stichting
mathematisch
centrum

MC

AFDELING MATHEMATISCHE BESLISKUNDE (DEPARTMENT OF OPERATIONS RESEARCH)

BW 106/79

MEI

E.L. LAWLER

EFFICIENT IMPLEMENTATION OF DYNAMIC PROGRAMMING ALGORITHMS FOR SEQUENCING PROBLEMS

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS subject classification scheme (1979): 90B35, 90C39, 68C05, 68C15, 68C25

EFFICIENT IMPLEMENTATION OF DYNAMIC PROGRAMMING ALGORITHMS FOR SEQUENCING PROBLEMS

E.L. LAWLER

Computer Science Division, University of California, Berkeley, CA 94720, USA

ABSTRACT

Dynamic programming has proved to be a fruitful approach for the solution of a variety of sequencing problems, including single-machine sequencing problems and assembly line balancing problems. However, certain technical difficulties have been experienced in fully exploiting the reduction in computational complexity which should result from the existence of precedence constraints in these problems. In particular, no completely satisfactory technique has existed for generating "feasible" sets of jobs, as determined by precedence constraints. In this paper we present a simple computer implementation of the dynamic programming algorithms which overcomes these difficulties. The implementation permits sequencing problems with precedence constraints to be solved in $O(Kn)$ time, where K is the number of feasible sets, and in $O(n+k_{max})$ space, where k_m is the number of feasible sets of size m and $k_{max} = \max_m\{k_m\}$. It is also shown how similar techniques can be applied to improve the efficiency of dynamic programming computations for the traveling salesman problem, when the network is sparse or when there are precedence constraints.

KEY WORDS & PHRASES: *dynamic programming, sequencing problems, assembly-line balancing, linear arrangement, traveling salesman problem, precedence constraints, computer implementation, data structures, computational complexity.*

NOTE: This report is not for review; it will be submitted for publication in a journal.

1. INTRODUCTION

Dynamic programming has proved to be a fruitful approach for the solution of a variety of sequencing problems [4]. However, certain technical difficulties have been encountered by BAKER and SCHRAGE [2], HELD, KARP and SHARESHIAN [6], and others, in implementing the dynamic programming algorithms when the sequencing problems are to be solved subject to precedence constraints. In this paper we present a simple and efficient computer implementation which overcomes there technical difficulties.

The general type of sequencing problem we are concerned with is to find a permutation π of the n elements of a set N such that the value of a specified objective function $f(\pi)$ is minimized. When *precedence constraints* are specified by an acyclic digraph $G = (N,A)$, the minimization is to be carried out over *admissible* permutations, where a permutation π is admissible only if $(i,j) \in A$ implies that i precedes j in π.

It is well known that a number of important special cases of this general problem can be solved by dynamic programming. We are concerned with those cases in which dynamic programming calls for the computation of F(N) by recursion over subsets $S \subseteq N$, subject to equations of the general form

$$F(S) = \min_{j \in S} \{g(F(S-j),S,j)\}, \tag{1.1}$$

$$F(\emptyset) = 0,$$

where g is a readily computed function derived from the objective function f. (Here, and below, we let S+j denote $S \cup \{j\}$ and S-j denote $S-\{j\}$.)

For example, suppose n jobs are to be sequenced for processing by a single machine. For each job j, $j = 1,2,\ldots,n$, there is a specified processing time $p_j \geq 0$ and a penalty function f_j. A given permutation π induces a completion time C_j^{π} for each job j. The objective is to find an admissible permutation π which minimizes

$$f(\pi) = \sum_{j=1}^{n} f_j(C_j^{\pi}).$$

For this problem,

$$g(F(S-j),S,j) = F(S-j) + f_j(\sum_{k \in S} p_k),$$

and F(S) is the cost of an optimal sequence for the jobs in S, where the first job starts at time t = 0 [5,7].

