{if } 1 \leqq j \leqq i-1, \\ 1 & \text{if } j = i, \quad i = 1, 2, \cdots, N; \\ j & \text{if } j > i, \end{cases}$$

$$\tau' = \{\tau_i\}_{i=1}^N, \quad \text{where } \tau_i(j) = \begin{cases} j-1 & \text{if } j = i \text{ and } j \neq 1, \\ j+1 & \text{if } j = i-1, \\ j & \text{otherwise.} \end{cases}$$

Thus, τ^* places whatever item is demanded at the left-hand end of the shelf and moves items one position to the right as necessary to make room. τ' exchanges an item which is demanded with the one immediately preceding it, while leaving the

* Received by the editors July 1, 1975, and in revised form January 7, 1976.

† Department of Mathematics and Statistics, Case Western Reserve University, Cleveland, Ohio 44106. During the preparation of this paper the author was partially supported by the Science Research Council of Great Britain, while visiting at Westfield College of the University of London.

remaining ones fixed. In either scheme, if the item demanded is at the left-hand end of the shelf, none of the items are moved until the next demand. We assume that items are replaced in the pile prior to the next demand.

The introduction of the p_i gives the entire problem a probabilistic setting. Hence at a given time we may not know where an item is on the shelf but hopefully we could calculate or estimate the probability of its being at or near a particular position. Moreover, given any initial ordering of the items and a particular τ, the system will reorganize itself. Thus we shall be interested in how the system behaves under a given τ, as well as what sort of τ might have desirable properties. Our purposes are threefold:

1. to provide a survey of known results on problems of this sort;
2. to arrive at some new results, as well as to provide new (and hopefully more illuminating) proofs of existing theorems;
3. to pose several problems of theoretical as well as practical interest;

A frequent tool which has been used in the analysis of the stochastic properties of self-organizing schemes has been that of Markov chains. Thus, each of the $N!$ arrangements is considered as a state in a Markov chain and one-step transition probabilities from one state to another are determined by the p_i. A vivid account of Markov chains is given in Feller [3]. For simplicity of terminology and statement of results, we assume that we consider only those τ which give rise to aperiodic chains. Roughly speaking, this means that transitions from one state to some other state are not restricted to certain integral multiples of time. Therefore any self-organizing system can be studied as a finite, aperiodic irreducible Markov chain. Such chains are ergodic and exhibit limiting (or "steady state") probabilities for each state, as discussed in the next section.

2. Known results. With the advent of computers, interest in self-organizing systems has shown a marked increase. There are many potential applications of such study to computers (see Burville [1] or Rivest [9]), and undoubtedly to other areas as well.

McCabe [8] studied the move-to-front scheme as early as 1965. We explain McCabe's work in the context of more recent efforts (Hendricks [4], [5]) and then give an easier proof of some of McCabe's results. In [4], we regarded the move-to-front scheme as determining an ergodic Markov chain with $N!$ states transition matrix $[p_{ij}]$.

From a given state (configuration) it is possible to move in one step to any one of N states, including the present one. The conditional probability of passage from state i to state j (under some numbering of states) by selection and return of n items is denoted by $p_{ij}^{(n)}$. According to the theory of Markov chains,

$$\lim_{n\to\infty} p_{ij}^{(n)} \equiv u_j > 0, \quad \text{where } \sum_{j=1}^{N!} u_j = 1, \tag{1}$$

exists for all j and is independent of i. The $\{u_i\}_{i=1}^{N!}$ is referred to as a stationary distribution and can be found as the unique solution to the following system of equations:

$$u_k = \sum_{j=1}^{N!} u_j p_{jk}, \qquad \sum_{k=1}^{N!} u_k = 1. \tag{2}$$

We can think of the u_k as the long-term probability of the system being in state k.

There are several quantities of interest in any ergodic self-organizing scheme:

$$p(i, m) = \sum' u_j,$$

the sum being over those j such that I_i is in position m;

$$\mu_i = \sum_{m=1}^{N} mp(i, m);$$

$$\mu = \sum_{i=1}^{N} p_i \mu_i.$$

Thus $p(i, m)$ is the long-term probability of I_i being in position m, μ_i is the expected position of I_i in the steady state and μ is a measure of the expected time searching for the next item requested, starting from the left.

McCabe showed that the expected search time, μ_{τ^*}, for the move-to-front scheme τ^* satisfied:

$$\mu_{\tau^*} = 1 + 2 \sum_{1 \leqq i < j \leqq n} \frac{p_i p_j}{p_i + p_j}.$$

This result was also obtained in [2] and [9]. An easier derivation, using the method of indicator random variables, follows. In the steady state, I_j will lie to the left of I_i iff item I_j was selected more recently than I_i. Consequently there exists a nonnegative integer k such that the last request for I_j has been followed by selection of k items other than I_i or I_j. Now define random variables (r.v.) $X^{(i)}$ and $X_j^{(i)}$ by:

$$X_j^{(i)} = \begin{cases} 1 & \text{if } I_j \text{ lies to the left of } I_i, \quad j \neq i, \\ 0 & \text{otherwise,} \end{cases}$$

$$X^{(i)} = \text{position of } I_i \equiv 1 + \sum_{i \neq j} X_j^{(i)}.$$

If we use $E[X]$ to denote expectation of a r.v. X and $P[A]$ the probability of an event A, we have:

$$E[X_j^{(i)}] = P[X_j^{(i)} = 1] = p_j \sum_{k=0}^{\infty} (1 - p_i - p_j)^k = \frac{p_j}{p_i + p_j},$$

and

$$\mu_i = E[X^{(i)}] = 1 + \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{p_j}{p_i + p_j}.$$

Finally

$$\mu_{\tau^*} = \sum_{i=1}^{N} p_i \mu_i = 1 + \sum_{i \neq j} \frac{p_i p_j}{p_i + p_j}.$$

Burville and Kingman [2] note that if $p_i \geqq p_{i+1}$ for $1 \leqq i \leqq N-1$ and $m = \sum_{i=1}^{N} i p_i$, we have $m \leqq \mu < 2m - 1$. One can regard m as the expected search time for the next item demanded if items are kept in order of decreasing popularity.

Thus τ^* at most doubles the expected search time over this scheme. If the p_i are known in advance, one would naturally store items in order of decreasing popularity to minimize μ. When the p_i are unknown, we seek a system which ultimately organizes itself to give the smallest μ. In particular, we seek a τ such that μ_τ is minimized regardless of the choice of the p_i. One of the major problems in the study and application of self-organizing schemes is to determine whether or not such a τ exists and, if so, what it is. This question prompted McCabe to suggest that the move-back-one scheme τ' might be more efficient asymptotically than τ^* in the sense that $\mu_{\tau'} \leqq \mu_{\tau^*}$. Rivest [9] has recently shown this to be true, and we give a variant of his proof in the next section.

Rivest provides some evidence in support of the conjecture that τ' is indeed optimal. In extensive simulation studies he has compared $\mu_{\tau'}$ with μ_τ, where τ is a family of permutations, each of which preserves the relative order of unrequested items. For Zipf's distribution with $N = 7$ and under 5,000 selections of items, τ' proved superior to the other six rules considered. Also, for $N = 4$ and the geometric distribution $p_1 = \frac{8}{15}$, $p_2 = \frac{4}{15}$, $p_3 = \frac{2}{15}$, $p_4 = \frac{1}{15}$, his simulations showed that τ', for this distribution of the p_i, had a smaller μ than any of the τ which could possibly be optimal. To do this, he first showed that an optimal τ, if it exists, must have certain properties; this reduced the number of cases he had to consider.

Another area of investigation of self-organizing schemes in which recent progress has been made is in the determination of the stationary distribution $\{u_i\}_{i=1}^{N!}$ given by the equations in (1) and (2). In [4] it was found that for the move-to-front rule τ^*, we have u_α given by (3), where α denotes the configuration $I_1 I_2 \cdots I_N$:

$$u_\alpha = \prod_{i=1}^{N} \left(\frac{p_i}{\sum_{j=i}^{N} p_j} \right), \tag{3}$$

with analogous formulas holding for other configurations. In [5] this was extended to the situation where replacement of requested items is always at a fixed position k, $1 < k \leqq N$, on the shelf and unrequested items are moved to the right or left as necessary to vacate position k. The resulting stationary distribution is given as a product of an appropriate binomial coefficient term and two expressions similar to (3). We observed that a mechanism for producing expressions such as (3) is to use a model of sampling without replacement, using renormalized conditional probabilities following successive selection of I_1, I_2 from $\{I_j\}_{j=2}^{N}, \cdots, I_{N-1}$ from $\{I_{N-1}, I_N\}$, and finally I_N from $\{I_N\}$. It would be helpful to establish the precise analytical connection between this mechanism and the scheme τ^*.

Another, and more illuminating, derivation of (3) can be given as follows. We consider the system to be in the state $I_1, I_2 \cdots I_N$ and seek the corresponding term in the stationary distribution. This can be achieved by beginning initially with a distribution of probabilities among the $N!$ states in accordance with those assigned by (3) and using the fact that after n transitions, the absolute probability of being in a given state remains unchanged. Alternatively, but less precisely, one can suppose that any initial distribution of probabilities was given, that an infinite number of transitions have been made, and that we are considering the "end result". In either case we look at the present state, say $I_1 I_2 \cdots I_N$, and attempt to reconstruct the past history by looking backwards in time. In order to be in state

$I_1 I_2 \cdots I_N$, the past history (listed from most remote to most recent) of selections must be:

1. A last selection of I_N is made.
2. Selections of items from $\{I_1, I_2, \cdots, I_{N-1}\}$ are made.
3. A last selection of I_{N-1} is made.
4. Selections of items from $\{I_1, I_2, \cdots, I_{N-2}\}$ are made.
5. A last selection of item I_{N-2} is made.

.
.
.

$2N-1$. A last selection of I_2 is made.

$2N$. At least one selection of I_1 is made.

Since the probabilities in step $2j$ are given by $p_1+p_2+\cdots+p_{N-j}(1 \leqq j \leqq N-2)$, the desired probability is:

$$p_N \cdot \sum_{k=0}^{\infty} (p_1+p_2+\cdots+p_{N-1})^k \cdot p_{N-1} \cdot \sum_{k=0}^{\infty} (p_1+p_2+\cdots + p_{N-2})^k \cdot p_{N-2} \cdots p_2 \cdot \sum_{k=1}^{\infty} p_1^k,$$

which gives the desired result.

Letac [6] gives an elegant extension of τ^* to the case $N=+\infty$. In order to obtain a countable state space, he considers only those configurations of $\{I_i\}_{i=1}^{\infty}$ which can be reached from the natural ordering $I_1 I_2 I_3 \cdots$ by a finite number of selections and replacements. Thus we consider only those states whose ordering agrees with natural ordering from some point onward. Letac uses generating functions and a careful combinatorial argument which again involves looking at time in reverse to obtain the limiting probability of being in a given state. He shows:

(i) the chain is transient iff

$$\sum_{n=1}^{\infty} \left(\prod_{i=1}^{n} \frac{p_i}{1-s_n} \right) < \infty,$$

where $s_n = \sum_{i=1}^{n} p_i$;

(ii) the chain is positively recurrent iff

$$\sum_{n=1}^{\infty} \frac{1-s_n}{p_n} < \infty,$$

and that in this case the term of the stationary distribution corresponding to $I_1 I_2 I_3 \cdots$ is

$$\prod_{i=1}^{\infty} \left(\frac{p_i}{\sum_{j=i}^{\infty} p_j} \right).$$

He also gives examples (in terms of $\{p_i\}$) of chains of each of the three types: transient, null recurrent, positive recurrent. Positive recurrence seems to require a fairly rapid rate of convergence of the p_i; Letac's example is $p_i = K e^{-i^{\alpha}}$ for some

$\alpha, K > 0$, whereas geometric type of p_i give rise to either null recurrent or transient chains.

Nelson [7] has studied the case $N = +\infty$ when replacement always takes place at some fixed position $k \geqq 2$. He showed, somewhat surprisingly, that under these circumstances a stationary distribution cannot exist. The difficulty in this situation arises from the fact that when replacements are at a fixed position other than the end, it becomes more difficult to obtain a desired ordering. To date, it has not been possible to establish the transience or null recurrence of chains of this type.

We conclude this section by giving brief mention of other aspects of self-organizing systems which have been studied. McCabe [8] gives, but does not derive, an expression for the variance of the position of the next item demanded under τ^*. Nelson [7] calculates the variance of the position of I_i in the stationary distribution for τ^*. He then constructs a prediction interval J such that J will, with prescribed probability, contain I_i. Nelson also obtains a ratio limit theorem for the n-step transition probabilities, and he calculates the Martin boundary for the nonergodic chains when $N = +\infty$. Finally, he considers a modification of the τ^* scheme by allowing for geometric holding times for items which are demanded. That is, items can be removed from the shelf and returned with probability $p < 1$ prior to the next demand. In many such problems, the combinatorics become quite complicated very quickly as N becomes large, but often a pattern does emerge as cases $N = 3$ or 4 are considered.

3. New results. By way of new results, we begin by finding a fairly manageable expression for the stationary distribution for the move-back-one scheme τ' and then give a theorem about the optimality problem.

