BA

stichting
mathematisch
centrum

MC

AFDELING MATHEMATISCHE BESLISKUNDE BW 50/75 JULI

J.K. LENSTRA & A.H.G. RINNOOY KAN

A RECURSIVE APPROACH TO THE GENERATION OF COMBINATORIAL CONFIGURATIONS

2e boerhaavestraat 49 amsterdam

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

AMS (MOS) subject classification scheme (1970): 05A05, 05A10, 90B35, 90C10

A RECURSIVE APPROACH TO THE GENERATION OF COMBINATORIAL CONFIGURATIONS

J.K. LENSTRA
Mathematisch Centrum, Amsterdam

A.H.G. RINNOOY KAN
Graduate School of Management, Delft

ABSTRACT

Algorithms for generating subsets, lattice-points, combinations and permutations by means of both lexicographic and minimum-change methods are presented. The use of a recursive approach not only leads to concise and elegant descriptions, but also facilitates programming and correctness proofs. The resulting algorithms turn out to be certainly no less efficient than previous iterative generators. Some applications of explicit enumeration to problems of combinatorial optimization, exploiting the minimum-change property, are indicated, and a recursive approach to implicit enumeration methods is discussed.

KEY WORDS AND PHRASES: *recursion, generation of combinatorial configurations, lexicographic generators, minimum-change generators, subsets, lattice-points, Gray code, combinations, permutations, hamiltonian path, combinatorial optimization, explicit enumeration, implicit enumeration*
CR CATEGORIES: 5.30, 5.32, 5.39, 5.40
ADDRESSES OF AUTHORS: J.K. Lenstra, Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam, The Netherlands; A.H.G. Rinnooy Kan, Graduate School of Management, Poortweg 6-8, Delft, The Netherlands.

1. INTRODUCTION

In this paper we present a recursive approach to the generation of combinatorial configurations. More specifically, we consider the generation of *subsets*, *lattice-points*, *combinations* and *permutations* by means of both *lexicographic* and *minimum-change* methods. The first mentioned type of method generates the configurations in a "dictionary" order, whereas the second type produces a sequence in which successive configurations differ as little as possible. In itself, these two approaches are not new. The relative advantages of minimum-change methods have been discussed previously: the entire sequence is generated efficiently, each configuration being derived from its predecessor by a simple change; moreover, a minimum-change generator "may permit the value of the current arrangement to be obtained by a small correction to the immediate previous value" [27].

The very "cleanliness" [21] of combinatorial problems allows a proper demonstration of what we believe to be the advantages of a recursive approach (cf. [1,2.1.5]). Apart from the elegance of the recursive descriptions, both programming and correctness proofs are substantially facilitated by the recursive structure, whereas the algorithms turn out to be certainly no less efficient than previous iterative generators.

Our algorithms are defined as ALGOL 60 procedures. They contain no labels and generate the entire sequence of configurations after one call. Each time a new configuration has been obtained, a call of a procedure "problem" is made. Parameters of this procedure are the configuration and, for minimum-change generators, the positions in which it differs from its predecessor. It has to be defined by the user to handle each configuration in the desired way.

Most previously published procedures [3;4;6;7;8;10;11;12;24;27;35] are organized in such a way that each call generates only the next configuration. This necessitates continual recomputation of the point that has been reached in the sequence [26]. A mechanism for performing this kind of computations efficiently has been devised by Ehrlich [10;13]. We do feel, however, that much of the clarity of essentially recursive algorithms is lost within any iterative implementation.

Our recursive generators are presented in sections 2, 3 and 4 and compared to previously published procedures in section 5. Section 6 contains some applications of explicit enumeration to problems of combinatorial optimization, exploiting the minimum-change property of generators. We conclude with some remarks on a recursive approach to implicit enumeration methods in section 7.