A similar treatment can be given to problems in which

$$f(\pi) = \max_j\{f_j(C_j^{\pi})\},$$

by letting

$$g(F(S-j),S,j) = \max\{F(S-j), \quad f_j(\sum_{k\in S} p_k)\}.$$

This approach is of no value, if the functions f_j are monotone nondecreasing, since such problems can be solved much more efficiently by other means [8]. However, dynamic programming can be worthwhile for related minmax problems. For example, whereas the "cumulative cost" or "initial resource requirement" problems can be efficiently solved for series parallel precedence constraints [1,11,12], these same problems are NP-hard for arbitrary precedence constraints and dynamic programming may then be worthwhile.

Another problem admitting of solution by recursion equations of the general form (1.1) is the linear arrangement or one-dimensional module placement problem [9]. For this problem,

$$g(F(S-j),S,j) = F(S-j) + c(S,N-S),$$

where c(S,N-S) is the capacity of cutset (S,N-S) in a specified graph. The bin packing, one dimensional stock cutting and assembly line balancing problems also yield to this approach [5,6]. For these problems,

$$g(F(S-j),S,j) = F(S-j) + \Delta(F(S-j),p_j),$$

where p_j is a specified parameter and Δ is a certain readily computed function.

Let us consider the running time required to solve equations (1.1) for F(N), subject to the assumption that g can be computed in constant time. In the absence of precedence constraints, there is an equation for each subset $S \subseteq N$ and F(N) can be computed in time proportional to

$$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1},$$

or $O(n2^n)$ time. Although this time bound is exponential, it does represent very much less running time than would be required for an exhaustive examination of n! permutations.

When precedence constraints are added, the computational complexity is in principle reduced. Let us call a set $S \subseteq N$ *feasible* if $j \in S$ implies that all predecessors of j in the digraph G = (N,A) are contained in S. It is well known that equations (1.1) need be solved only for feasible sets S, with the minimization in each equation only over $j \in S$ such that S-j is also feasible. It follows that the time required to compute F(N) should be bounded by O(Kn), where K is the number of feasible sets. Moreover, it has been shown that in practice K is often very much smaller than 2^n [2].

However, there are a number of questions of implementation that need to be resolved, in order to take complete advantage of the reduction in complexity that should result from precedence constraints. In particular, one must have an efficient procedure for generating the K feasible sets and, for a given feasible set S, identifying and locating in memory the feasible sets S-j.

Various schemes have been proposed for generating and addressing feasible sets. BAKER and SCHRAGE [2], for example, proposed an approach whereby each element j is assigned an integer label a_j. Each feasible set S is then assigned an index equal to the sum of the labels of the elements contained within it. Ideally, these indices should provide a one-one mapping of the feasible sets onto the integers 0,1,...,K-1, so that no space is wasted when these indices are used to determine memory addresses. It remains an open question whether a labelling with this property exists for all precedence digraphs, and it appears that the labeling procedure proposed in [2] does waste an indefinite amount of space for some digraphs. In the opinion of the present author, the labeling approach is inherently more cumbersome and time consuming than is necessary to effect an efficient implementation of the dynamic programming algorithms.

In this paper we present a simple and efficient computer implementation whereby the dynamic programming computations can be carried out in O(Kn) time, and more significantly, in only $O(n+k_{max})$ space, where k_m is the number of feasible sets of size m and $k_{max} = \max_m\{k_m\}$. Not only does this implementation overcome the difficulties encountered in [2,6], but it provides

an answer to the observation in [2] that "There are a number of algorithms in the literature for generating subsets ... However, we are unaware of any efficient algorithms for generating subsets subject to precedence restrictions".

In the final section of this paper we indicate how the implementation can be extended to solve the traveling salesman problem. The approach presented may be especially useful in cases where the network over which the problem to be solved is sparse, or where precedence constraints exist.

2. GENERATING FEASIBLE SETS

We assume that we are dealing with a computer with binary word length at least n. This is actually not a very restrictive assumption, and it is one which appears to have been made, at least implicitly, by previous authors. Note that at least $\log_2 K$ bits are required to address K feasible sets. Thus, if $K = \frac{1}{1024}\, 2^n$, a word length of n-10 is required to store a single address. From a strictly theoretical point of view, our assumption enables us to obtain time and space bounds that are smaller by a factor of n than those which would otherwise be obtained.