THEOREM 1. *The term of the stationary distribution which corresponds to the state $I_1 I_2 \cdots I_N$ when the scheme is τ' is given by*:

$$\prod_{i=1}^{N} \frac{p_i^{N-i}}{\lambda_N(p_1, p_2, \cdots, p_N)},$$

where λ_N is defined successively by:

$$\lambda_2(x_1, x_2) = x_1 + x_2$$

$$\lambda_3(x_1, x_2, x_3) = x_1x_2(x_1 + x_2) + x_1x_3(x_1 + x_3) + x_2x_3(x_2 + x_3)$$

$$\lambda_4(x_1, x_2, x_3, x_4) = \sum_{\substack{1 \leqq i < j \\ < k \leqq 4}} x_i x_j x_k \lambda_3(x_i, x_j, x_k), \quad \textit{etc.}$$

Proof. Under τ', the state $I_1 I_2 \cdots I_N$ can be reached in one step by starting in any one of the N states listed below and selection of the item indicated:

$$\begin{array}{llllllll}
I_1 & I_2 & I_3 & I_4 & \cdots & I_{N-1} & I_N & \text{select } I_1, \\
I_2 & I_1 & I_3 & I_4 & \cdots & I_{N-1} & I_N & \text{select } I_1, \\
I_1 & I_3 & I_2 & I_4 & \cdots & I_{N-1} & I_N & \text{select } I_2, \\
I_1 & I_2 & I_4 & I_3 & \cdots & I_{N-1} & I_N & \text{select } I_3,
\end{array}$$

$$\cdot$$
$$\cdot$$
$$\cdot$$

$$I_1 \quad I_2 \quad I_3 \quad I_4 \quad \cdots \quad I_N \qquad I_{N-1} \quad \text{select} \quad I_{N-1}.$$

Number these states in descending order from 1 to N, and let $u_1, u_2, \cdots, u_N$ denote the corresponding terms in the stationary distribution. By the steady-state property, we have

$$u_1 = p_1u_1 + p_1u_2 + p_2u_3 + \cdots + p_{N-1}u_N. \tag{4}$$

We want to show that the u_i which are defined below satisfy (4):

$$u_1 = \prod_{i=1}^{N} \frac{p_i^{N-i}}{\lambda_N(p_1, p_2, \cdots, p_N)},$$

$$u_2 = p_2^{N-1}p_1^{N-2} \prod_{i=3}^{N} \frac{p_i^{N-i}}{\lambda_N(p_1, p_2, \cdots, p_N)},$$

$$u_3 = p_1^{N-1}p_3^{N-2}p_2^{N-2} \prod_{i=4}^{N} \frac{p_i^{N-i}}{\lambda_N(p_1, p_2, \cdots, p_N)},$$

$$\cdot$$
$$\cdot$$
$$\cdot$$

$$u_N = \prod_{i=1}^{N-2} p_i^{N-1} \frac{p_N}{\lambda_N(p_1, p_2, \cdots, p_N)}.$$

Since the denominator is assumed to be the same in each of the u_i, we neglect writing it. With the above expressions for u_i, the right-hand side of (4) becomes:

$$\prod_{i=1}^{N} p_i^{N-i}(p_1 + p_2 + \cdots + p_N) = \prod_{i=1}^{N} p_i^{N-i}.$$

This completes the proof once we observe that $\sum_{i=1}^{N!} u_i = 1$ by virtue of the manner in which λ_N is defined.

With these observations, the proof of the following corollary is obvious.

COROLLARY. *Let I_i and I_j be any two items, k an integer which satisfies $0 \leqq k \leqq N-2$ and α and β two configurations of $I_1, I_2, \cdots, I_N$ which satisfy*:

(i) *I_i precedes I_j in α*;
(ii) *I_j precedes I_i in β*;
(iii) *α and β are identical except for the positions of I_i and I_j*;
(iv) *there are exactly k items between I_i and I_j in α and β.*

Then the terms u_α and u_β corresponding respectively to α and β satisfy

$$u_\alpha / u_\beta = (p_i/p_j)^{k+1}.$$

Remarks. We can again imagine a mechanism at work in the τ' scheme if we regard (for state $I_1 I_2 \cdots I_N$) I_1 as being moved back $N-1$ places, I_2 back $N-2$

places, $\cdots$ to give factors $p_1^{N-1}, p_2^{N-2}, \cdots$. For a specific distribution $\{p_i\}_{i=1}^N$, the quantities μ_i and μ are now calculable under τ', but we have been unable to determine any concise expression for them.

Our second theorem gives a means of comparison of the expected search time for different self-organizing schemes. For ease of notation, we assume that $p_1 \geqq p_2 \geqq \cdots \geqq p_N > 0$. If this condition does not hold, simply relabel the I_i to achieve it. Let τ be any self-organizing scheme and $b_\tau(j, i)$ the probability (in the resulting stationary distribution) that I_j lies to the left of I_i. Let μ_τ denote the expected search time.

THEOREM 2. *Let τ and $\bar{\tau}$ be any two self-organizing schemes and suppose that for any $\{p_i\}_{i=1}^N$ and i, j such that $1 \leqq i < j \leqq N$, we have:*

$$b_{\bar{\tau}}(j, i) \leqq b_\tau(j, i) \tag{5}$$

Then $\mu_{\bar{\tau}} \leqq \mu_\tau$.

Remark. If a $\bar{\tau}$ exists such that (5) holds for all possible τ, then $\bar{\tau}$ will be optimal.

Proof. First observe that $b_\tau(i, j) + b_\tau(j, i) = 1$. Then use the notation as set forth in the introduction to obtain

$$\mu_\tau = \sum_{i=1}^N p_i E_\tau[X^{(i)}] = \sum_{i=1}^N p_i \Big(1 + \sum_{\substack{j=1 \\ j \neq i}}^N E_\tau[X_j^{(i)}]\Big)$$

$$= \sum_{i=1}^N p_i \Big(1 + \sum_{\substack{j=1 \\ j \neq i}}^N b_\tau(j, i)\Big).$$

Consequently,

$$\mu_{\bar{\tau}} = \sum_{i=1}^N p_i \Big[1 + \sum_{j>i} b_{\bar{\tau}}(j, i) + \sum_{j<i} (1 - b_{\bar{\tau}}(i, j))\Big]$$

$$= \sum_{i=1}^N i p_i + \sum_{i<j} (p_i - p_j) b_{\bar{\tau}}(j, i)$$

$$\leqq \sum_{i=1}^N i p_i + \sum_{i<j} (p_i - p_j) b_\tau(j, i) = \mu_\tau.$$

We now obtain Rivest's [9] result about the relative asymptotic efficiency of τ^* and τ'.

COROLLARY. $\mu_{\tau'} \leqq \mu_{\tau^*}$.

Proof. In § 2 we showed that $b_{\tau^*}(j, i) = p_j/(p_i + p_j)$. Now use the corollary of Theorem 1 to obtain (when $j > i$)

$$b_{\tau'}(j, i) = \sum_{k=0}^{N-2} (\textstyle\sum' \mu_\alpha)$$

(inner sum over α for which I_j precedes I_i, with exactly k items between I_j and I_i)

$$= \sum_{k=0}^{N-2} (p_j/p_i)^{k+1} (\textstyle\sum'' \mu_\alpha)$$

(inner sum as above with positions of I_j and I_i interchanged)

$$\leqq (p_j/p_i)(1-b_{\tau'}(j,i)),$$

from which we conclude that

$$b_{\tau'}(j,i) \leqq p_j/(p_i+p_j) \quad \text{if } j>i.$$

4. Open problems. We conclude by listing some of the open problems which arise from this discussion.

1. For a given $\{p_i\}_{i=1}^N$, find that τ (from the finite list of possible τ) for which μ_τ is minimized.

2. Determine whether or not a τ exists which minimizes μ_τ regardless of the choice of the $\{p_i\}_{i=1}^N$. Such a τ would give rise to an optimal self-organizing scheme.

3. Investigate the stationary distribution and expected search times for schemes other than τ' and τ^*.

4. For a given scheme, consider the case $N=+\infty$.

5. For a given τ, for which the stationary distribution can be found, determine a mechanism for the process in the stationary state and establish the analytical connection of the mechanism with the scheme τ.

6. Relax on the independence assumptions. That is, if I_i is selected at time n, assume its probability of selection at time $n+1$ is increased, or perhaps dscreased.

Note added in proof. Letac (U.E.R. de Mathématique, Université Paul Sabatier, Toulouse, France) has recently written an excellent account of libraries in the case $N=+\infty$. He discusses questions of transience and recurrence, and poses further problems for study.

REFERENCES

[1] P. J. BURVILLE, *Heaps: A concept of optimization*, J. Inst. Maths. Appl., 13 (1974), pp. 263–278.
[2] P. J. BURVILLE AND J. F. C. KINGMAN, *On a model for storage and search*, J. Appl. Probability, 10 (1973), pp. 697–701.
[3] W. FELLER, *An Introduction to Probability Theory and its Applications*, vol. 1, 2nd ed., John Wiley, New York, 1957.
[4] W. J. HENDRICKS, *The stationary distribution of an interesting Markov chain*, J. Appl. Probability, 9 (1972), pp. 231–233.
[5] ———, *An extension of a theorem concerning an interesting Markov chain*, Ibid., 10 (1973), pp. 886–890.
[6] GÉRARD LETAC, *Transcience and recurrence of an interesting Markov chain*, Ibid., 11 (1974), pp. 818–824.
[7] P. R. NELSON, *Library-type Markov chains*, Doctoral dissertation, Dept. Math. and Statist., Case Western Reserve Univ., Cleveland, Ohio, 1975.
[8] J. MCCABE, *On serial files with relocatable records*, Operations Res., 13 (1965), pp. 609–618.
[9] RONALD RIVEST, *On self-organizing sequential search heuristics*, Comm. ACM, 19 (1976), pp. 63–67.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

FINDING MINIMUM SPANNING TREES*

DAVID CHERITON† AND ROBERT ENDRE TARJAN‡

Abstract. This paper studies methods for finding minimum spanning trees in graphs. Results include

1. several algorithms with $O(m \log\log n)$ worst-case running times, where n is the number of vertices and m is the number of edges in the problem graph;
2. an $O(m)$ worst-case algorithm for dense graphs (those for which m is $\Omega(n^{1+\varepsilon})$ for some positive constant ε);
3. an $O(n)$ worst-case algorithm for planar graphs;
4. relationships with other problems which might lead to a general lower bound for the complexity of the minimum spanning tree problem.

Key words. equivalence algorithm, graph algorithm, minimum spanning tree, priority queue

1. Introduction. Let $G=(V, E)$ be a connected undirected graph with $|V|=n$ vertices and $|E|=m$ edges. Given a value $c(v, w)$ for each edge $(v, w)\in E$, we wish to find a spanning tree $T=(V, E')$, $E'\subseteq E$, such that $\sum_{\{v,w\}\in E'} c(v, w)$ is minimum. Several efficient algorithms exist for solving this problem [3, p. 179], [5], [10], [12], [13], [20]. All of these algorithms are based on the following lemma.

LEMMA A. *Let $X\subseteq V$. Let $\{v, w\}\in E$ be such that $v\in X$, $w\in V-X$, and $c(v, w)=\min\{c(x, y)|\{x, y\}\in E,\ x\in X,\ \text{and}\ y\in V-X\}$. Then some minimum spanning tree contains $\{v, w\}$.*

This paper presents a very general minimum spanning tree algorithm and studies its running time for various implementations. The paper also gives relationships between the minimum spanning tree problem and certain data manipulation problems which may lead to a general lower bound on the complexity of the problem. Section 2 describes the general algorithm, the data manipulation methods needed to implement it, and two "classical" minimum spanning tree algorithms. Section 3 discusses possible implementations for a crucial part of the algorithm. Section 4 proves an $O(m \log\log n)$ worst-case running time for several variations of the algorithm. Section 5 proves an $O(n)$ worst-case running time for planar graphs and an $O(m)$ worst-case running time for dense graphs. Section 6 gives relationships with other problems. Section 7 presents conclusions and open problems.

2. A general algorithm. To find a minimum spanning tree in a connected graph G, we use the following general method.

First step. Pick some vertex. Choose the smallest edge incident to the vertex. This is the first edge of the minimum spanning tree.

* Received by the editors June 10, 1975, and in revised form November 11, 1975.

† Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 361. This research was supported by the National Research Council of Canada.

‡ Computer Science Department, Stanford University, Stanford, California 94305. This research was supported by the National Science Foundation under Grant GJ-35604X1 and a Miller Research Fellowship at University of California, Berkeley; and by the National Science Foundation under Grants DCR72-03752 A02 and MCS75-22870 at Stanford University.

General step. The edges so far selected define a forest F (a graph consisting of a set of trees) which is a subgraph of G. (Isolated vertices, not incident to any edges selected so far, are regarded as trees with one vertex.) Pick a tree T in the forest. Select the smallest unselected edge $\{v, w\}$ incident to a vertex in T and to a vertex in a *different* tree T'. Delete all edges smaller than $\{v, w\}$ which are incident to T (these edges form cycles with edges in T). Add $\{v, w\}$ to the minimum spanning tree (updating the forest by combining T and T'). Repeat the general step until all vertices are connected.

Cleanup step. (This optional step may be executed after any general step. It simplifies future calculations by deleting superfluous edges.) Delete all unselected edges with both endpoints in the same tree of the forest. For each pair of trees connected by an unselected edge, delete all but the minimum such edge connecting the pair of trees.

It is immediate from Lemma A that this general method correctly finds a minimum spanning tree. The method requires a mechanism for choosing the tree T to examine and certain bookkeeping mechanisms for keeping track of the trees in the forest. To keep track of the vertices in each tree of the forest, we use a simple, efficient disjoint set union algorithm described in [8] and [15]. Given a collection of disjoint sets (in this case, sets of the vertices in each tree), the set union algorithm implements three operations:

FIND(x): returns the name of the set containing vertex x;

UNION(i, j): combines sets i and j, naming the new set i;

INIT(i, L): initializes set i to contain all vertices in list L.

The worst-case time required for $O(m)$ FINDs and $O(n)$ UNIONs is $O(m\alpha(m, n))$, where α is a functional inverse of Ackermann's function [15]; this time is bounded by $O(n \log^* n + m)$, where

$$\log^* n = \min \{i \mid \overbrace{\log \log \cdots \log}^{i \text{ times}} n \leqq 1\}$$

To keep track of the edges incident to each tree of the forest, we use a mechanism of priority queues. We maintain queues of all edges incident to each tree of the forest. We need three operations on these queues.

QUNION(i, j): combines queues i and j, naming the new queue i;

MIN(i): returns the smallest edge in queue i which has an endpoint outside tree i. MIN(i) also deletes this edge and all smaller edges from queue i. (Note: MIN(i) must use the FIND operation to test whether a given edge has an endpoint outside tree i.)

QINIT(i, L): initializes queue i to contain all edges in the list L.

Implementation of these priority queue operations is discussed in § 3.

An implementation of the general algorithm using these operations as primitives appears below. We assume that the graph G has vertex set $V = \{1, 2, \cdots, n\}$ and that for each vertex v, $I(v) = \{(v, w) \mid \{v, w\} \in E\}$ is a list of the edges incident to v. (Note that (v, w) is an *ordered* pair; the undirected edge $\{v, w\}$ is represented twice; once as (v, w) in $I(v)$ and once as (w, v) in $I(w)$.) Each tree i of the forest is represented by a set i of its vertices and by a queue i of its incident edges. Initially each vertex v is represented by the set $\{v\}$ and by a queue

consisting of the edges in $I(v)$. At all times during execution of the algorithm, every edge (v, w) in queue k satisfies FIND$(v) = k$.

```
algorithm MINSPAN;
    begin
        for i := 1 until n do
            begin
                INIT(i, {i});
                QINIT(i, I(i));
            end;
        while more than one tree do
            begin
                execute cleanup step if desired;
                pick some tree k;
                (i, j) := MIN(k);
                add edge (i, j) to minimum spanning tree;
                x := FIND(j);
                UNION(x, k);
                QUNION(x, k)
            end
    end MINSPAN;
```

There are two "classical" minimum spanning tree algorithms. One, due to Kruskal [12], works as follows: initially all edges are sorted by value. Then the edges are examined in order, smallest to largest. If an edge, when examined, connects two different trees in the forest of selected edges, it is selected and added to the forest. Otherwise it is deleted. The initial edge sorting eliminates the need for a priority queue mechanism; the running time of Kruskal's algorithm is $O(m \log n)$ for the sorting plus $O(m\alpha(m, n))$ for the necessary set union operations. Thus the total running time is $O(m \log n)$, and if the edges are initially given in sorted order, the running time is $O(m\alpha(m, n))$.