2. SUBSETS AND LATTICE-POINTS

We start by discussing recursive generators of all subsets of a finite set. A *subset* S of a set $\{e_1,\ldots,e_n\}$ will be represented by a binary n-vector x with $x_k = 1$ iff $e_k \in S$. These 2^n vectors correspond to the vertices of the n-dimensional cube. A hamiltonian path on the n-cube defines a sequence of subsets in which *each subset is derived from its predecessor by adding or removing one element*. Such a sequence is called a *binary Gray code* [14;16;37].

The particular sequence which is generated by our algorithm is the binary *reflected* Gray code. Starting from the empty subset, we may produce it in the following way. First, we list the sequence for n-1 elements and add 0's as the n-th components. Secondly, we list the (n-1)-sequence in reversed order, adding 1's as the n-th components. Obviously, the sequence for 0 elements consists only of the empty configuration. Figure 1(a) shows the code for n = 4.

In the above description we can replace "0" by "x^*_n" and "1" by "$1-x^*_n$", where x^* denotes an arbitrary starting configuration. The last configuration in the sequence is adjacent to the first one, since they differ only in their n-th component. It follows that the binary reflected Gray code defines a hamiltonian *circuit* on the n-cube.

If the rules are written down in a more formal way, the following *minimum-change generator of subsets* results.

```
procedure ss mc (problem,n,x); value n,x;
integer n; integer array x; procedure problem;
begin   integer x1;

        procedure gray(n); value n; integer n;
        if n > 1 then
        begin   gray(n-1);
                x[n]:= 1-x[n]; problem(x,n);
                gray(n-1)
        end      else
        begin   x[1]:= x1:= 1-x1; problem(x,1)
        end;

        x1:= x[1];
        problem(x,0); gray(n)
end ss mc;
```

A call "ss mc (problem,n,x*)" has the following effect:

- a hamiltonian path on the n-cube from x* to $y^* = (x_1^*,\dots,x_{n-1}^*,1-x_n^*)$ is traversed;
- in vertex x* a call "problem(x*,0)" is made;
- in each vertex x, reached by a change of the k-th component, a call "problem(x,k)" is made.

The latter two assertions are clear from inspection. To prove the first one, it suffices to show that a call "gray(k)" accomplishes the following: starting from a configuration x, all x' for which $x' \neq x$, $x'_\ell = x_\ell$ for $k < \ell \leq n$, are reached, each exactly once, while no other vertices are reached; the final vertex y is given by $y_k = 1-x_k$, $y_\ell = x_\ell$ for $\ell \neq k$. The proof, which is by induction on k, is clear from the following diagram:

$$\begin{tikzcd}[row sep=small]
 & & (x_1,\dots,x_{k-2},\ x_{k-1},\ x_k,x_{k+1},\dots,x_n) = x \arrow[d] \\
 & \text{gray}(k-1) & (x_1,\dots,x_{k-2},1-x_{k-1},\ x_k,x_{k+1},\dots,x_n) \\
\text{gray}(k) & x_k := 1-x_k & (x_1,\dots,x_{k-2},1-x_{k-1},1-x_k,x_{k+1},\dots,x_n) \arrow[d] \\
 & \text{gray}(k-1) & (x_1,\dots,x_{k-2},\ x_{k-1},1-x_k,x_{k+1},\dots,x_n) = y
\end{tikzcd}$$

Here a broken arrow means that the component does not remain constant; an unbroken arrow indicates that it remains unchanged.

In "ss mc" the deepest level of recursion has been written out explicitly. This device has been applied to all our minimum-change generators and clearly reduces the number of checks to see if the bottom of the recursion has been reached already. It enables us also to deal separately with the first component of x, which is involved in half of the changes.

Iterative implementations of the binary reflected Gray code have been given by Boothroyd [4], Ehrlich [12] and Bitner et al. [3].