We shall represent the precedence digraph G = (N,A) by its adjacency matrix and store each column of the matrix in a separate word. Thus, column 4 of the adjacency matrix of the digraph pictured in Figure 1 is stored as:

$$\begin{matrix}(1 & 1 & 0 & 0 & 0 & 0). \\ 1 & 2 & 3 & 4 & 5 & 6\end{matrix} \qquad (2.1)$$

We shall represent a set S by its incidence vector on N and store this vector in a single word. The incidence vector for S has numerical value

$$n(S) = \sum_{j \in S} 2^{n-j}.$$

Thus the set $S = \{1,2,3\} \subseteq \{1,2,\ldots,6\}$ is represented by the vector

$$\begin{matrix}(1 & 1 & 1 & 0 & 0 & 0), \\ 1 & 2 & 3 & 4 & 5 & 6\end{matrix} \qquad (2.2)$$

and this vector has value 56.

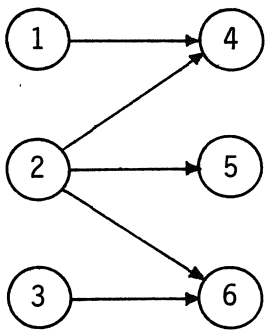


Figure 1. Digraph for example.

We assert that each feasible set of size m+1 is of the form S+j, where S is a feasible set of size m, $j \notin S$, and there is no arc (i,j) in G such that $i \notin S$. Given a set S of size m, it is easy to verify whether S+j satisfies these conditions. First check that element j of the incidence vector for S is zero. Then compare the incidence vector for S with column j of the adjacency matrix for G, to see if the incidence vector for S has a one in each position that column j has a one. Thus, for example, comparison of the vectors (2.1) and (2.2) shows that {1,2,3,4} is a feasible set for the digraph in Figure 1.

Exactly how these tests for S+j should be made depends upon the machine language instructions available in the computer we are dealing with. In any case, these tests can be assumed to be carried out, and the incidence vector for S+j formed, in constant time.

We shall generate feasible sets in order of size, beginning with $\emptyset$, then generating feasible sets of size one, size two, ..., and ending with the set N. Suppose that there are k_m feasible sets of size m, and that these are S_i, $i = 1,2,\ldots,k_m$, where $n(S_i) < n(S_{i+1})$, $i = 1,2,\ldots,k_m-1$. The k_{m+1} feasible sets of size m+1 are generated as follows.

First make one list of all feasible sets of the form S_i+1 and a second list of all feasible sets of the form S_i+2, with both lists in increasing order of the index i, and therefore in increasing numerical order for the sets contained in them. Then merge the two lists, *eliminating duplicate entries in the course of the merge*. Each of the two lists merged contains no more than k_{m+1} entries, and the list resulting from the merge also contains no more than k_{m+1} entries. Next generate a list of feasible sets of the form S_i+3 and merge it (again eliminating duplicate entries) with the list previously obtained. Continue in this way until the list S_i+n has been merged. The final list contains the k_{m+1} feasible sets of size m+1.

As an example, consider the digraph depicted in Figure 1. There are five feasible sets of size three, and from these we obtain the list indicated below (dispensing with brackets and commas in the representation of sets):

S_i	S_i+1	S_i+2	S_i+3	S_i+4	S_i+5	S_i+6
123				1234	1235	1236
124			1234		1245	
125			1235	1245		
235	1235					2356
236	1236				2356	

Let us ignore the list for S_i+2 since it is empty. Merging lists S_i+1 and S_{i+3}, we obtain a list containing 1234, 1235, 1236, a duplicate entry for 1235 being discarded, merging this list with S_i+4, we obtain 1234, 1235, 1236, 1245, eliminating a duplicate entry for 1234. Merging this list with S_i+5, we obtain 1234, 1235, 1236, 1245, 2356, eliminating duplicate entries for 1235 and 1245. Merging with S_i+6, we obtain 1234, 1235, 1236, 1245, 2356, eliminating duplicate entries for 1236, 2356. The final list contains all five feasible sets of size four.