Another algorithm, discovered by Prim [13] and independently by Dijkstra [5], is a version of our general algorithm in which the same tree is considered at each execution of the general step. One priority queue is needed for this tree (all other trees of the forest F consist of single vertices). In the original implementation, this queue is represented as a single array, with one (possibly filled) position for each vertex not in the tree, giving the size of the smallest edge from the tree to that vertex. A general step requires examination of the entire array and thus takes $O(n)$ time. After each general step, a cleanup step is executed; this cleanup step requires only $O(n)$ time because only a single vertex at a time is added to the unique, growing tree. Thus the original implementation of the Prim–Dijkstra algorithm runs in $O(n^2)$ time. If implemented differently, the algorithm runs in $O(m \log n)$ time [10].

To beat $O(m \log n)$, we must use a good priority queue implementation plus a careful selection of the tree to be considered in the general step. Two selection rules lead to efficient algorithms. One, based on a minimum spanning tree algorithm of Sollin [3, p. 189], is to process the trees uniformly. The uniform selection rule works as follows. Initially all of the n trees (each a single vertex) are placed on a queue. At the general step, the first tree T at the front of the queue is

examined. When this tree T is connected to another tree T', both T and T' are deleted from the queue and the new combined tree is placed at the rear of the queue. It is easy to implement this selection rule so that the total overhead for selection is $O(n)$.

An alternative selection rule is the *smallest tree rule*: always examine the tree with the smallest number of vertices. This rule can also be implemented so that the total overhead for selection is $O(n)$ (though the constant of proportionality is higher than for the uniform selection rule). This is done by keeping track of the number of vertices in each tree, and by using an array A of size n. The array element $A(i)$ is a list of all trees with i vertices. Since the number of vertices in every tree is between 1 and n, and the number of vertices in a tree never decreases, we can use a pointer which marches along array A to find the tree with the smallest number of vertices. It is easy to update the array if each tree has a pointer to its position in the array.

3. Implementation of priority queues. We consider three implementations of priority queues, ranging from simplest to most sophisticated. Time bounds for the various implementations are given in Table 1.

TABLE 1

Time bounds for priority queue operations.
$m(i)$ = *number of edges in queue* i (*and list* L).
$s(i)$ = *number of sets in queue* i (*implementation* (b)).
$l(i)$ = *number of edges deleted from queue* i *by* MIN(i).
$h(i)$ = *number of delayed merges into queue* i (*implementation* (c)).

	INIT(i, L)	MIN(i)	QUNION(i)
(a) Unordered sets	$O(m(i))$	$O(m(i))+m(i)$ FINDs	$O(1)$
(b) Ordered subsets of size k	$O(m(i)\log k)$	$O(s(i)+l(i))$ $+s(i)+l(i)-1$ FINDs	$O(1)$
(c) Leftist trees with delayed merge	$O(m(i))$	$O\left((l(i)+h(i))\log\left(\frac{m(i)}{l(i)+h(i)}+1\right)\right)$ $+O(l(i)+h(i))$ FINDs	$O(1)$

(a) *Unordered sets.* The simplest representation of a priority queue i is as an unordered list of items. Then MIN(i) requires $O(m(i))$ time plus time for $m(i)$ FINDs, QUNION(i, j) requires $O(1)$ time, and INIT(i, L) requires $O(m(i))$ time, where $m(i)$ is the size of queue i (and of L).

(b) *Ordered subsets of size k.* Another possibility is to represent a queue i as a list of sets, each set initially of size k, plus possibly some "special" sets, each initially of size less than k. Each set is in sorted order, but there is no ordering relationship among the sets. We carry out INIT(i, L) by dividing L into $\lfloor |L|/k \rfloor$ sets of size k plus at most one set of size less than k. We then sort these sets. This process requires $O(|L| \log k)$ time. We carry out QUNION(i, j) by merging the

list of sets in queue i and the list of sets in queue j. This requires $O(1)$ time. We carry out MIN(i) by inspecting the edges in each set of queue i in order, discarding those that don't connect two different trees. We then compare the smallest edge from each set and select the overall minimum edge. MIN(i) requires $O(s(i)+l(i))$ time plus time for $s(i)+l(i)-1$ FINDs, where $s(i)$ is the number of sorted sets in queue i and $l(i)$ is the number of edges deleted from queue i by MIN(i).

A similar priority queue method was used by Yao [20] in the first $O(m \log\log n)$ time minimum spanning tree algorithm. However, initialization of Yao's data structure requires the use of a linear-time median-finding algorithm such as [4], which is complicated to implement and requires more comparisons than the sorting used here for initialization.

(c) *Leftist trees with delayed merge.* This representation is an extension of one discovered by Crane [11]. In Crane's implementation, each queue is represented by a leftist tree. (A leftist tree is a binary tree such that, given any node v, there is a shortest path from v to a node with a missing left or right son, such that this path contains the right son of v.) Each node in such a tree represents an edge in the corresponding queue. The nodes are heap-ordered (ordered so that the node with smallest value is at the root of the tree and any node has value smaller than the values of both its sons).

A basic operation on two leftist binary trees is:

MERGE(i, j): combines trees i and j into a single leftist binary heap-ordered tree named i.

The MERGE operation can be carried out by finding, in each tree, a shortest (rightist) path from the root to a missing node, merging the two paths into a single path on which the nodes are sorted by value, attaching the remaining subtrees of each original tree to the appropriate nodes on the combined path, and switching left sons and right sons of nodes along the path (if necessary) to make the new tree leftist. To implement this operation, we must store four parameters with each node: its value, pointers to its left and right sons, and the length of the shortest path from the node to a missing node. See [11] for implementation details. Figure 1 illustrates such a MERGE operation. Since a leftist binary tree with $m(i)$ nodes has a rightist path of at most $\log(m(i)+1)$ nodes, the time required for MERGE(i, j) is $O(\log(m(i))+\log(m(j))+1)$. (All logarithms in this paper are base two.)

Here we extend Crane's idea. We represent each priority queue by a binary tree. Some of the nodes in the tree (called *queue nodes*) correspond to edges in the graph. The rest of the nodes (called *dummy nodes*) correspond to previous QUNION operations. Each queue node is the root of a subtree which consists only of queue nodes and is leftist and heap-ordered. The dummy nodes define a subtree rooted at the root of the entire tree.

Each node v requires five associated parameters:

leftson(v), rightson(v): pointers to the left and right sons of v;

path(v): the length of the shortest path from v to a missing element (only necessary if v is a queue node);

$c(v)$: the value of the edge associated with v (only defined if v is a queue node);

$q(v)$: a Boolean variable true if and only if v is a queue node.

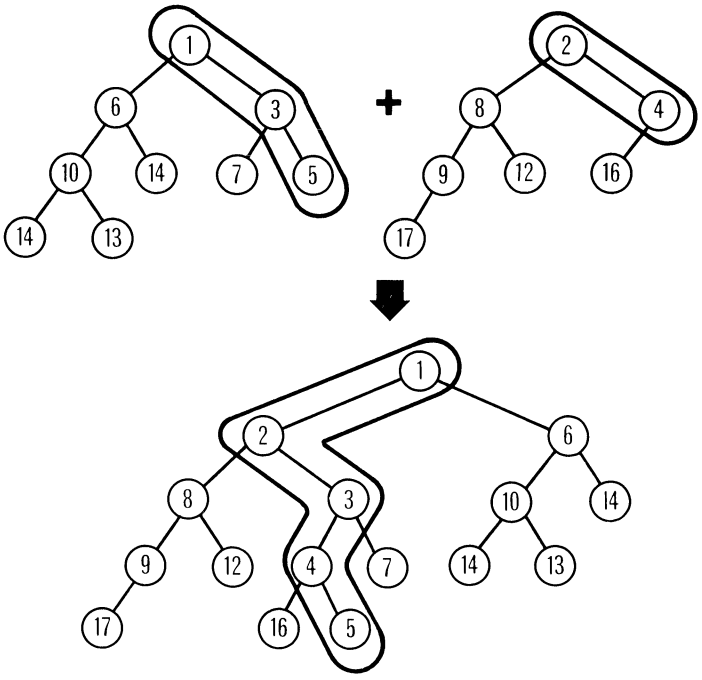


FIG. 1. *Merging two leftist binary trees*

To carry out QUNION(i, j) using this data structure, we create a new dummy node, make the roots of the trees for queues i and j the left and right sons of the new node, and mark the new node as the root of the new tree for queue i. QUNION clearly requires $O(1)$ time. We call this operation *delayed merge* of queue j into queue i.

We carry out MIN(i) in three steps. Suppose there are d dummy nodes in queue i. First, we explore the binary tree for queue i, starting from the root and working through its descendants, stopping at queue nodes which correspond to edges connecting two different trees in the forest. That is, we locate the set of queue nodes $\{v | v$ is in queue i, v represents an edge connecting two different trees, and no ancestor of v represents an edge connecting two different trees$\}$. We discard all ancestors of such minimal elements. The number of nodes discarded is $d+l-1$, where l is the total number of edges which will be removed from queue i by the MIN(i) instruction.

We are left with $d+l$ or fewer leftist binary trees, each rooted at one of the minimal nodes. We place these trees in a queue, merge the first two trees in the queue using MERGE, insert the resultant leftist tree at the rear of the queue, and repeat until only one tree is left. The root of this leftist, heap-ordered tree represents the desired edge of minimum value connecting two different trees. We select this edge and convert the root of the leftist tree to a dummy node. (This step has the effect of discarding one more element from queue i.)

The overall effect of the MIN(i) operation is to discard from tree i all dummy nodes and all queue nodes up to and including the one of minimum value

connecting two different trees and to combine the remaining nodes into a single leftist tree with a single dummy node, namely the root. An implementation for MIN(i) is presented below, in ALGOL-like notation. The implementation uses a recursively programmed procedure SEARCH, which explores a binary tree to find the minimal queue nodes connecting two different trees.

```
procedure MIN(i);
    begin
        Boolean procedure p(x);
            begin
                let (v, w) be the edge corresponding to queue node x;
                p := (i ≠ FIND(w));
            end;
        procedure SEARCH(x);
            comment we assume that the next if statement is implemented
                so that p(x) is evaluated only if q(x) = true;
            if q(x)∧p(x) then add tree rooted at x to L;
            else
                begin
                    if leftson(x) ≠ 0 then SEARCH(leftson(x));
                    if rightson(x) ≠ 0 then SEARCH(rightson(x));
                end;
        L := the empty list;
        let r be the root of tree i;
        SEARCH(r);
        while |L| > 1 do
            begin
                delete first two trees j and k from L;
                MERGE(j, k);
                insert new tree j at rear of l;
            end;
        tree remaining on L is new tree i;
        let r be the root of this tree;
        MIN := edge associated with r;
        q(r) := false;
    end MIN;
```

Suppose MIN is applied to queue i, initially containing $m(i)$ queue nodes and d dummies, and that l queue nodes are deleted by MIN. The running time of MIN is $O(l+d)$ plus the time required for $2l+d-1$ FIND operations plus the time required to merge $l+d$ or fewer trees formed from $m(i)-l+1 \leqq m(i)$ nodes.

During the merging process, the original trees (together containing all the remaining nodes) are merged in pairs, leaving no more than $\lceil (l+d)/2 \rceil$ trees containing all the nodes. These trees in turn are merged in pairs, and the process continues until one tree is left. Let $b = \lceil \log (l+d) \rceil$. A bound on the total merge time is

$$O\left(\sum_{j=0}^{b} \max \left\{ \sum_{k=1}^{2^{b-j}} \log (n_k + 1) \,\middle|\, \sum_{k=1}^{2^{b-j}} n_k \leqq m(i), n_k \geqq 0 \text{ for all } k \right\} \right).$$

The maximum is achieved when all the terms in the inner sum are equal, so the merge time is

$$O\left(\sum_{j=0}^{b} 2^{b-j} \log\left(\frac{m(i)}{2^{b-j}}+1\right)\right) = O\left((l+d)\log\left(\frac{m(i)}{l+d}+1\right)\right).$$

Note that d, the initial number of dummy nodes in tree i, is at most one plus twice the number of queues merged into queue i between MIN(i) instructions. (Here we count all queues merged into queue i through a sequence of QUNION instructions with no intervening MIN instruction.) In the use of priority queues in this paper, only one MIN instruction is performed on each queue, after which the queue is merged into another queue. Thus in this case, $d \leqq 2h+1$, where h is the number of queues merged into queue i by delayed merge. Hence the total time for the MIN(i) instruction is

$$O\left((l+2h+1)\log\left(\frac{m(i)}{l+2h+1}+1\right)\right) = O\left((l+h)\log\left(\frac{m(i)}{l+h}+1\right)\right)$$

plus time for $2l+2h$ FINDs.

To carry out INIT(i, L), we interpet each element of L as a leftist tree consisting of a single node. We place these trees into a queue and merge them as in MIN. The bound above reduces to $O(|L|)$ in this case.

It is worth noting that the delayed merge idea can be used with other priority queue data structures, such as 3-2 trees [1] and S_n trees with a heap ordering [19], to achieve the same time bounds as given above. It is unclear which data structure achieves the best constant factor in the running time.

By using an idea of Aho, Hopcroft and Ullman [2], we can implement the following additional operation on queues:

ADD(i, x): adds a constant of x to the value of every queue node in queue i.

To implement this operation, we do not use the value $c(v)$ to encode the true value of queue node v, but instead maintain the values $c(v)$ so that the true value of queue node v is $\sum_{j=1}^{k} c(v_j)$, where $r = v_1, v_2, \cdots, v_k = v$ is the sequence of (dummy and queue) nodes on the path from the root r to v in the leftist tree for queue i. To carry out ADD(i, x), we add x to $c(r)$. Thus ADD(i, x) requires $O(1)$ time. INIT, MIN and QUNION must be modified slightly (this does not change their time bounds). The ADD operation is not useful for finding minimum spanning trees, but it is useful for finding optimum branchings [17].