A *lexicographic generator of subsets* is even simpler to construct. Configurations x are generated in such a way that $x_n x_{n-1} \ldots x_1$ is an *increasing binary number*. At each level of recursion exactly one component of x is defined and at the bottom a call "problem(x)" is made. Again, the recursive approach makes the correctness proof a trivial one.

```
procedure ss lex (problem,n); value n;
integer n; procedure problem;
begin   integer array x[1:n];

        procedure node(n); value n; integer n;
        if n = 0 then problem(x) else
        begin   x[n]:= 0; node(n-1);
                x[n]:= 1; node(n-1)
        end;

        node(n)
end ss lex;
```

The subset generators are easily adapted to the generation of lattice-points. An n-dimensional *lattice* is defined by two integer n-vectors ℓ and u; its vertices are given by the integer n-vectors x with $\ell_k \leq x_k \leq u_k$ for k = 1,...,n. The n-cube is a lattice with $\ell_k = 0$ and $u_k = 1$ for all k. Thus, a sequence in which *each lattice-point is derived from its predecessor by increasing or decreasing exactly one component by one* may be obtained as a straightforward generalization of the binary reflected Gray code. However, not each lattice contains a hamiltonian circuit, as can be seen by taking $n = 1$, $\ell_1 < u_1-1$ or $n = 2$, $\ell_1 = \ell_2 = 0$, $u_1 = u_2 = 2$; the property that we can start in an arbitrary vertex has been lost. Figure 1 shows some examples.

ℓ	0000	1111	1111
u	1111	1234	4321
1	0000	1111	1111
2	1000	1211	2111
3	1100	1221	3111
4	0100	1121	4111
5	0110	1131	4211
6	1110	1231	3211
7	1010	1232	2211
8	0010	1132	1211
9	0011	1122	1311
10	1011	1222	2311
11	1111	1212	3311
12	0111	1112	4311
13	0101	1113	4321
14	1101	1213	3321
15	1001	1223	2321
16	0001	1123	1321
17		1133	1221
18		1233	2221
19		1234	3221
20		1134	4221
21		1124	4121
22		1224	3121
23		1214	2121
24		1114	1121
	(a)	(b)	(c)

Figure 1 Reflected Gray codes.

Our *minimum-change generator of lattice-points* is presented below.

```
procedure lp mc (problem,n,l,u); value n,l,u;
integer n; integer array l,u; procedure problem;
begin   integer k,x1,l1,u1; boolean array even[1:n]; integer array x[1:n];

        procedure rise(n); value n; integer n;
        if n > 1 then
        begin   boolean rm; integer xn,un,m;
                un:= u[n]; m:= n-1;
                rm:= true; rise(m);
                for xn:= l[n]+1 step 1 until un do
                begin   x[n]:= xn; problem(x,n,0);
                        rm:= ¬rm; if rm then rise(m) else fall(m)
                end
        end      else
                for x1:= l1+1 step 1 until u1 do
                begin   x[1]:= x1; problem(x,1,0)
                end;

        procedure fall(n); value n; integer n;
        if n > 1 then
        begin   boolean rm; integer xn,ln,m;
                ln:= l[n]; m:= n-1;
                rm:= even[n]; if rm then rise(m) else fall(m);
                for xn:= u[n]-1 step -1 until ln do
                begin   x[n]:= xn; problem(x,0,n);
                        rm:= ¬rm; if rm then rise(m) else fall(m)
                end
        end      else
                for x1:= u1-1 step -1 until l1 do
                begin   x[1]:= x1; problem(x,0,1)
                end;

        for k:= 2 step 1 until n do
        begin   x[k]:= l1:= l[k]; u1:= u[k]-l1; even[k]:= (u1÷2)×2 ≠ u1
        end;    x[1]:= l1:= l[1]; u1:= u[1];
        problem(x,0,0); rise(n)
end lp mc;
```

One can check easily that a call "lp mc (problem,n,ℓ,u)" has the following effect:

- a hamiltonian path in the lattice, starting from ℓ, is traversed;
- in vertex ℓ a call "problem(ℓ,0,0)" is made;
- in each vertex x, reached by an increase (decrease) of one in the k-th component, a call "problem(x,k,0)" ("problem(x,0,k)") is made.