Each list containing feasible sets of the form S_i+j can be obtained in $O(k_m)$ time and space, exclusive of space required to store the adjacency matrix of G. Hence the generation of all n lists can be carried out in $O(nk_m)$ time and $O(k_m)$ space. Each list entering a merge contains no more than k_{m+1} entries. Hence each merge can be carried out with no more than $2k_{m+1}-1$ numerical comparisons and in $O(k_{m+1})$ time and space. Hence all n-1 merges can be carried out in $O(nk_{m+1})$ time and $O(k_{m+1})$ space. Taking account of time and space requirements for both list generation and merging, the list of all feasible sets of size m+1 can be obtained in $O(n\max\{k_m,k_{m+1}\})$ time and $O(\max\{k_m,k_{m+1}\})$ space.

It follows from the above that the time required to generate all feasible sets is bounded by $O(Kn)$. If feasible sets of size m are outputted as soon as those of size m+1 have been obtained, the space required to generate all feasible sets is bounded by $O(n+k_{max})$, where $O(n)$ space is required to store the adjacency matrix of G. Otherwise space is bounded by $O(K)$.

The key idea which has enabled us to attain these time and space bounds is the generation of lists of the form S_i+j and then merging them, with elimination of duplicate entries. This is precisely the same technique employed by the present author to attain small time and space bounds for fast approximation algorithms for knapsack problems [10].

3. SOLVING THE RECURRENCE EQUATIONS

It is a simple matter to compute the values F(S) in the course of generating the feasible sets, as follows.

Suppose with each set S_i, $i = 1,2,\ldots,k_m$, there is recorded the value $F(S_i)$. When forming the list of feasible sets of the form S_i+j, make a single entry in the list for each feasible set S_i+j consisting of its incidence vector (in one word) and the computed value $g(F(S_i),S_i+j,j)$ (in a second word). In the course of merging lists, whenever a given set is found to be duplicated, retain the list entry with the smaller g-value. When all lists have been merged, the g-value recorded for each (m+1)-element set S in the final list is the value of F(S), as determined by (1.1).

If we seek only to compute the value F(N), then it is possible to discard all feasible sets of size m (and their computed F-values) as soon as those of size m+1 have been obtained. The entire computation can thus be carried out in $O(Kn)$ time and $O(n+k_{max})$ space. However, if we wish to construct an optimal sequence π for which $f(\pi) = F(N)$, then it is necessary to elaborate the procedure a bit, especially if we are to attain the same time and space bounds.

4. CONSTRUCTING AN OPTIMAL SEQUENCE

The most straightforward way to construct an optimal sequence is simply to record with each feasible set S a sequence $\pi(S)$ yielding the value $F(S)$.

Suppose with each set S_i, $i = 1,2,\ldots,k_m$, there is recorded a sequence $\pi(S_i)$ yielding the value $F(S_i)$. When forming the list of feasible sets of the form S_i+j, the sequence $\pi(S_i),j$ (sequence $\pi(S_i)$, followed by j) is made part of the list entry for S_i+j (in addition to the incidence vector for S_i+j and the value $g(F(S_i),S_i+j,j)$). When all lists have been merged, as described in the previous section, the desired sequence $\pi(S)$ is recorded in the entry for each (m+1)-element set S in the final list.

The difficulty with this straightforward approach is the amount of space required to store each $\pi(S)$. Since $\lceil \log_2 n \rceil$ words of length n are required to store a permutation of n elements, the space bound is increased to $O(n+k_{max}\log n)$. Moreover, the time bound is increased to $O(Kn\log n)$, because of the time required to record sequences in list entries and the extra time required to merge lists with the larger entries.

The time bound is easily restored to $O(Kn)$ by using pointers. Instead of recording the sequence $\pi(S_i),j$ in the entry for S_i+j, record the index j and a pointer to the sequence $\pi(S_i)$. Then when the list of (m+1)-element feasible sets has been obtained, use the index and the pointer recorded in the entry for each set S to obtain the sequence $\pi(S)$.