4. Implementations without cleanup. In this section we analyze the running time of MINSPAN, implemented without cleanups, for various combinations of selection rules and priority queue implementations. Table 2 summarizes the time bounds derived in this section.

MINSPAN is implemented so that after MIN(i) is executed, queue i is merged into some other queue. MIN(i) is executed exactly $n-1$ times, each time for a different value of i. Without loss of generality, assume that the vertices of the graph are numbered so that the ith execution of MIN is MIN(i). Let $m(i)$ be the number of edges in queue i when MIN(i) is executed. For each i, $1 \leqq i \leqq n-1$, let T_i be the tree in F containing vertex i when MIN(i) is executed. Let T_n be the minimum spanning tree constructed by MINSPAN.

TABLE 2

Time bounds for implementations without cleanup

	Uniform selection	Smallest tree selection
(a) Unordered sets	$O(m \log n)$	$O(m \log n)$
(b) Ordered subsets of size 1, $\log \log n$, $\log n$	$O(m \log \log n)$ ($2m \log \log n + O(m \log \log \log n)$ comparisons)	$O(m \log \log n)$ ($2m \log \log n + O(m \log \log \log n)$ comparisons)
(c) Leftist trees with delayed merge	$O(m \log \log n)$	$O(m \log \log n)$

Suppose the uniform selection rule is used. For each tree T which ever occurs in F, we define a *stage number* stage(T) for T by stage(T) = 0 if T contains a single vertex, stage(T) = min {stage(T'), stage(T'')} + 1 if T is formed by connecting trees T' and T'' by an edge. It is easy to prove by induction that if stage(T) = j, then T has at least 2^j vertices, so there are at most $(\log n) + 1$ stages.

LEMMA 1. *With uniform selection, two different trees T_i and T_j with the same state number are vertex disjoint.*

Proof. Suppose trees T_i and T_j share a vertex. Then either $T_i \subseteq T_j$ or $T_j \subseteq T_i$. Without loss of generality, assume $T_i \subseteq T_j$. The trees T initially on the queue used to implement the uniform selection rule have nondecreasing stage numbers. Furthermore, the uniform selection rule preserves this property as the algorithm proceeds. Just before MIN(i) is executed, T_i has a stage number as small as any tree on the queue. Thus if $T_i \neq T_j$, all trees T on the queue which are subtrees of T_j have stage(T) $\geqq$ stage(T_i), which means that stage(T_j) $\geqq$ stage(T_i) + 1. Hence $T_i \neq T_j$ implies stage(T_i) $\neq$ stage(T_j). □

COROLLARY 1. *With uniform selection,*

$$\sum_{\text{stage}(T_i)=k} m(i) \leqq 2m \quad \textit{for any constant } k,$$

$$\sum_{i=1}^{n-1} m(i) \leqq 2m \log n.$$

Proof. The proof is immediate from Lemma 1, since each edge is represented twice in the queues, and $0 \leqq \text{stage}(T_i) \leqq \log n - 1$ if $1 \leqq i \leqq n-1$ by the proof of Lemma 1 and the fact that $|T_i| \geqq 2^{\text{stage}(T_i)}$. □

For the smallest tree selection rule, we have similar results, but we must define stage(T) somewhat differently. For smallest tree selection, let stage(T) = $\lfloor \log |T| \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer not greater than x. Clearly there are at most $(\log n) + 1$ stages.

LEMMA 2. *With smallest tree selection, two different trees T_i and T_j with the same stage number are vertex disjoint.*

Proof. Suppose T_i and T_j share a vertex. Then $T_i \subseteq T_j$ or $T_j \subseteq T_i$, and without loss of generality we may assume $T_i \subseteq T_j$. After MIN(i) is executed, T_i is connected by an edge to some tree T having $|T| \geqq |T_i|$. Thus if $T_i \neq T_j$, $|T_j| \geqq 2|T|$ and stage(T_j) $\geqq$ stage(T_i) + 1. □

COROLLARY 2. *With smallest tree selection,*

$$\sum_{\text{stage}(T_i)=k} m(i) \leqq 2m \quad \text{for any constant } k,$$

$$\sum_{i=1}^{n-1} m(i) \leqq 2m \log n.$$

For either selection rule, define the *i-th stage* of the minimum spanning tree algorithm to be the time during which the algorithm performs MIN operations on trees with stage number i. (The ith stage includes time spent combining these trees via UNIONs to form trees with stage number $\geqq i+1$.)

With these preliminaries, we can now bound the running time for the algorithm, assuming no cleanups are performed, for various priority queue implementations. Suppose implementation (a) (unordered lists) is used. Since only $n-1$ UNION operations are ever carried out, and since $\sum_{i=1}^{n-1} m(i) \leqq 2m \log n$, the time for priority queue and other operations is $O(m \log n)$ plus time for $O(m \log n)$ FINDs and $O(n)$ UNIONs. The time for these operations is $O(m \log n)$ [15]. Thus the total time is $O(m \log n)$ (for either the uniform or the smallest tree selection rule).

Suppose implementation (b) (ordered subsets of size k) is used. Suppose we initialize the queues at the beginning of the pth stage of the algorithm and that we use this queue implementation until the qth stage of the algorithm. Since there are no more than $n/2^p$ queues at the beginning of the pth stage, after initialization there are no more than $2m/k + n/2^p$ subsets in all the queues. The queue initialization time is $O(m \log k)$. The time required to execute any stage j is proportional to $m/k + n/2^p + L(j)$, where $L(j)$ is the number of edges deleted from queues during stage j (see Table 1). Thus the total time required to initialize and execute from stage p to stage q is $O([m/k + n/2^p](q-p) + m)$.

Now consider the following implementation of MINSPAN:

Step 1. Initialize all queues as unordered sets.

Step 2. Execute MINSPAN until stage $\log \log \log n$.

Step 3. Re-initialize all queues as lists of ordered subsets of size $k_1 = \log \log n$.

Step 4. Execute MINSPAN until stage $\log \log n$.

Step 5. Re-initialize all queues as lists of ordered subsets of size $k_2 = \log n$.

Step 6. Execute MINSPAN to completion.

The total time required for this process is $O(m + m \log \log \log n + m \log \log n)$ for initialization and re-initialization of queues plus $O(m \log \log \log n + m + m)$ for stage execution (including all UNIONs, FINDs, MINs, and QUNIONs). Thus the total time is $O(m \log \log n)$ (for either selection rule). Furthermore, the running time is asymptotically dominated by the time required for sorting sets of size $\log n$ in Step 5; if all sorting is done using a fast sorting method [11], then this version of MINSPAN requires $2m \log \log n + O(m \log \log \log n)$ comparisons. Yao's $O(m \log \log n)$ minimum spanning tree algorithm requires at least $6m \log \log n$ comparisons if the best median-finding algorithm known is used; Yao's algorithm is also more complicated to implement than ours. The running time of our algorithm is still $O(m \log \log n)$ (though the constant factor may grow) if Steps 2 and 3 are dropped.

Now suppose queue implementation (c) (leftist trees with delayed merge) is used. For each i, let $l(i)$ be the number of edges deleted from queue i during execution of MIN(i), and let $h(i)$ be the number of queues merged into queue i by delayed merge. Clearly $\sum_{i=1}^{n-1} l(i) \leqq 2m$ and $\sum_{i=1}^{n-1} h(i) \leqq n-1$. To estimate the running time of MINSPAN, we must bound

$$\sum_{i=1}^{n-1} (l(i)+h(i)) \log\left(\frac{m(i)}{l(i)+h(i)}+1\right).$$

This will give a bound on the time required for all MIN operations (see Table 1).

We can break this sum into two parts: a sum over i such that $l(i)+h(i) \leqq m(i)/(\log n)^2$ and a sum over i such that $l(i)+h(i) > m(i)/(\log n)^2$. The sum is then bounded by

$$\sum_{i=1}^{n-1} \frac{m(i)}{\log n} + \sum_{i=1}^{n-1} (l(i)+h(i)) \log((\log n)^2+1)$$

$$\leqq 2m + (2m+n-1)2 \log(\log n+1) = O(m \log\log n).$$

The time for all other operations is also $O(m \log\log n)$. Thus queue implementation (c) (with either selection rule) gives an $O(m \log\log n)$ algorithm.

The time bounds computed in this section are summarized in Table 2. Though queue implementations (b) and (c) give better asymptotic running times than queue implementation (a), other factors, such as ease of implementation, constant factors, and lower order terms in the running time, may govern the best choice of implementation in practice. Experimental tests of these algorithms have yet to be done.

5. Implementations with cleanup. The cleanup step is useful for graphs in which MINSPAN generates many redundant edges when it combines trees. Two such cases are (i) planar graphs and (ii) dense graphs (graphs for which m/n is large). This section analyzes several versions of MINSPAN with cleanup. Table 3 summarizes the time bounds derived here.

TABLE 3

Time bounds for implementation with cleanup

	Uniform selection	Smallest tree selection
(a) Unordered sets, cleanup every stage	$O(\min(m \log n, n^2))$; $O(n)$ if planar.	$O(\min(m \log n, n^2))$; $O(n)$ if planar.
(b) Ordered subsets		
(c) Leftist trees with delayed merge, cleanup every $[\log a(j)]$th stage		$O\left(m \log\left(\frac{\log n}{\log\left(\frac{m}{n \log n}\right)}\right)\right)$ if $m > n(\log n)^3$; $O(m \log\log n)$ otherwise.

To execute a cleanup, we assign the number i to each vertex in set i. Then we replace each edge $\{v, w\}$ by a corresponding edge whose endpoints are the numbers assigned to v and w. (For each new edge $\{v', w'\}$, we remember the corresponding edge $\{v, w\}$ in the original graph.) We sort the new edges $\{v', w'\}$ lexicographically, using a two-pass radix sort [1]. We delete all new edges $\{v', w'\}$ with $v' = w'$, and we replace multiple copies of an edge $\{v', w'\}$ with $v' \neq w'$ by a single copy whose value is the minimum of the values of all the copies. This entire process requires $O(m)$ time. We then re-initialize the queues and the sets representing the trees F.

The net effect of these operations is to shrink each tree to a single vertex and to delete loops and multiple edges. Each vertex in the resulting new graph corresponds to a tree in the original graph. Subsequently, when selecting an edge for the spanning tree, we use the corresponding original edge $\{v, w\}$ rather than the new edge $\{v', w'\}$.

Suppose queue implementation (a) is used and that a cleanup is performed after each stage. The time for a cleanup is $O(m)$ (including re-initialization time), so the total cleanup time is $O(m \log n)$ and the total running time is $O(m \log n)$, using the bound from § 4.

After stage j, there are at most $n/2^{j+1}$ trees. Thus there are $n/2^{j+1}$ new vertices and $(n/2^{j+1})^2$ edges remaining after the cleanup following stage j. If $m(i)$ is the number of edges in queue i when MIN(i) is executed, then $\sum_{i=1}^{n-1} m(i) \leqq \sum_{j=0}^{\infty} (n/2^j)^2 = O(n^2)$. Thus MINSPAN, with queue implementation (a) and a cleanup after every stage, runs in $O(\min(m \log n, n^2))$ time, and the running time is linear in m for graphs with $m \sim n^2$. This algorithm, with the uniform selection rule, is a version of a method proposed originally by Sollin [3, p. 179].

Suppose the problem graph is planar. Any planar graph has $m \leqq 3n - 3$ ($m \leqq 3n - 6$ if $n \geqq 3$). After a cleanup, the edges remaining must define a planar graph whose vertices correspond to the subtrees of the minimum spanning tree so far constructed. Thus, for planar graphs, $\sum_{i=1}^{n-1} m(i) \leqq \sum_{j=0}^{\infty} 3n/2^j \leqq 6n$, and MINSPAN with queue implementation (a) and a cleanup after every stage runs in $O(n)$ time on planar graphs. This observation is due to Yao [21].

Now suppose MINSPAN is implemented using queue implementation (c), smallest tree selection, and with a cleanup before the $[\log a(j)]$th stage for $j = 1, \cdots$, where $a(j)$ is recursively defined as follows:

$$a(1) = \left\lceil \max\left\{(\log n)^2, \frac{m}{n \log n}\right\}\right\rceil, \qquad a(j+1) = \left\lceil \frac{a(j)^2}{\log n}\right\rceil,$$

where $\lceil x \rceil$ denotes the smallest integer not less than x. (That is, the jth cleanup is performed when the smallest tree remaining contains at least $a(j)$ vertices, if we ignore the shrinking caused by the cleanups.)

We can bound $a(j)$ from below as follows:

$$a(j) \geqq (\log n)^{f(j)},$$

where $f(j+1) = 2f(j) - 1$, hence $f(j) - 1 = 2^{j-1}(f(1) - 1)$. Thus if C is the total number of cleanups, $C > 0$ implies

$$\frac{n}{2} \geqq (\log n)^{f(C)} = (\log n)^{2^{C-1}(f(1)-1)+1},$$

which means C is

$$O\left(\log\left(\frac{\log n}{(\log\log n)(f(1)-1)}\right)\right).$$

By definition $f(1)\geqq 2$, so C is $O(\log\log n)$. Also, if $(\log n)^2 \leqq m/(n\log n)$, i.e., $n(\log n)^3 \leqq m$, then

$$f(1)\geqq \frac{\log\left(\frac{m}{n\log n}\right)}{\log\log n}$$

and C is

$$O\left(\log\left(\frac{\log n}{\log\left(\frac{m}{n\log n}\right)}\right)\right).$$

The time required for all the cleanups is $O(mC)$ (including re-initialization time). The rest of the time required by MINSPAN is dominated by the time spent in MIN. The time spent in MIN is

$$O\left(\sum_{i=1}^{n-1}(l(i)+h(i))\log\left(\frac{m(i)}{l(i)+h(i)}+1\right)\right),$$

where $m(i)$ is the number of edges in queue i when MIN(i) is executed, $l(i)$ is the number of edges deleted from queue i by execution of MIN(i), and $h(i)$ is the number of queues merged into queue i after the last cleanup which occurs before execution of MIN(i). The total time spent in MIN is $O(m\log\log n)$ by § 4. If the graph is dense, we can get a better bound.