In "lp mc" we have distinguished explicitly between increases and decreases in a component by means of two separate procedures calling themselves and each other. Similar constructions have been applied to all remaining minimum-change generators in order to add to their transparency and efficiency.

A *lexicographic generator of lattice-points* is again particularly simply described recursively. In this case, $x_n x_{n-1} \ldots x_1$ is an *increasing mixed-radix number*.

```
procedure lp lex (problem,n,l,u); value n,l,u;
integer n; integer array l,u; procedure problem;
begin   integer array x[1:n];

        procedure node(n); value n; integer n;
        if n = 0 then problem(x) else
        begin   integer un,m;
                un:= u[n]; m:= n-1;
                for x[n]:= l[n] step 1 until un do node(m)
        end;

        node(n)
end lp lex;
```

3. COMBINATIONS

The approach, developed in section 2, will now be used to obtain generators of combinations. A *combination* C of m out of n elements $e_1,\dots,e_n$ is represented by a binary n-vector x with $x_k = 1$ iff $e_k \in C$. We define an undirected graph G(n,m) whose vertices are given by these $\binom{n}{m}$ vectors; (x,y) is an edge of G(n,m) iff x and y differ in exactly two components. A hamiltonian path in G(n,m) corresponds to a sequence of combinations in which *each combination is derived from its predecessor by adding one element and removing one element.*

We will use the notation δ^{ℓ} for the concatenation of ℓ δ's; e.g., $1^2 0^3 = 11000$. If I is a sequence of combinations, then $\overline{I}$ denotes the reverse of I and Iδ denotes I with δ added everywhere as the last component.

From the binary reflected Gray code with the empty set as starting configuration we take the subsequence J(n,m) consisting of the subsets that contain exactly m elements. We shall prove that J(n,m) is a hamiltonian path in G(n,m) from $x^* = 1^m 0^{n-m}$ to $y^* = 1^{m-1} 0^{n-m} 1$ (note that x^* and y^* are adjacent) if $1 \le m \le n-1$; J(n,0) and J(n,n) consist of only one vertex.

The proof proceeds by induction on n, the case n = 1 being obvious. For $n > 1$, $1 \le m \le n-1$, it follows from the recursive structure of the reflected Gray code that

$$J(n,m) = J(n-1,m)0, \overline{J}(n-1,m-1)1.$$

By the induction hypothesis these two parts are hamiltonian paths which look as follows:

$$J(n,m) = \begin{cases} 1^m 0^{n-m-1} 0, \dots, 1^{m-1} 0^{n-m-1} 1 0 \overset{*}{,} 1^{m-2} 0^{n-m} 1 1, \dots, 1^{m-1} 0^{n-m} 1 & \text{if } m > 1, \\ 1 0^{n-2} 0, \dots, 0^{n-2} 1 0 \overset{*}{,} 0^{n-1} 1 & \text{if } m = 1. \end{cases}$$

Inspection shows that the transitions * are edges in G(n,m), so J(n,m) is a hamiltonian path, as was to be proved. Figure 2 shows J(5,2) and J(5,3).

Combining the recursion scheme of "ss mc" and the results presented above, we obtain the following *minimum-change generator of combinations.*

⋮

```
procedure cb mc (problem,n,m); value n,m;
integer n,m; procedure problem;
begin   integer k; integer array x[1:n];

        procedure over(n,m); value n,m; integer n,m;
        if m > 1 then
        begin   if n-1 > m then over(n-1,m);
                x[n]:= 1; x[m-1]:= 0; problem(x,n,m-1);
                revo(n-1,m-1)
        end      else
        for m:= 2 step 1 until n do
        begin   x[m]:= 1; x[m-1]:= 0; problem(x,m,m-1)
        end;

        procedure revo(n,m); value n,m; integer n,m;
        if m > 1 then
        begin   over(n-1,m-1);
                x[n]:= 0; x[m-1]:= 1; problem(x,m-1,n);
                if n-1 > m then revo(n-1,m)
        end      else
        for m:= n step -1 until 2 do
        begin   x[m]:= 0; x[m-1]:= 1; problem(x,m-1,m)
        end;

        for k:=   1 step 1 until m do x[k]:= 1;
        for k:= m+1 step 1 until n do x[k]:= 0;
        problem(x,0,0); if n > m ∧ m > 0 then over(n,m)
end cb mc;
```