It is possible to obtain a space bound of $O(n+k_{max})$, while maintaining the time bound at $O(Kn)$. However, we admit that the "divide and conquer" technique we shall present may be of more theoretical than practical significance.

Virtually all of the problems for which recursion equations of the form (1.1) have been formulated can be solved equally well by constructing an optimal sequence in one direction or the other, i.e. first-to-last or last-to-first. For example, in the case of the single-machine problem discussed in Section 1, let

$$\bar{F}(S) = \max_{j\in S}\{\bar{F}(S-j) + f_j(P-\textstyle\sum_{k\in S-j} p_k)\}, \tag{4.1}$$

with $\bar{F}(\emptyset) = 0$, where

$$P = \sum_{j=1}^{n} p_j.$$

If equations (4.1) are solved for sets S,S-j which are feasible with respect to the precedence digraph $\bar{G}(N,\bar{A})$ obtained by reversing the directions of all arcs in G = (N,A), then $\bar{F}(S)$ is the cost of an optimal sequence for the jobs in S, with the completion of the last job at time P, and the other jobs immediately preceding.

Let S_m ($\bar{S}_m$) denote the family of sets of size m which are feasible with request to G ($\bar{G}$). Note that $S \in S_m$ if and only if $N-S \in \bar{S}_{n-m}$, hence $k_m = |S_m| = |S_{n-m}|$. It should be clear that, for any m,

$$F(N) = \min\{F(S)+\bar{F}(N-S) \mid S \in S_m\}. \tag{4.2}$$

Assume, for simplicity, that n is a power of two. Taking m = n/2, it is a simple matter to compute F(N) by (4.2). Generate a list for $S_{n/2}$, in increasing numerical order of the sets contained within it, and a list for $\bar{S}_{n/2}$, in decreasing numerical order. Form the sums indicated in (4.2) and choose the smallest. This can be done in O(Kn) time and $O(n+k_{max})$ space.

Let S^* be a set in $S_{n/2}$ yielding a minimum value in (4.2). There is an optimal sequence in which the jobs in S^* precede the jobs in $N-S^*$. We now, in effect, have two problems, each with n/2 jobs. We can now apply equation (4.2) with respect to each of these subproblems to obtain four problems each with n/4 jobs, and so forth.

The time required to solve the first subproblem of size n/2 is proportional to $(n/2)K_1$ and the time required to solve the second subproblem is proportional to $(n/2)K_2$ where $K_1+K_2 \leq K+1$. (K_1+K_2 can be expected to be considerably less than K in practice). It follows that the time required to construct an optimal sequence by this method is bounded by a function of order

$$Kn + (K+1)\frac{n}{2} + (K+3)\frac{n}{4} + \ldots + (K-n+1),$$

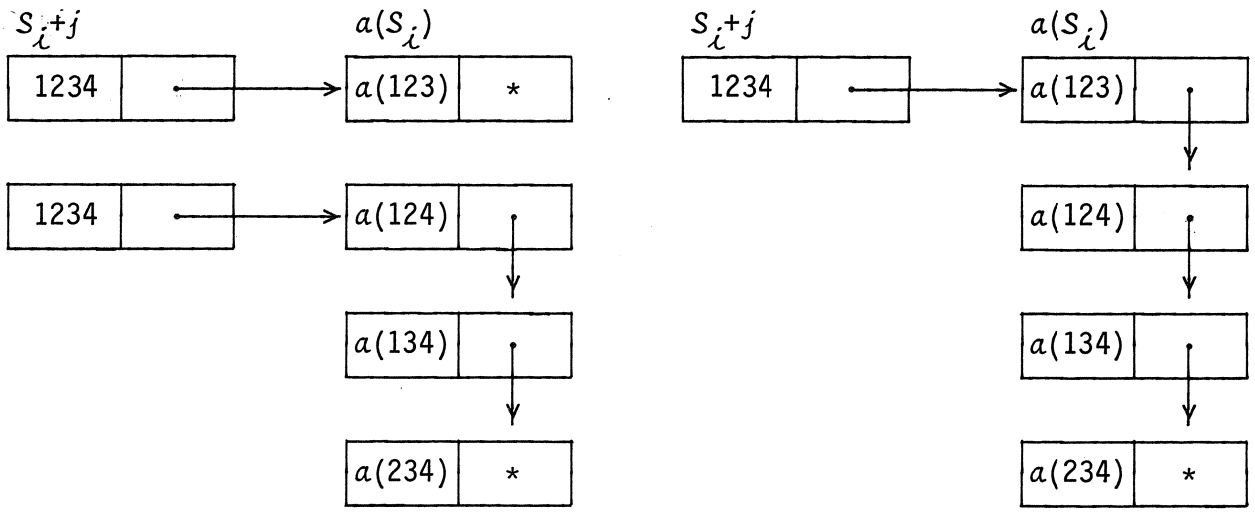
which is, of course, O(Kn).