Suppose $m\geqq n(\log n)^3$. We bound $\sum_{i=1}^{n}(l(i)+h(i))$ by dividing this sum among the cleanups. Consider a tree T (in the original graph) corresponding to a queue i_0 such that MIN(i_0) is executed between the jth and $(j+1)$st cleanups, and such that T is part of no larger tree corresponding to a queue on which a MIN operation is performed between the jth and $(j+1)$st cleanups. We call such a tree T a *j-tree*. Suppose T is formed from b trees existing at the jth cleanup, using a sequence of $b-1$ MIN operations, say MIN(i_1), $\cdots$, MIN(i_{b-1}).

Then $\sum_{k=0}^{b-1} l(i_k)\leqq b(b-1)+1$, since after the jth cleanup each of the b trees making up T has at most *one* edge incident with each of the other $b-1$ trees. Also, $\sum_{k=0}^{b-1} h(i_k)\leqq b$. Each of the b trees making up T has at least $a(j)$ vertices, and T contains fewer than $a(j+1)$ vertices, so $b\leqq |T|/(a(j))\leqq a(j+1)/(a(j))$.

Let $T_1,\cdots,T_d$ be all the j-trees. Then $\sum\{l(i)+h(i)|\text{MIN}(i)$ executed between jth, $(j+1)$st cleanups$\}$ is

$$O\left(\sum_{k=1}^{d}\frac{|T_k|^2}{a(j)^2}\right)=O\left(\frac{a(j+1)}{a(j)^2}\sum_{k=1}^{d}|T_k|\right)=O\left(\frac{n}{\log n}\right).$$

A similar argument shows that $\sum\{l(i)+h(i)|\text{MIN}(i)$ executed before first cleanup$\}$ is $O(a(1)\cdot n)=O(m/(\log n))$.

Thus $\sum_{i=1}^{n-1}[l(i)+h(i)] = O(m/(\log n))$ and the time spent in MIN is $O(m)$ if $m > n(\log n)^3$. The total time for this implementation is thus

$$O(mC) = O\left(m \log\left(\frac{\log n}{\log\left(\frac{m}{n \log n}\right)}\right)\right)$$

if $m > n(\log n)^3$. If $m \geqq cn^{1+\varepsilon}$ for some positive constants c and ε, this version of the general algorithm runs in $O(m \log (1/\varepsilon))$ time, and it always runs as fast as the version without cleanups (to within a constant factor).

6. Relationships with other problems. In this section we consider relationships between the minimum spanning tree problem and other problems, which might lead to a general lower bound. We show the following. 1. If the edges of the graph are presented in sorted order, the minimum spanning tree problem is equivalent (to within a constant factor) to a special type of disjoint set manipulation. 2. The worst case for finding a minimum spanning tree occurs when the problem graph is sparse, in particular when all vertices are degree three. 3. If the edges are not given in sorted order, finding a minimum spanning tree on certain graphs of m edges requires computing maximum elements for $\Omega(m)$ sets, each of size $\Omega(\log m)$. This leads to an $\Omega(m \log \log m)$ lower bound for a special class of minimum spanning tree algorithms. (Note: $\Omega(f(m))$ denotes a function which exceeds $cf(m)$ for some positive constant c and infinitely many m.)

Suppose the edges of the problem graph are presented in sorted order, smallest to largest. Consider a family of disjoint sets which partition the set $\{1, 2, \cdots, n\}$. Let the operation EQUIV(v, w) have the effect of combining the set containing v and the set containing w into a single set, and returning the value **true** if v and w were already in the same set, **false** otherwise.

Suppose $\{v_1, w_1\}, \cdots, \{v_m, w_m\}$ are the edges of the problem graph, in sorted order. If we begin with the singleton sets $\{1\}, \{2\}, \cdots, \{n\}$ and execute EQUIV(v_i, w_i) in order for each i, then the edges $\{v_i, w_i\}$ such that EQUIV(v_i, w_i) returns **false** give a minimum spanning tree (this is just an implementation of Kruskal's algorithm). Conversely, given a list of EQUIV(v_i, w_i) operations (each with a different ordered pair $\{v_i, w_i\}$), to be performed on the singleton sets $\{1\}$, $\{2\}, \cdots, \{n\}$, the (unique) minimum spanning tree of the graph with vertex set $\{1, 2, \cdots, n\}$ edge set $\{\{v_i, w_i\}\}$, and edge values $c(v_i, w_i) = i$ contains exactly the edges $\{v_i, w_i\}$ such that EQUIV(v_i, w_i) returns **false.**

Thus finding a minimum spanning tree if the edges are presented in order is equivalent to executing a list of EQUIV instructions. Using the algorithm described in [8] and [15], m EQUIV instructions can be executed in $O(m\alpha(m, n))$ time, where $\alpha(m, n)$ grows very slowly. A nonlinear lower bound on the time to execute a list of EQUIV instructions would give a nonlinear lower bound on the data-manipulation cost (as opposed to the comparison cost) of the minimum spanning tree problem.

To study the comparison cost of the minimum spanning tree problem, we need a simple definition of a comparison algorithm. For our purposes, an algorithm will be a comparison tree. Each vertex of the tree represents a comparison between the values of two edges in the problem graph. Depending

upon the outcome of the comparison, the next comparison to be made is either the left son or the right son of the previous comparison. We allow a different comparison tree for each possible graph. Given this model, we want to know the minimum number of comparisons $c(m)$ required to determine a minimum spanning tree for the worst-case graph G of no more than m edges.

We first show that, ignoring constant factors in running time, there are worst-case graphs which are sparse, in particular regular of degree three. Let G be any graph. Consider applying the following (nondeterministic) procedure to G.

```
procedure REGULARIZE(G);
    begin
        while G has a vertex v of degree ≦2 do
            if degree (v) = 0 then
                delete v;
            else if G has a vertex v of degree 1 do
                delete v and its incident edge;
            else if G has a vertex v of degree 2 do
                begin
                    let {v, w} be the minimum value edge incident to v;
                    delete {v, w} and collapse v and w into a single vertex;
                end;
            while G has a vertex v of degree ≧4 do
                begin
                    create a new vertex w;
                  for half of the edges {u, v} in G do
                        convert {u, v} to an edge {u, w};
                    create a new edge {v, w} of value less than that of all other
                    edges;
                end;
    end REGULARIZE;
```

Let G' be a graph produced when REGULARIZE is applied to a connected graph G having m edges. G' is connected, regular of degree three and has at most $3m$ edges. REGULARIZE can be implemented to run in $O(m)$ time (e.g., see [9]).

A spanning tree for G is characterized by its set of edges. The edges of a minimum spanning tree for G can be found from the edges of a minimum spanning tree for G' by

(i) adding all edges deleted from G by REGULARIZE;
(ii) deleting all edges added to G by REGULARIZE; and
(iii) restoring the original endpoints of each edge converted by REGULARIZE.

This process requires $O(m)$ time.

Let $t(m)$ be the minimum time required to find a minimum spanning tree of any connected graph with no more than m edges. Let $t_3(m)$ be the minimum time required to find a minimum spanning tree of any graph, regular of degree three, with no more than m edges. We desire a bound on $t(m)$ as a function of $t_3(m)$. It follows from the preceding paragraphs that $t(m)$ is $O(t_3(3m)+m)$. But we also know that $k_1 m \leqq t_3(m) \leqq k_2 m \log\log m$, so $t_3(3m)$ is $O(t_3(m))$, and $t(m)$ is

$O(t_3(m))$. Similarly $c(m)$ is $O(c_3(m))$, if $c_3(m)$ is the minimum number of *comparisons* required to find a minimum spanning tree in a graph regular of degree three with no more than m edges. (Note: $m-1 \leqq c_3(m)$ since finding a minimum spanning tree for a graph which is a cycle is equivalent to finding the maximum edge of the cycle.)

Now we derive a special case lower bound. We shall assume that the values of all the edges are distinct. This guarantees that the minimum spanning tree is unique. As a comparison-type algorithm proceeds, there will be certain edges known to be in the minimum spanning tree, called *included* edges, certain edges known not to be in the minimum spanning tree, called *excluded* edges, and other edges, called *unresolved* edges. The next two lemmas (which extend Lemma A) characterize the moment when an edge becomes resolved.

LEMMA 3. *An edge $\{v, w\}$ becomes included exactly when there is a set $X \subseteq V$ such that $v \in X$, $w \in V-X$, the most recent comparison shows $\{v, w\}$ to have minimum value in $\partial(X) = \{\{x, y\} | \{x, y\} \in E, x \in X \text{ and } y \in V-X\}$, and all edges in $\partial(X) - \{\{v, w\}\}$ are unresolved or excluded (just after the most recent comparison.*

Proof. Suppose $\{v, w\}$ is an edge such that a set X exists satisfying the hypotheses of the lemma. Let T be any spanning tree not containing $\{v, w\}$. Some cycle exists composed of edges of T and $\{v, w\}$. Some edge on this cycle other than $\{v, w\}$ is in $\partial(X)$. Deleting this edge from T and adding $\{v, w\}$ produces a new spanning tree of smaller total value. Thus T is not minimum, and $\{v, w\}$ must be in any minimum spanning tree.

Conversely, suppose that after some comparison, edge $\{v, w\}$ becomes included. Choose distinct edge values so that, subject to the constraints of the comparisons made so far, as many edges as possible have values smaller than $c(v, w)$. Let T be a minimum spanning tree in the resultant graph. Removal of $\{v, w\}$ from T breaks T into two parts. Let X consist of the vertices in one of these parts. Then X must satisfy the hypotheses of the lemma, since if the value of $\{v, w\}$ is greater than the value of some edge in $\partial(X)$, T can be modified to have smaller total value by deleting $\{v, w\}$ and adding an edge in $\partial(X)$. Furthermore, no edge in $\partial(X) - \{\{v, w\}\}$ can be among the included edges. □

LEMMA 4. *An edge $\{v, w\}$ becomes excluded exactly when there is a cycle containing $\{v, w\}$ and edges unresolved or included (just after the most recent comparison) such that the most recent comparison shows that $\{v, w\}$ is the maximum value edge on this cycle.*

Proof. The proof is analogous to that of Lemma 3. □

We would like to prove a worst-case lower bound of $\Omega(m \log \log m)$ comparisons for finding a minimum spanning tree in a graph with m edges. Here we show that this bound holds for a certain subclass of comparison algorithms. Consider only algorithms which obey the following rule:

Max: Any unresolved or included edge which is used in a comparison must be a possible maximum among unresolved and included edges.

The following "demon" (sometimes called an oracle or adversary in such proofs by other authors) generates a bad case for such a comparison algorithm:

(i) If two excluded edges are compared, the demon declares either as bigger.

(ii) If an excluded and a nonexcluded edge are compared, the demon declares the excluded one as bigger.

(iii) If two nonexcluded edges are compared, the demon declares the one with more edges known to be smaller as bigger.

LEMMA 5. *Suppose the demon above determines the results of comparisons for a comparison algorithm obeying rule* Max. *Then at all times during the comparison process, any nonexcluded edge known to be bigger than k edges must have been directly compared to at least* $\log k$ *such edges.*

Proof. Because of rule Max, rule (ii) of the demon and Lemma 4, no excluded edge is ever known to be smaller than any nonexcluded edge. Let $\{v, w\}$ be any nonexcluded edge. By rule Max and rule (iii) of the demon, any comparison which adds to the number of nonexcluded edges known to be smaller than $\{v, w\}$ must be a direct comparison with $\{v, w\}$ and can at most double the number of edges known to be less than $\{v, w\}$. The lemma follows. □

By appealing to a result of Tutte, we can use Lemma 5 to give the desired lower bound. The *girth* of a graph is the length of the shortest cycle in the graph.

LEMMA 6 (Tutte [18], p. 82]). *For all n, there is a graph of n vertices, with all vertices of degree three, having a girth which is* $\Omega(\log n)$.

THEOREM 1. *Any comparison algorithm obeying rule* Max *requires* $\Omega(m \log\log m)$ *comparisons for some graph regular of degree three having m edges.*

Proof. Suppose a comparison algorithm obeying rule Max is applied to one of the graphs given by Lemma 6. Any such graph has exactly $m = \frac{3}{2}n$ edges. Thus $\frac{1}{3}m + 1$ edges must be excluded before the minimum spanning tree is determined. For an edge to be excluded, it must be known to be a maximum over a cycle of nonexcluded edges (Lemma 4), but since any such cycle has length $\Omega(\log m)$, Lemma 5 implies that $\Omega(\log\log m)$ direct comparisons with the excluded edge are required. These $\Omega(\log\log m)$ comparisons are distinct for each excluded edge; thus a total of $\Omega(m \log\log m)$ comparisons are required. □

This proof of an $\Omega(m \log\log m)$ lower bound works only for a restricted class of algorithms. However, Lemma 4 and Lemma 6 together imply that any method which computes a minimum spanning tree for one of Tutte's graphs must compute the maximum element of $\Omega(m)$ sets, each of size $\Omega(\log m)$. Priority queue method (b) for implementing MINSPAN in fact works by computing minima over sets of size $\log m$. This step (of computing maxima or minima over sets of size $\log m$) seems to be the costly part of finding a minimum spanning tree, and if we could get a better understanding of the cycle structure in Tutte's graphs, we might be able to prove a general $\Omega(m \log\log m)$ lower bound for the minimum spanning tree problem.

7. Conclusions and conjectures. We have presented a general minimum spanning tree algorithm and studied its worst-case running time for various implementations and various types of graphs. We have given $O(m \log\log n)$ implementations for arbitrary graphs, an $O(n)$ implementation for planar graphs, and an $O(m)$ implementation for dense graphs (those for which $m > cn^{1+\varepsilon}$ for some positive constants c and ε). We believe the algorithms we have presented are not hard to implement and have constants of proportionality small enough to make them competitive in practice with older algorithms. (We have not yet performed any experiments to verify this conjecture.)

We have shown connections between the minimum spanning tree problem and a set manipulation problem and an ordering problem which may lead to

general nonlinear lower bounds. It is interesting to note that the problem of testing whether a given spanning tree is in fact minimum requires only $O(m\alpha(m, n))$ time [16]; thus testing a minimum spanning tree may be easier than finding a minimum spanning tree.

In addition to the problem of lower bounds, another area needs study; that of *average* running time. A "random" graph can be defined in several ways [6]; for any such definition, what is a minimum spanning tree algorithm with a good average running time? We can show that Kruskal's algorithm runs in $O(n+m)$ time on the average if the edges are presented in sorted order. We have devised an algorithm with a provable $O(n \log\log n + m)$ average running time (for edges not in sorted order) and a related algorithm which we believe has an $O(n+m)$ average running time. We also conjecture that MINSPAN, implemented without cleanups and with queues represented as leftist trees with delayed merge, has an $O(n+m)$ average running time. We hope to report on these ideas in a future paper.