A call "cb mc (problem,n,m)" has the following effect:

- the hamiltonian path $J(n,m)$ in $G(n,m)$ from $x^* = 1^m0^{n-m}$ to $y^* = 1^{m-1}0^{n-m}1$ is traversed;
- in vertex x^* a call "problem(x^*,0,0)" is made;
- in each vertex x, reached by adding e_k and removing e_ℓ, a call "problem(x,k,ℓ)" is made.

These assertions are proved along the same lines as those for "ss mc". Calls "over(n,m)" and "revo(n,m)" generate J(n,m) and $\overline{J}$(n,m) respectively, and the case m = 1 has been handled separately.

The above method has been discovered independently by Tang and Liu [33;24]. It is instructive to compare their presentation to the above one; the justification of their iterative description [33] and algorithm [24] is an arduous task, involving the analysis of eleven special cases. Recently, Bitner et al. [3] have given a recursive description and iterative implementation of the same method.

As a more general result it is easily proved that in the subsequence of the binary reflected Gray code consisting of those *subsets which contain at least m_1 and at most m_2 elements*, each subset is derived from its predecessor by adding one element and/or removing one element. The construction of a recursive generator of these configurations is left as a challenge to the reader.

At the same time one might consider the problem of the sultan, who, being in the possession of fourteen wives but only four spare places on his couch, seeks for a *maximum-change* sequence of thousand-and-one different nights.

1	11000	11000	00011	11100	11100	00111
2	01100	10100	10001	10110	11010	10011
3	10100	01100	01001	01110	10110	01011
4	00110	01010	00101	11010	01110	01101
5	01010	10010	00110	10011	01101	10101
6	10010	00110	10010	01011	10101	11001
7	00011	00101	01010	00111	11001	11100
8	00101	10001	01100	10101	10011	11010
9	01001	01001	10100	01101	01011	10110
10	10001	00011	11000	11011	00111	01110
	J(5,2)	K(5,2)	L(5,2)	J(5,3)	K(5,3)	L(5,3)

Figure 2 **Minimum-change combination sequences.**

Now let G'(n,m) be a subgraph of G(n,m) on the same vertex set; an edge (x,y) of G(n,m) is an edge of G'(n,m) iff all components of x and y between the exchanged elements are zero. A hamiltonian path in G'(n,m) corresponds to an *order preserving* sequence of combinations. One of these paths, K(n,m) from 1^m0^{n-m} to $0^{n-m}1^m$ is defined by

$$K(n,m) = K(n-1,m)0,\overline{K}(n-2,m-1)01,K(n-2,m-2)11;$$

another one, L(n,m), starting from $0^{n-m}1^{m}$ and ending in $1^{m}0^{n-m}$ if m is even and in $0^{n-m-1}1^{m}0$ is m is odd, is given by

$$L(n,m) = \begin{cases} L(n-1,m-1)1, L(n-1,m)0 & \text{if } m \text{ is even,} \\ L(n-1,m-1)1, K(n-1,m)0 & \text{if } m \text{ is odd.} \end{cases}$$

Figure 2 shows some examples. The inductive proofs and recursive implementations are left to the reader.

The recursive definition of K(n,m) is due to Knuth [9]. An iterative description, based on Lathroum's work, has been given by Chase [9]; see also [13;10]. The iterative algorithms of Chase [8] and Ehrlich [11] generate L(n,m) and K(n,m) respectively.