It is not hard to verify that if this approach is properly implemented, space requirements are bounded by $O(n+k_{max})$.

5. PROVIDING FOR REPETITIVE COMPUTATIONS

There may be situations in which it is desired to carry out computations for a large number of different objective functions, but with respect to the same precedence digraph in each case. It may then be desirable to generate the feasible sets once only, and to record the feasible sets in memory in such a way that repetitive computations can be carried out as quickly as possible.

As soon as feasible sets of size m+1 have been generated, we propose to assign them to consecutive memory addresses, immediately following those for sets of size m. With each feasible set S we shall store a pointer to a list of the addresses a(S-j) of feasible sets S-j. This data structure is indicated in Figure 2.



(a) Data structure before elimination of duplicate list entry.

(b) Data structure after elimination of duplicate list entry.

Figure 2.

This data structure is generated by a simple modification of the procedure presented in Section 2. When the list of feasible sets of the form S_i+j is generated, each entry in the list is provided with a pointer to a record containing the address $a(S_i)$ and a special symbol *, as shown in Figure 2(a). When duplicate entries are found in the course of merging list S_i+j, the entry in the list S_i+j is retained, but the pointer in the other

entry is substituted for "*" in the record specified by the pointer in the entry for S_i+j. This procedure is indicated in Figure 2.

It is thus possible, within O(Kn) time, to generate all feasible sets, assign them to consecutive memory locations, and to provide, for each set S, a list of addresses of sets S-j. The total space required is proportional to the total number of pairs (S,S-j). Space may, of course, be compacted somewhat by reforming the lists of addresses a(S-j) and eliminating pointers.

6. SOLVING THE TRAVELING SALESMAN PROBLEM

It is well known that the dynamic programming approach can be applied to the traveling salesman problem [3,5]. Let H be an (n+1)-node network on node set $N \cup \{0\}$ over which the problem is to be solved. For $S \subseteq N$, let $F(S,j)$ denote the length of a shortest path from node 0 to node j which passes through each of the nodes in S. Then the problem is solved by computing $F(N,0)$ by the recursion equations

$$F(S,j) = \min_{i \in S}\{F(S-i,i) + d_{ij}\}, \tag{6.1}$$

$$F(\emptyset,j) = d_{0j},$$

where d_{ij} is the length of arc (i,j) in H.

If H is complete, the time required to solve equations (6.1) for $F(N,0)$ is $O(n^2 2^n)$. However, if H is sparse or if there are precedence constraints specified by an acyclic digraph G = (N,A), the complexity of the computation may be considerably reduced.

Let us call a path *admissible* if it passes through nodes in an order consistent with the precedence constraints specified by G, and let us call a pair (S,j) *feasible* if there exists an admissible path in H from node 0 to node j which passes through the nodes in S.

A feasible pair (S,j) is said to have size m if $|S| = m$. Feasible pairs can be generated in order of size by a simple adaptation of the procedure given in Section 2. Each feasible pair of size m+1 is of the form (S+i,j), where (S,i) is a feasible pair of size m, $j \notin S+i$, there is an arc (i,j) in H, and there is no arc (k,j) in G such that $k \notin S+i$. If there are K feasible pairs, they can be generated in O(Kn) time and $O(n+k_{max})$ space, where k_m is the number of feasible pairs of size m and $k_{max} = \max_m\{k_m\}$.