Acknowledgment. The authors wish to thank Andrew Yao for providing stimulus and many helpful suggestions for this work.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] ———, *On finding lowest common ancestors in trees*, Proc. 5th Ann. ACM Symp. on Theory of Computing, 1973, pp. 253–265.

[3] C. BERGE AND A. GHOUILA-HOURI, *Programming, Grames, and Transportation Networks*, John Wiley, New York, 1965.

[4] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST AND R. TARJAN, *Time bounds for selection*, J. Comput. Systems Sci., 7 (1973), pp. 448–461.

[5] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[6] P. ERDÖS AND A. RÉNYI, *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Közl, 5 (1960), pp. 17–61.

[7] J. HOPCROFT AND R. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[8] J. HOPCROFT AND J. ULLMAN, *Set-merging algorithms*, this Journal, 2 (1973), pp. 294–303.

[9] J. HOPCROFT AND J. WONG, *Linear time algorithm for isomorphism of planar graphs*, Proc. 6th Ann. ACM Symp. on Theory of Computing, 1974, pp. 172–184.

[10] A. KERSCHENBAUM AND R. VAN SLYKE, *Computing minimum spanning trees efficiently*, Proc. 25th Ann. Conf. of the ACM, 1972, pp. 518–527.

[11] D. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[12] J. B. KRUSKAL, JR., *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 48–50.

[13] R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

[14] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[15] ———, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[16] ———, *Applications of path compression on balanced trees*, STAN-CS-75-512, Computer Sci. Dept., Stanford Univ., Stanford, Calif., 1975.

[17] ———, *Finding optimum branchings*, Networks, to appear.

[18] W. T. TUTTE, *Connectivity in Graphs*, Toronto University Press, Toronto, 1967.

[19] J. VUILLEMIN, private communication, 1975.

[20] A. C. YAO, *An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees*, Information Processing Letters, 4 (1975), pp. 21–23.
[21] ———, private communication, 1975.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

ANALYSIS OF THE OPTIMAL, LOOK-AHEAD DEMAND PAGING ALGORITHMS*

ALAN JAY SMITH†

Abstract. We express the future behavior of programs that may be described by two common program behavior models, the independent reference model and the LRU stack model, by a discrete time Markov chain. Using this Markov chain model, we are able to calculate the theoretical minimum number of page faults for a program representable by either of these models in either a fixed or variable size memory. The behavior of optimal look-ahead and optimal realizable demand paging algorithms are compared, and it is seen that look-ahead paging demonstrates an inherent advantage sufficient to account for the differences observed between currently implemented demand paging algorithms and theoretically optimal algorithms.

Key words. paging, virtual memory, memory management, LRU stack model, independent reference model, Markov chains, MIN algorithm

1. Introduction. A paged *virtual memory*, implemented as early as 1958 in the Atlas computer (Kilburn et al. (1962)), has become a common feature of large modern operating systems. Memory in such systems is divided into fixed size blocks called *page frames* and the address space of each process is divided into *pages*. The memory consists of at least two levels; a four or five level memory hierarchy including fast buffer storage (cache), main memory (core), drums, disks and tapes is common. If the pages are transferred from secondary storage to main storage only when needed, then the process is called *demand paging*. Every time an attempt is made to access a page which is not in main memory, a *page fault* is said to occur and this page is brought into memory. A page fault is costly in terms of lost processing time in two ways: the processor must execute the *page replacement algorithm* which removes a page from memory and thereby frees a page frame for the incoming page and it must execute the *page fetch algorithm* which finds the incoming page and initiates the transfer. The processor may also have to wait, if there is no other process ready to run, while that page is fetched. Effective operation of a computer system with virtual memory therefore requires that the amount of processor time wasted due to page faults be minimized.

The choice of a good page replacement algorithm is very important to minimizing the number of page faults. A poor replacement algorithm would choose pages likely to be needed again in the near future, thus precipitating further page faults. It can be shown that given complete knowledge of the future behavior of the program, the optimal paging algorithm for fixed memory size, MIN (the one that causes the minimum number of page faults), will remove that page among those in memory that will be referenced furthest in the future (Mattson et al. (1970), Pomeranz (1971)). Belady presents an algorithm for

* Received by the editors April 8, 1975, and in final revised form December 2, 1975.

† Computer Science Division, Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California at Berkeley, Berkeley, California 94720. This research was supported in part by the National Science Foundation under Grant DCR 74-18375, and in part by the University of California. Computer time was provided by the Energy Resources Development Administration under Contract AT(04-3)-515.

calculating this minimum number (Belady (1966)) in one pass over a program trace tape and he extends it to calculating the minimum number of faults for a number of different memory sizes at once in Belady and Palermo (1974). Mattson et al. (1970) present an algorithm for calculating the minimum number of faults (they call their algorithm OPT) in two passes over the trace tape. Lewis and Nelson (1974) also present algorithms for calculating this minimum number of faults. We show below that for certain models of program behavior it is possible to estimate analytically the number of faults generated by the MIN algorithm.

A number of paging algorithms have been proposed, such as working set (Denning (1968)) and page fault frequency (Chu and Opderbeck (1972)) that are designed to vary the number of page frames allocated to a process as its memory needs change. These algorithms assume that the computer system is multiprogrammed and that page frames may be transferred between processes. Prieve and Fabry (1975) present an algorithm which yields the optimal variable space memory allocation for a process when measured in virtual time. That is, this algorithm minimizes the number of page faults for any given average memory size when that average size is calculated during the period the process is active. In § 4 of this paper we show that it is also possible to estimate the behavior of this optimal algorithm, VMIN, analytically.

A model for program behavior that has been proposed for its convenience of analysis is the *independent reference model* (IRM) in which the probability of referencing page i at time t is p_i for all t. Baskett and Rafii (1975) have shown that the proper choice of the reference probabilities allows a number of program characteristics to be accurately captured by this model. Aho, Denning and Ullman (1971) demonstrate an algorithm A_0 which they prove is the optimal demand paging algorithm for the independent reference model. This algorithm keeps in memory the $M-1$ pages, for memory size M, with the largest values of p_i. The remaining page frame is used for other pages that are referenced. King (1971) calculates the fault rate for the independent reference model when using the least recently used (LRU), first in, first out (FIFO) or A_0 paging algorithms. Franaczek and Wagner (1974), Coffman and Denning (1973) and Denning and Schwartz (1972) also consider the independent reference model.

Another model for program behavior which has a number of useful features is the LRU *stack model*, which is discussed by Coffman and Denning (1973), Denning, Savage and Spirn (1972) and Oden and Shedler (1972). In this model, the sequence of LRU stack distances, $D=\{d_i\}$, (Mattson et al. (1970)) are a sequence of independent, identically distributed random variables. This model provides for locality of reference (Denning (1972)), but it fails to represent changes in the size of the locality or abrupt changes in the content of the locality. Lewis and Yue (1971) and Lewis and Shedler (1973) reject the LRU stack model using formal statistical techniques, but as we show in Fig. 5 (see § 5), there is reason to believe it to be a good approximation. If q_i is the probability of a hit to level i of the stack, and if $q_i \geqq q_j$ for $i \leqq j$, then it can be shown that LRU is the optimal demand paging algorithm (Coffman and Denning (1973)).

In the next two sections of this paper we show that it is possible to calculate analytically the behavior of the MIN algorithm when run on a trace of a program obeying either the LRU stack model or the independent reference model. In § 4

we analyze the behavior of the VMIN algorithm when applied to a program that can be described by either the independent reference model or the LRU stack model. Because MIN and VMIN minimize the fault rate for a specific reference string, they have an inherent advantage over realizable algorithms, even optimal ones such as A_0, which only minimizes the fault rate on the average over all IRM reference strings. Measurements and calculations to be presented in § 5 (some of which also appear in Baskett and Rafii (1975)) indicate that the observed difference between optimal (look-ahead) algorithms and realizable ones when run on real program traces is similar to the difference found when such a comparison is made on traces generated from independent reference model or LRU stack model simulations. In § 6, the conclusion, we indicate the applicability of our results.

2. The independent reference model. Let us consider a *program B* which consists of *M pages* using a memory of *N page frames* ($N<M$). The *reference string* $R=(r_1, r_2, \cdots, r_i, \cdots)$ is the sequence of memory references, listed in the order in which they occur. We need know only which page is referenced; thus r_i will denote a page number, $1 \leqq r_i \leqq M$. The independent reference model assumes that the reference string R has the following property: $P[r_i = j] = q_j$ for all i. We assume without loss of generality that $q_j > 0$, $1 \leqq j \leqq M$.

We define the *state S of the process B* as (F, W) where F is the (unordered) set of pages currently in memory and W is the set of all pages in the program B listed in the order of their first future occurrence. It may be seen that the reference string R is independent of the state of the memory F; thus the sequence of states $\langle W_i \rangle$ for the future behavior of the process is independent of the memory state also. Except where it will cause confusion, we will use the word "state" interchangeably to refer to the state of the entire system, (F, W), the future reference state W and the memory state F. We shall also omit super- and subscripts where it causes no confusion.

A future reference state W is denoted by a permutation of the numbers $1, \cdots, M$. The state $W_t = (w_1^t, w_2^t, \cdots, w_M^t)$ at time t is defined by: $w_1^t = r_{t+1}$, and for $j>1$, $w_j^t = \min_k (r_k,\ k>t$, such that $r_k \notin (w_1^t \cdots w_{j-1}^t))$. The steady state probability of state W is

$$P[W] = \prod_{k=1}^{M} \frac{q_{w_k}}{\sum_{j=k}^{M} q_{w_j}} \tag{1}$$

which may be shown (Mattson et al. (1970)) to be the same as the steady state probabilities for the LRU stack, as derived by King (1971) and by Coffman and Denning (1973). This formula may be quickly obtained by noting that the probability that $w_k = j$ is simply q_j, the probability that page j is referenced at an arbitrary time t, normalized by the "remaining reference probability" to be accounted for, $1 - \sum_{j=1}^{k-1} q_{w_j}$ or $\sum_{j=k}^{M} q_{w_j}$.

The set of possible successors $\{W_{t+1}\}$ to W_t, for $W_t = (w_1^t, \cdots, w_M^t)$, is

$$\begin{aligned}\{&(w_1^t, w_2^t, w_3^t, \cdots, w_M^t), (w_2^t, w_1^t, w_3^t, \cdots, w_M^t), (w_2^t, w_3^t, w_1^t, w_4^t, \cdots, w_M^t), \cdots \\ &(w_2^t, w_3^t, \cdots, w_1^t, w_M^t), (w_2^t, w_3^t, \cdots, w_M^t, w_1^t)\}.\end{aligned} \tag{2}$$

For all W_t,

$$P[W_{t+1} = W^* | W_t] = \begin{cases} \dfrac{P[W^*]}{\sum_{\forall W' \in \{W_{t+1}\}} P[W']} & \text{if } W^* \in \{W_{t+1}\}, \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

where $P[W^*]$ and $P[W']$ are defined in (1) and $\{W_{t+1}\}$ in (2). Equation (3) follows from the simple probability relation $P(A|B) = P(AB)/P(B)$ where A is the event that page w_1^t becomes the kth element in the future references state, B is the event that the remainder of the elements are ordered as $(w_2^t, w_3^t, \cdots, w_M^t)$ and AB is the occurrence of both of these events.

Let $S_t = (F_t, W_t)$ be the state of the process at the time of the tth reference. Define

$$p_{i,j} = p\{S_{t+1} = j | S_t = i\} \quad \text{for all } t; \tag{4}$$

that is, $p_{i,j}$ is the state transition probability. Let $p_{i,j}^w$ be the state transition probabilities for the future reference state, W, as defined in (3).

We define the *page fault rate*, PFR, as

$$\text{PFR} = \sum_{\forall S_i \text{ such that } w_1 \notin F_i} P[S_i]. \tag{5}$$

We show below that we can calculate the steady state page fault rate given the state transition probabilities.

The state transition probabilities depend on the paging algorithm. We define a paging algorithm as follows:

Case 1. $r_{t+1} \in F_t$. Then no page fault occurs and $F_{t+1} = F_t$.

Case 2. $r_{t+1} \notin F_t$. Then a page fault must occur. Let $F = (f_1, \cdots, f_N)$. Let $f_v = \max_k (w_k \text{ such that } \exists f_i = w_k)$. That is, f_v is that page in memory that will not be referenced for the longest time. The memory state changes as follows:

$$F_{t+1} = F_t - \{f_v^t\} + \{r_{t+1}\}. \tag{6}$$

The paging algorithm we have defined is identical to Belady's (1966) MIN algorithm, and it has the property that it is the demand paging algorithm that causes the minimum number of faults. We note also that our choice of a state space is such that we are able to specify the behavior of the system when the MIN paging algorithm is used; a paging algorithm such as LRU would require not the future state of the system but that the past state of the system be specified (King (1971)).

We may combine the transition information given above for the memory state F and the future reference state W to get the transition probabilities p_{ij} and matrix $P = \{p_{ij}\}$) for the state of the system.

$$p_{ij} = \begin{cases} p_{ij}^w & \text{if } w_1 \in F_i,\ F_j = F_i,\ W_j \in \{W_i'\} \text{ or} \\ & \text{if } w_1 \notin F_i,\ F_j = F_i - \{f_v\} + \{w_1\}, \\ & \text{and } W_j \in \{W_i'\}, \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

where f_v is defined in (6) and $\{W_i'\}$ is the set of successor future reference states to W_i. These state transition probabilities are immediate from the memory transition specification in (6) and the future reference state transition probabilities in (3).

Let $S_j = ((f_1 \cdots f_N), (w_1 \cdots w_M))$ and let S_i be some arbitrary other state. Then the following sequence of page references will always serve to take us from state S_i through state S_j:

$$f_1 \cdots f_N, f_1 \cdots f_N, w_1 \cdots w_M; \tag{8}$$

that is, we will be in state S_j immediately after the second reference to f_N in the string given above. We may also enter state S_j by the following reference string:

$$f_1 \cdots f_N, f_1 \cdots f_N f_N, w_1 \cdots w_M, \tag{9}$$

where we enter S_j after the third reference to f_N. From our definition of the transition probabilities in (7), it is evident that the state transitions depend only on the immediately prior state. It can thus be seen that we have defined an irreducible, nonperiodic, finite state Markov chain.