Finally, a *lexicographic generator of combinations* produces the configurations in such a way that $x_n x_{n-1} \ldots x_1$ is an *increasing binary number*.

```
procedure cb lex (problem,n,m); value n,m;
integer n,m; procedure problem;
begin   integer array x[1:n];

        procedure node(n,m); value n,m; integer n,m;
        if m = 0 then
        begin   for n:= n step -1 until 1 do x[n]:= 0; problem(x)
        end      else
        if m = n then
        begin   for n:= n step -1 until 1 do x[n]:= 1; problem(x)
        end      else
        begin   x[n]:= 0; node(n-1,m);
                x[n]:= 1; node(n-1,m-1)
        end;

        node(n,m)
end cb lex;
```

4. PERMUTATIONS

We now consider the generation of permutations. An *n-permutation* of a set $\{x_1^*,\dots,x_n^*\}$ is determined by an n-vector consisting of the elements in some order. We define an undirected graph G(n) whose vertices are given by these n! vectors; (x,y) is an edge of G(n) iff x and y differ only in two neighbouring components. A hamiltonian path in G(n) corresponds to a sequence of permutations in which *each permutation is derived from its predecessor by transposing two elements in adjacent positions.*

We may construct such a sequence inductively as follows. For n = 1, it consists of the 1-permutation. Let the sequence of (n-1)-permutations be given. Placing x_n^* at the right of the first (n-1)-permutation, we obtain the first n-permutation. The n-1 next ones are obtained by successively interchanging x_n^* with its left neighbour. After that, x_n^* is found at the left of the first (n-1)-permutation. Replacing this (n-1)-permutation by its successor in the (n-1)-sequence gives us the (n+1)-th n-permutation, and the n-1 next ones arise from successive transpositions of x_n^* with its right neighbour. Then x_n^* is found at the right of the second (n-1)-permutation, which is now replaced by the third one, and the process starts all over again. It is easily seen that the first and last permutations in the sequence are given by $x^* = (x_1^*,\dots,x_n^*)$ and $y^* = (x_2^*,x_1^*,x_3^*,\dots,x_n^*)$ respectively; again, they are adjacent and we have found a hamiltonian circuit in G(n).

Figures 3 and 4(mc1) show the graphs G(n) for $n \le 4$ and the sequence for n = 4. Note that G(4) is the edge graph of a solid truncated octahedron, replicas of which fill entire 3-space. Similar statements of this remarkable property hold for all n.

The following *minimum-change generator of permutations* produces the sequence described above.