The computation of the values $F(S,j)$ can be carried out in the course of state generation. Moreover, the divide-and-conquer technique of Section 4 can be applied. Let

$$\bar{F}(S,j) = \min\{\bar{F}(S-i,i) + d_{ji}\}, \tag{6.2}$$

$$\bar{F}(\emptyset,j) = d_{j0}.$$

Let S_m ($\bar{S}_m$) denote the family of feasible pairs with respect to H and G ($\bar{H}$ and $\bar{G}$). It is not necessarily true that $(S,j) \in S_m$ if and only if $(N-(S+j),j) \in \bar{S}_{n-m-1}$. Nevertheless, we have by analogy with (4.2), for any m,

$$F(N,0) = \min\{F(S,j) + \bar{F}(N-(S+j),j) \mid (S,j) \in S_m, (N-(S+j),j) \in \bar{S}_{n-m-1}\}. \quad (6.3)$$

Using the approach of Section 4, equations (6.3) can be used to construct an optimal sequence in $O(Kn)$ time and $O(n+k_{max})$ space. (In fact, there may be considerable advantage to this approach, resulting from the fact that not both (S,j) and $(N-(S+j),j)$ may be feasible.) In the (conventional) case that H is complete and G is empty, note that

$$\begin{aligned} k_{max} &= k_{n/2} \\ &= \frac{n}{2}\binom{n}{n/2} \\ &\approx \frac{\sqrt{n}}{2} 2^n, \end{aligned}$$

and an optimal sequence can be constructed in $O(n^{1/2}2^n)$ space.

ACKNOWLEDGMENTS

The author wishes to thank Kenneth Baker for suggesting this research with his presentation at the POPCORN Festival (Prominent Open Problems in Combinatorial Optimization (Relevant or Not)), held at the Mathematisch Centrum, Amsterdam, April 6, 1979, and Jan Karel Lenstra and Alexander Rinnooy Kan for organizing the festival.

This research was supported in part by NSF Grant MCS76-17605 and by NATO Special Research Grant 9.2.02 (SRG.7).

REFERENCES

1. H.M. ABDEL-WAHAB, T. KAMEDA, Scheduling to minimize maximum cumulative cost subject to series-parallel precedence constraints. *Operations Res.* 26(1978)141-158.
2. K.R. BAKER, L.E. SCHRAGE, Finding an optimal sequence by dynamic programming: an extension to precedence-related tasks. *Operations Res.* 26 (1978)111-120.
3. R.E. BELLMAN, Dynamic programming treatment of the traveling salesman problem. *J. Assoc. Comput. Mach.* 9(1962)61-63.
4. R.L. GRAHAM, E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Discrete Math.*, to appear.
5. M. HELD, R.M. KARP, A dynamic programming approach to sequencing problems. *J. SIAM* 10(1962)196-210.
6. M. HELD, R.M. KARP, R. SHARESHIAN, Assembly-line balancing - dynamic programming with precedence constraints. *Operations Res.* 11(1963)442-459.
7. E.L. LAWLER, On scheduling problems with deferral costs. *Management Sci.* 11(1964)280-288.
8. E.L. LAWLER, Optimal sequencing of a single machine subject to precedence constraints. *Management Sci.* 19(1973)544-546.
9. E.L. LAWLER, The quadratic assignment problem: a brief review. In: B. ROY (ed.), *Combinatorial Programming: Methods and Applications*, Reidel, Dordrecht (1975)351-360.
10. E.L. LAWLER, Fast approximation algorithms for knapsack problems. *Math. Oper. Res.*, to appear.
11. E.L. LAWLER, Sequencing problems with series parallel precedence constraints. *Proc. Summer School in Combinatorial Optimization*, Urbino, Italy, 1978, to appear.
12. C.L. MONMA, J.B. SIDNEY, Sequencing with series parallel precedence constraints. *Math. Oper. Res.*, to appear.