It was shown in the paragraph above that any state S_j is reachable from any other state S_i. The number of possible states then is the number of possible future states, times the number of possible memory states, or:

$$M! \cdot \binom{M}{N}. \tag{10}$$

The steady state probabilities may be found as the solution of

$$\pi = \pi P, \tag{11}$$

where π is the vector of steady state probabilities, i.e., $\pi_i = P\{S_i\}$. The elements of the matrix P were defined in (7). The calculation for the steady state probabilities thus involves the solution of a set of equations whose number grows as the expression in (10), which grows more quickly than factorially with the number of pages in the program.

3. The LRU stack model. We consider, as in the previous section, a program B which consists of M pages using a memory of N page frames ($N < M$). The reference string R is the sequence of page numbers in the order that program B references them. Let $D = (d_1, d_2, \cdots, d_i, \cdots)$ be the LRU distance string (Mattson et al. (1970)). That is, a distance of d_t at reference t (to page r_t) means that page r_t was, immediately prior to reference t, the d_tth most recently referenced page. The LRU stack model assumes that the distance string has the following property: $P[d_t = j] = q_j$ for all t. We assume $q_j > 0$, $j = 1, \cdots, M$.

The state S of the process B may be defined again as (F, W) where F is the unordered set of pages currently in memory and W is the set of all pages in program B, listed in the order of their first future occurrence. We note that the LRU stack model for program behavior is a probabilistic description that is independent of the specific identities of the pages. Coffman and Denning (1973) show, as might be expected, that the equilibrium probability of a reference to page i is $1/M$ for all i and that therefore the mean time between referencing a page is M.

We let the future reference state W_t be $(w_1^t \cdots w_M^t)$ as in § 2. The set of successors to W_t, $\{W_t'\} = \{W_{t+1}\}$, is as defined in (2). Mattson et al. (1970) show that each LRU stack hit at distance d_i (at time $t+u$) is also a future stack hit at distance d_i (at time t) where t and $t+u$ are the times of successive references to the given page, so the probability of a page that has just been referenced appearing next in the future stack at distance j is q_j. Thus for all W_t and all t.

$$P[W_{t+1} = (w_2^t \cdots w_j^t w_1^t \cdots w_M) | W_t = (w_1^t \cdots w_M^t)] = q_j. \tag{12}$$

It is also possible to obtain this result by direct calculation, as the interested reader may easily discover. Let $(w_2 \cdots w_j w_1 \cdots w_M)$ be defined as the jth successor of $(w_1 \cdots w_M)$.

It is possible to greatly reduce the state space for this process. Let $G_t = (g_1^t \cdots g_N^t)$ be the set of pages in main memory at time t identified by their position in the future stack. For example, if the future stack contents are $W_t = (4^*, 1, 5^*, 2^*, 6, 3)$, where $M = 6$, $N = 3$, and $F_t = (4, 5, 2)$, then $G_t = (1, 3, 4)$. The "*" has been used to denote those pages in main memory and thus the elements of G_t. The updating of both the future stack and the memory state is independent of the specific identities of the pages, so that this state space reduction has not eliminated any useful information.

We now let $S_t = (G_t)$ be the state of the process at the time t. Let p_{ij} be the state transition probabilities, which are defined as

$$p_{ij} = P[S_{t+1} = j | S_t = i] \quad \text{for all } t. \tag{13}$$

We define the page fault rate, PFR, as

$$\text{PFR} = \sum_{\forall S_i \text{ such that } 1 \notin G_i} P\{S_i\}. \tag{14}$$

The paging algorithm we will employ is defined as follows:

Case 1. $1 \in G_t$. Then no page fault occurs. Let the page r_t next appear at a distance of k in the future stack. Then $G_{t+1} = (g_1^{t+1}, g_2^{t+1}, \cdots, g_N^{t+1})$, where

$$g_i^{t+1} = \begin{cases} g_i^t & \text{if } k < g_i^t, \\ g_i^t - 1 & \text{if } k \geqq g_i^t \quad \text{and} \quad g_i^t \neq 1, \\ k & \text{if } g_i^t = 1. \end{cases}$$

In this case, let G_{t+1} be the kth successor of G_t.

Case 2. $1 \notin G_t$. A page fault occurs. Let r_t and k be as in Case 1. Let $j = \max_i (g_i^t)$. Let $G_t^0 = G_t - \{j\} + \{1\}$. Then the successors of state G_t are the same as the successors of state G_t^0 and occur with the same probabilities. The kth successor of G_t is defined as the kth successor of G_t^0.

We have, as in § 2, defined an algorithm that is identical to Belady's (1966) MIN algorithm; i.e., we have always chosen that page to remove from main memory that will not be referenced for the longest time.

The transition probabilities for the state G_t for all t are defined as:

$$p_{ij} = \begin{cases} q_k & \text{if } G_j \text{ is the } k\text{th successor of } G_i, \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

It was shown in § 2 that there exists a sequence of page references that will create any arbitrary state of pages in memory and any arbitrary future state. Since $q_k > 0$, for all k, it follows, therefore, that any state G_i is reachable from any state G_j in t or $t+1$ steps, for sufficiently large (but finite) t. By the same reasoning used earlier, we see that we have defined a discrete-time, finite state, nonperiodic, irreducible Markov chain. The equilibrium state probabilities, π_i, may be obtained as indicated in (11). The number of states is $\binom{M}{N}$, which is $M!/(N!(M-N)!)$. For a constant N, the size of the memory, the number of states grows factorially with the number of pages in the program, although not nearly as fast as the number of states in the independent reference model.

4. The VMIN algorithm.

4.1. The independent reference model. Let q_i be the probability of reference to page i, as defined in § 2. The time t between references to page i is distributed geometrically as

$$q_i(1-q_i)^{t-1} \tag{16}$$

with mean time between references

$$1/q_i. \tag{17}$$

The probability that the time between references is greater than τ (where τ is the working set parameter (Denning (1968)) is

$$(1-q_i)^{\tau}. \tag{18}$$

Because the time between references is geometrically distributed, the backward recurrence distance (Cox(1962)) is also geometrically distributed with the same mean. Therefore the probability that a given page i has not been referenced more recently than the preceding τ time units is also $(1-q_i)^{\tau}$ as in (18). This latter probability is just the probability that a reference to page i will result in a page fault when using the working set paging algorithm with working set parameter τ. Therefore the fault rate when using the working set algorithm on a program described by the independent reference model is

$$\sum_{i=1}^{M} q_i(1-q_i)^{\tau}. \tag{19}$$

The average memory space occupied for working set parameter τ is simply the sum of the probabilities, summed over the pages, that page i is in memory. Page i is in memory if it has been referenced in the preceding τ time units, so the average memory space used is

$$\sum_{i=1}^{M} (1-(1-q_i)^{\tau}). \tag{20}$$

The results above in this section have all been presented by Denning and Schwartz (1972) and Baskett and Rafii (1975).

The VMIN algorithm (Prieve and Fabry (1975)) is that algorithm that minimizes the number of page faults for given virtual (process) time average

memory use. It works as follows: for a fixed τ, remove all resident pages that will not be referenced in the following τ time units. By varying τ, a (piecewise linear) curve is traced out in the plane of page faults vs. average memory size. We note that since VMIN is a demand algorithm, it cannot behave symmetrically with working set. The time average probability that a page is in memory is the time average probability that the interreference interval is of length τ or less. This is the ratio of the probability that the interreference interval is $\leqq\tau$ times the mean length of such an interval to the mean length of all intervals. Thus the average memory size for parameter τ when using the VMIN algorithm is:

$$\begin{aligned}\sum_{i=1}^{M}\frac{(1-(1-q_i)^{\tau})\left(\sum_{t=1}^{\tau}\frac{q_i(1-q_i)^{t-1}t}{(1-(1-q_i)^{\tau})}\right)}{\sum_{t=1}^{\infty}q_i(1-q_i)^{t-1}t}&=\sum_{i=1}^{M}\frac{\sum_{t=1}^{\tau}q_i(1-q_i)^{t-1}t}{\sum_{t=1}^{\infty}q_i(1-q_i)^{t-1}t}\\&=\sum_{i=1}^{M}q_i^2\sum_{t=1}^{\tau}(1-q_i)^{t-1}t\\&=\sum_{i=1}^{M}q_i^2\left[\frac{1-\tau(1-q_i)^{\tau}}{q_i}+\frac{(1-q_i)(1-(1-q_i)^{\tau-1})}{(1-(1-q_i))^2}\right]\\&=M-\sum_{i=1}^{M}(1-q_i)^{\tau}(1+\tau q_i)\end{aligned}\tag{21}$$

(which, we observe, is less than for working set, as given in (20)). The page fault rate is given by (19); therefore we have calculated the expected fault rate and average memory allocation for the VMIN algorithm when using a given value of τ and when the program in question obeys the independent reference model.

4.2. The LRU stack model. Let q_i be the probability of a hit to level i in the stack, as defined in § 3. As discussed earlier (equation (12)), q_i is also the probability that the page just referenced will next be referenced when it is at level i in the stack. A page next referenced at level i will start at level 1 in the stack and successively occupy levels $1, 2, \cdots, i$ before it is referenced.

Let $\bar{Q}_i = \sum_{j=i+1}^{M} q_j$, and let $f_i(j)$ be the probability mass function (pmf) for the duration of time a page spends at level i in the stack, given that it is not referenced at level i in the stack. Then $f_i(j)$ is geometrically distributed, since all references are independent and follow the LRU stack model, and it is equal to

$$f_i(j)=\frac{\bar{Q}_i}{1-q_i}\left(1-\frac{\bar{Q}_i}{1-q_i}\right)^{j-1}.\tag{22}$$

This expression (equation (22)) holds even for $i=1$ if we define $0^0\equiv 1$.

Let $g_k(j)$ be the probability mass function for the time a page spends at level k in the stack, given that it will be referenced at level k. Then

$$g_k(j)=\frac{q_k}{1-\bar{Q}_k}\left(1-\frac{q_k}{1-\bar{Q}_k}\right)^{j-1}.\tag{23}$$

The probability mass function for the time until a page is referenced, given that it is next referenced at level k, is

$$h_k(j) = \left(* \prod_{i=1}^{k-1} f_i(j) \right) * g_k(j), \tag{24}$$

where " $*$ " denotes convolution and " $* \Pi$" means the convolution product. Summing over all k, we have as the pmf for the time between references to a page:

$$e(j) = \sum_{k=1}^{M} h_k(j) q_k \tag{25}$$

and for the cumulative function,

$$E(j) = \sum_{i=1}^{j} e(i). \tag{26}$$

The pmf for the time to the last reference to an arbitrary page at an arbitrary time is simply the backward recurrence time (Cox (1962, p. 61)) and is equal to

$$M(1-E(t)). \tag{27}$$

The probability that an arbitrary page is in core is simply the probability that the backward recurrence time is less than or equal to τ, the working set parameter; thus we have

$$M \sum_{t=1}^{\tau} (1-E(t)) \tag{28}$$

for the probability that an arbitrary page is in core. The mean number of pages in core is M times the value of (28) or

$$M^2 \sum_{t=1}^{\tau} (1-E(t)). \tag{29}$$

The fault probability for a parameter τ is

$$1-E(\tau), \tag{30}$$

the probability that a page has not been referenced in the preceding τ time units. Equations (29) and (30) trace out a fault rate/memory size curve that describes the behavior of the working set algorithm when executed on a program obeying the LRU stack model. There seems to be no simple closed form for these equations although the generating function for $e(t)$ is easily obtained. (In Denning et al. (1972) a recurrence relation is derived which permits computing $e(t)$ directly in t steps.)

As in (21), we can obtain an expression for the average memory usage for the VMIN algorithm when using parameter τ. The average memory space used is

$$M\frac{\sum_{t=1}^{\tau} e(t)t}{\sum_{t=1}^{\infty} e(t)t} = \sum_{t=1}^{\tau} e(t)t. \tag{31}$$

5. Calculations and simulations. In this section we present calculations of the MIN fault rate for two simple examples of the independent reference model and LRU stack model, and then we use simulation to compare the behavior of the algorithms discussed in this paper for two larger and more interesting cases. Exact numerical calculation appears limited, because of the combinatorial growth of the number of states, to systems with a dozen page frames or less.

Example 1. Consider a three page program obeying the LRU stack model, where $q_1 = q_2 = q_3 = \frac{1}{3}$. We note that a program described by the independent reference model with the same uniform values of $\{q_i\}$ will have exactly the same behavior. The state space (where the state G_t is given in the circles (see § 3)) and translation probabilities are diagrammed in Fig. 1, for a memory size of two page frames. The equilibrium state probabilities are: $p_{1,2} = \frac{4}{9}$, $p_{1,3} = \frac{1}{3}$ and $p_{2,3} = \frac{2}{9}$. The probability of a page fault occurring is then just $p_{2,3}$ or $\frac{2}{9}$. The probability of a page fault using any realizable demand paging algorithm is $\frac{1}{3}$, so that we observe that the optimal look-ahead algorithm has a 33% advantage over any realizable algorithm.

Example 2. We consider a program following either the independent reference model or the LRU stack model with $q_1 = q_2 = q_3 = q_4 = .25$. Let there be three page frames in primary memory. The state space and transition probabilities appear in Fig. 2. After calculation, we obtain as the steady state probabilities: $p_{1,2,3} = \frac{9}{22}$, $p_{1,2,4} = \frac{6}{22}$, $p_{1,3,4} = \frac{4}{22}$ and $p_{2,3,4} = \frac{3}{22}$. The fault probability is then $\frac{3}{22}$ (or .13636) which is much less than the fault rate of .25 to be expected from any realizable algorithm.

Because of the rapid growth in the number of states for the analysis of the MIN algorithm and the complexity of the analysis of the VMIN algorithm for the LRU stack model, and also because of the availability of an already written simulation program, we will present simulation results comparing these and other algorithms. We note that the complexity of our results does not negate their usefulness; rather it provides an incentive to develop a useful and simple approximation. In much the same manner that King's (1971) work provided the basis for

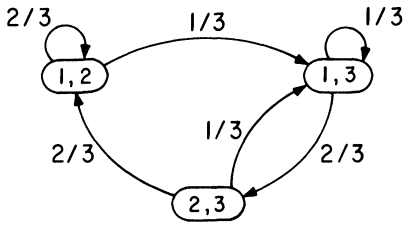


FIG. 1

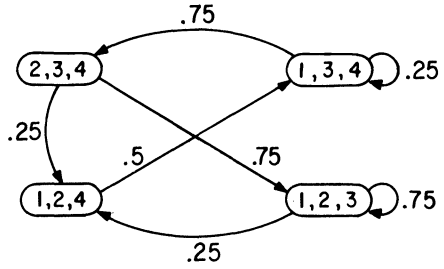


FIG. 2

some of the work leading to Baskett and Rafii's (1975) "A_0 inversion model", we believe that the existence of an exact solution to the fault behavior of nonrealizable algorithms will permit the development of more tractable approximations.