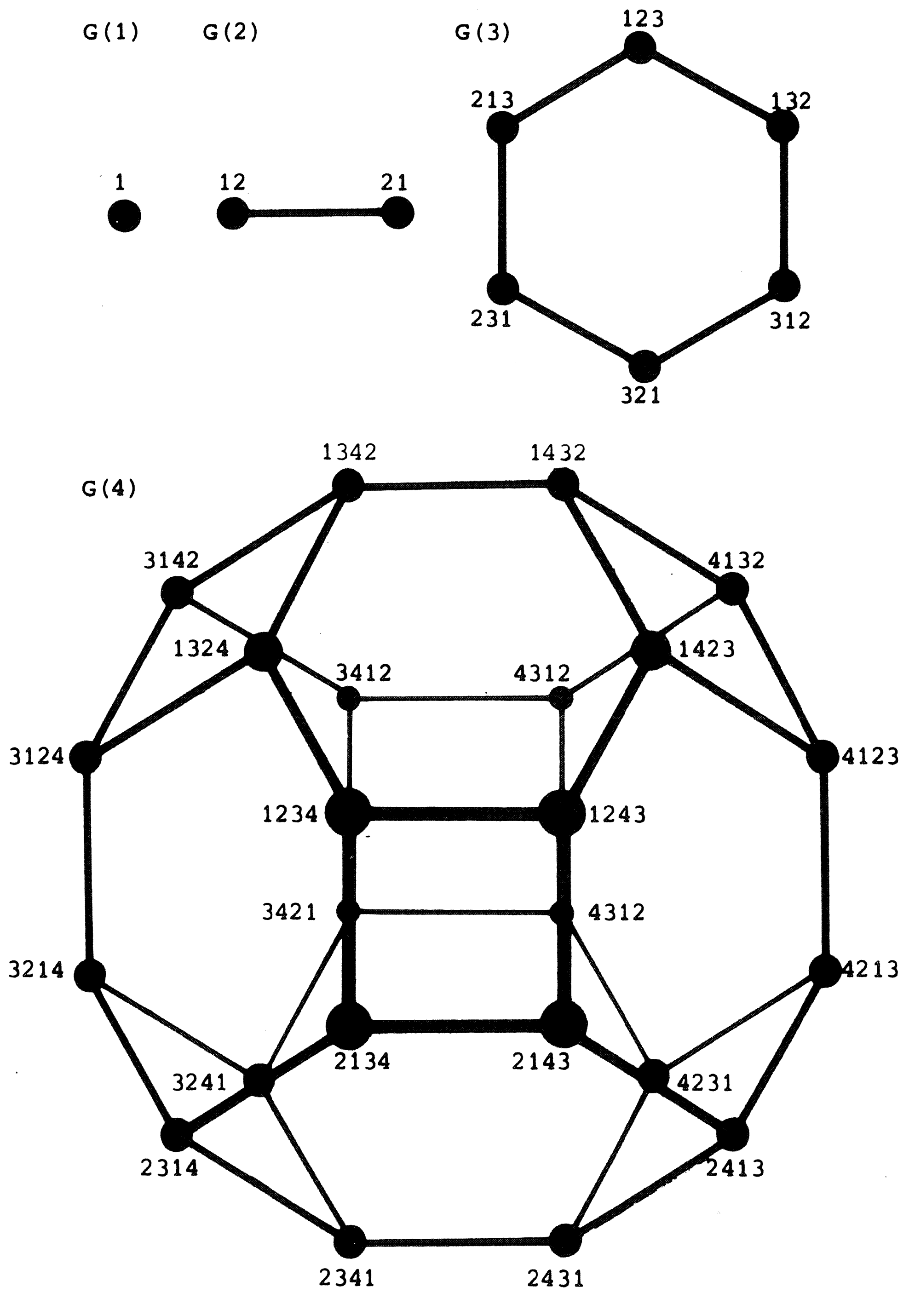

Figure 3 Graphs G(n).

```
procedure pm mc1 (problem,n,x); value n,x;
integer n; array x; procedure problem;
begin   real xn; integer k,q; boolean array r[1:n];

        procedure rite(i); value i; integer i;
        if i < n then
        begin   boolean rj; real xi; integer ti,j;
                xi:= x[q]; j:= i+1;
                q:= q-1;
                rj:= r[j]; if rj then rite(j) else left(j);
                for ti:= 2 step 1 until i do
                begin   k:= q+ti;
                        x[k-1]:= x[k]; x[k]:= xi; problem(x,k-1);
                        rj:= ¬rj; if rj then rite(j) else left(j)
                end;
                r[j]:= ¬rj
        end     else
        begin   q:= 0;
                for k:= 2 step 1 until n do
                begin   x[k-1]:= x[k]; x[k]:= xn; problem(x,k-1)
                end
        end;

        procedure left(i); value i; integer i;
        if i < n then
        begin   boolean rj; real xi; integer ti,j;
                xi:= x[q+i]; j:= i+1;
                rj:= r[j]; if rj then rite(j) else left(j);
                for ti:= i-1 step -1 until 1 do
                begin   k:= q+ti;
                        x[k+1]:= x[k]; x[k]:= xi; problem(x,k);
                        rj:= ¬rj; if rj then rite(j) else left(j)
                end;
                r[j]:= ¬rj;
                q:= q+1
        end     else
        begin   for k:= n-1 step -1 until 1 do
                begin   x[k+1]:= x[k]; x[k]:= xn; problem(x,k)
                end;
                q:= 1
        end;

        xn:= x[n]; q:= 0; for k:= 2 step 1 until n do r[k]:= false;
        problem(x,0); if n ≥ 2 then left(2)
end pm mc1;
```