The values chosen for $\{q_i\}$ for our simulation of both the independent reference model and the LRU stack model are:

$$q_i = 2^{-i}, \qquad 1 \leqq i \leqq 13,$$
$$q_i = 2^{-13}, \qquad i = 14,$$
$$q_i = 0, \qquad i > 14.$$

The simulations were run for one million references each.

For the independent reference model, the A_0, MIN, VMIN, working set (WS) and LRU algorithms were employed, and the fault rate is shown in Fig. 3 for each of these algorithms. As might be expected, the A_0 algorithm demonstrated the best performance of any of the demand algorithms, although MIN, which is unrealizable, did better.

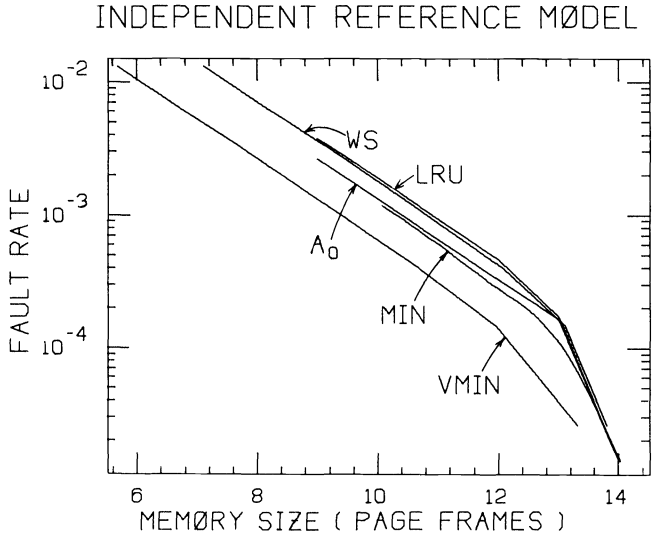


FIG. 3

The author conjectures that the observed result that working set performed better than LRU in this case is a general result for the independent reference model, although the author is not aware of any rigorous proof of this. (A "proof" by Denning and Schwarz (1972) was withdrawn by Denning (1973)). We do present the following heuristic argument; a complete proof (which we do not have) is, in any case, beyond the scope of this paper. We observe that, for the independent reference model, the reference pattern to any one page is completely independent of that to all other pages; thus each page may be studied in isolation for replacement. For both working set and LRU, a reference to a page constitutes a renewal epoch; that is, in both cases, replacement decisions for a given page are completely independent of the reference pattern preceding its last reference. Because of the independent reference patterns, the only measure of the expected time to the next reference to a page, given the renewal property described, is the *time* since the last reference, which is exactly what working set measures. LRU records the number of other pages referenced since the given one, which is a crude approximation to the time to last reference. Since the expected time to next reference is a monotonically increasing function of the time since last reference, the page to remove is one not referenced for some period τ in the past, which is precisely what working set does. Removal using LRU is somewhat more capricious than with working set, since whether a page is removed is a function of how many other pages have been referenced since it was last used. Working set thus uses the best possible estimator for the time to next reference among all algorithms with the given renewal property, and therefore is the best algorithm for the independent reference model among the class of such renewal algorithms. A_0 and Least Frequently Used, not belonging to this class of algorithms, are clearly superior.

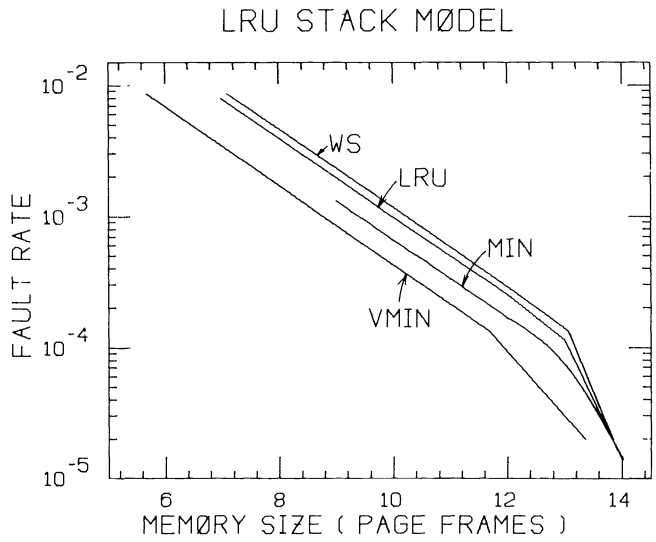


FIG. 4

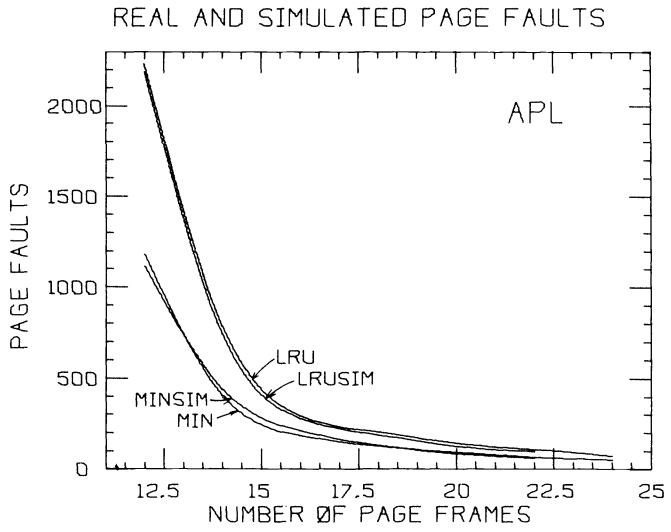


FIG. 5. (*From Baskett and Rafii* (1975))

We also observe, as expected, that VMIN, which varies the number of pages held in core according to the future behavior of the process, did the best. The reader should note that the vertical scale is logarithmic and thus vertical distances represent ratios.

In Fig. 4 we present the results of our simulation of the LRU stack model for the working set, LRU, MIN and VMIN algorithms. Our choice of $\{q_i\}$ convex and monotonically decreasing makes LRU the optimal realizable demand paging algorithm (Spirn (1973)), and, as expected, it performed better than working set. MIN as the optimal (unrealizable) algorithm can be seen to be significantly better (by a factor of about 35%), and VMIN is better than MIN by about the same amount.

For comparison with our simulation results, we show a comparison of simulated and measured results for the "APL" program, the execution of a program written in APL and described in detail in Smith (1974). This figure (Fig. 5) is taken from Baskett and Rafii (1975) and is reproduced with their permission.

The APL program was interpretively executed and a trace of the memory addresses accessed was produced. This trace was used to drive a trace driven simulation and the number of page faults generated by the trace was measured for both the MIN and the LRU algorithms. The observed set of LRU stack hit probabilities was then used to generate a reference string and the number of faults generated by the LRU and MIN algorithms was measured. The results are shown in Fig. 5.

Because MIN and VMIN are look-ahead algorithms, they have inherent advantages over optimal realizable algorithms, such as A_0 for the I.R.M. and LRU for the L.R.U.S.M. which can only minimize the fault rate on the average. This inherent advantage is apparent in our calculations, simulations and also in our trace driven simulation. Further, we note that the magnitude of this effect is approximately the same as the observed difference, in a number of studies,

between the results obtained with realizable and optimal algorithms. This is most clearly illustrated in Fig. 5 where the effect of MIN on a simulated reference string, generated from the LRU stack model, is almost identical to the effect on the real reference string.

6. Conclusions. We have demonstrated an algorithm for calculating the expected number of faults when a program is paged using either the MIN or VMIN replacement algorithm when that program can be described by either the LRU stack model or the independent reference model. This not only complements results by Denning and Schwartz (1972) and King (1971) on realizable algorithms, but it also demonstrates that it is possible to analyze look-ahead (nonrealizable) algorithms, certainly a nonobvious result. Although the complexity of our formulas makes direct calculation for most programs difficult or impossible, these exact results provide a basis from which a simple and effective approximation may be developed.

As noted in the last section, we have also shown that the difference in performance between realizable and nonrealizable algorithms on reference strings generated from our models of program behavior is very close to that seen on real program traces. This is all the more striking by being demonstrated analytically.

Acknowledgments. I'd like to thank Forest Baskett of Stanford University for pointing out the Denning, Savage and Spirn (1972) paper, and also to thank him and Abbas Rafii for permission to reproduce Fig. 5. Thanks are also due to Domenico Ferrari, Don Slutz and Ron Fagin for reading the manuscript and some helpful suggestions and to Ruth Suzuki for her excellent typing of this manuscript.

REFERENCES

A. V. AHO, P. J. DENNING AND J. D. ULLMAN (1971), *Principles of optimal page replacement*, J. Assoc. Comput. Mach., 18, p. 80–93.

F. BASKETT AND A. RAFII (1975), *Stochastic models of program paging behavior*, Rep., Stanford Univ. Computer Sci. Dept., Stanford, Calif., to appear.

L. A. BELADY (1966), *A study of replacement algorithms for a virtual storage computer*, IBM Systems J., 5, pp. 78–101.

L. A. BELADY AND F. P. PALERMO (1974), *On-line measurement of paging behavior by the multivalued MIN algorithm*, IBM J. Res. Develop., 18, pp. 2–19.

W. W. CHU AND H. OPDERBECK (1972), *The page fault frequency replacement algorithm*, Proceedings AFIPS 1972 Fall Joint Computer Conference, 46, no. 1, AFIPS Press, Montvale, N.J., pp. 597–609.

E. G. COFFMAN JR. AND P. J. DENNING (1973), *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N.J.

D. R. COX (1962), *Renewal Theory*, Methuen, London (U.S. distrib. Barnes and Noble, New York).

P. J. DENNING (1968), *The working set model for program behavior*, Comm. ACM, 11, pp. 323–333.

———, (1972), *On modeling program behavior*, Proceedings AFIPS 1972, Spring Joint Computer Conference, AFIPS Press, Montvale, N.J., pp. 937–944.

P. J. DENNING, J. E. SAVAGE AND J. R. SPIRN (1972), *Models for locality in program behavior*, Tech. Rep. 107, Computer Sci. Lab., Dept. of Electrical Engrg., Princeton Univ., Princeton, N.J.

P. J. DENNING AND S. C. SCHWARTZ (1972), *Properties of the working-set model*, Comm. ACM, 15, pp. 191–198.

P. J. DENNING (1973), *Corrigendum to Denning and Schwartz* (1972), Comm. ACM, 16, p. 122.

P. A. FRANACZEK AND T. J. WAGNER (1974), *Some distribution free aspects of paging algorithm performance*, J. Assoc. Comput. Mach., 21, pp. 31–39.

T. KILBURN, D. B. G. EDWARDS, M. J. LANIGAN AND F. H. SUMNER (1962), *One-level storage system*, IRE Trans. Elec. Comp., EC-11, pp. 223–235.

W. F. KING III (1971), *Analysis of demand paging algorithms*, Proc. IFIPS Conf., Ljubljana, Yugoslavia, pp. TA-3-155–TA-3-160.

C. H. LEWIS AND R. A. NELSON (1974), *Some one pass algorithms for the generation of OPT distance strings*, IBM Rep. RC 4758.

P. A. W. LEWIS AND P. C. YUE (1971), *Statistical analysis of program reference patterns in a paging environment*, IEEE Comp. Soc. Conf., Boston, Mass., pp. 153–154.

P. A. W. LEWIS AND G. S. SHEDLER (1973), *Empirically derived micromodels for sequences of page exceptions*, IBM J. Res. Develop., pp. 86–100.

R. L. MATTSON, J. GECSEI, D. R. SLUTZ AND I. L. TRAIGER (1970), *Evaluation techniques for storage hierarchies*, IBM Systems J., 9, pp. 78–117.

P. H. ODEN AND G. S. SHEDLER (1972), *A model of memory contention in a paging machine*, Comm. ACM, 15, pp. 761–771.

J. E. POMERANZ (1971), *Paging with fewest expected replacements*, Proc. IFIPS Conf., Ljubljana, Yugoslavia, pp. TA-3-160–TA-3-162.

B. G. PRIEVE AND R. S. FABRY (1975), *An optimal variable space page replacement algorithm*, Proc. Fifth SIGOPS Conf., Austin, Tex., Nov. 1975; Comm. ACM, 19 (1976), pp. 295–297.

A. J. SMITH (1974), *A modified working set paging algorithm*, Rep. STAN-CS-74-451, Stanford Univ. Computer Sci. Dept., Stanford, Calif.; IEEE Trans. Computers, to appear Sept., 1976.

J. R. SPIRN (1973), *Program locality and dynamic memory management*, Ph.D. thesis, Dept. of Electrical Engrg., Princeton Univ., Princeton, N.J.

SIAM J. COMPUT.
Vol. 5, No. 4, December 1976

ERRATA: ON OPTIMAL PROCESSOR SCHEDULING FOR MULTIPROGRAMMING*

L. J. BASS†

Modify the definition of processor bound on p. 278 as follows: We say two programs P_1 and P_2 are *processor bound with respect to each other* if

$$T_{1,i} \geqq \sum_{j=1}^{n_2-1} t_{2,j} \quad \text{for } 1 < i \leqq n_1$$

and

$$T_{2,i} \geqq \sum_{j=1}^{n_2-1} t_{1,j} \quad \text{for } 1 < i \leqq n_2,$$

and either

$$T_{1,1} \geqq t_{2,1} \quad \text{or} \quad T_{2,1} \geqq t_{1,1}$$

Intuitively, this definition says that, with the possible exception of one of the initial compute times, all of the compute times of P_1 are greater than the sum of the wait times of P_2, and vice versa.

Modify the statement of Theorem 2 on pp. 278–279 as follows:

THEOREM 2. *Let P_1 and P_2 be processor bound with respect to each other. Then*

$$R(P_1, P_2) = \sum_{i=1,2} \sum_{j \leqq n_i} T_{i,j}.$$

Furthermore, if $T_{1,j} - t_{2,1} \geqq T_{2,1} - t_{1,1}$, then

$$R_2(P_1, P_2) = R(P_1, P_2).$$

The proof is as previously stated.

I am grateful to Eike Riedemann of the Universität Dortmund for bringing this error to my attention.

* This Journal, 2 (1973), pp. 273–280. Received by the editors October 21, 1975.

† Department of Computer Science and Experimental Statistics, University of Rhode Island, Kingston, Rhode Island 02881.