A call "pm mc1 (problem,n,x*)" has the following effect:
if n = 1, then a call "problem(x*,0)" is made, and else
- a hamiltonian path in G(n) from x* to $y^* = (x^*_2,x^*_1,x^*_3,\dots,x^*_n)$ is traversed;
- in vertex x* a call "problem(x*,0)" is made;
- in each vertex x, reached by transposition of the elements in positions k and k+1, a call "problem(x,k)" is made.

The latter two assertions are clear from inspection. The proof of the first one may be left to the reader. As a hint, we note that just before a call "rite(i)" or "left(i)" and immediately after the execution, x, r and q satisfy the following conditions: $\{j|i \le j \le n,\ r_j\}$ has exactly q elements, and if we write

$$\{j|i \le j \le n,\ r_j\} = \{j_1,\dots,j_q\} \quad \text{with } j_1 > \dots > j_q,$$
$$\{j|i \le j \le n,\ \neg r_j\} = \{j_{q+i},\dots,j_n\} \text{ with } j_{q+i} < \dots < j_n,$$

then $x_k = x^*_{j_k}$ for $k = 1,\dots,q,q+i,\dots,n$.

Using the integer q to determine the place of the transpositions is simpler and more efficient than keeping track of the inverse permutation for that purpose, as is done in [10;11]. As usual, we have distinguished between two types of changes, in this case the leftward and rightward moves of the elements. Since the n-th element is transposed in (n-1)/n of the cases (cf.[10]), it again pays to write out explicitly the bottom of the recursion.

Permutation generators have been surveyed by Lehmer [21], Ord-Smith [26;27] and Wells [37]. The above method has been discovered independently by Trotter [35] and by Johnson [18]; Trotter's iterative algorithm was for a number of years the fastest permutation generator [27]. A more efficient iterative implementation has been presented by Ehrlich [11]; see also [13;10]. We will discuss below a different minimum-change method which has been found by Wells

[36] and simplified by Boothroyd in recursive [5] and iterative [6;7] implementations. In 1971 [27], the latter algorithm [7] was found to be the fastest of six generators, including [35] and [6].

	mc1	mc2	lex	plex
1	1234	1234	4321	4321
2	1243	2134	3421	3421
3	1423	2314	4231	4231
4	4123	3214	2431	2431
5	4132	3124	3241	2341
6	1432	1324	2341	3241
7	1342	1342	4312	4312
8	1324	3142	3412	3412
9	3124	3412	4132	4132
10	3142	4312	1432	1432
11	3412	4132	3142	1342
12	4312	1432	1342	3142
13	4321	1423	4213	4123
14	3421	4123	2413	1423
15	3241	4213	4123	4213
16	3214	2413	1423	2413
17	2314	2143	2143	2143
18	2341	1243	1243	1243
19	2431	3241	3214	1324
20	4231	2341	2314	3124
21	4213	2431	3124	1234
22	2413	4231	1324	2134
23	2143	4321	2134	2314
24	2134	3421	1234	3214

Figure 4 Permutation sequences.

Let G'(n) be an extension of G(n) on the same vertex set; (x,y) is an edge of G'(n) iff x and y differ in only two components. A hamiltonian path in G'(n) corresponds to a sequence of permutations in which *each permutation is derived from its predecessor by transposing two elements*. Such a path is defined by a sequence of n!-1 transpositions. Denoting the transposition of the elements in positions k and ℓ by $k \leftrightarrow \ell$, we may define the transposition sequence corresponding to the Wells-Boothroyd method by

$$T(n) = T(n-1), m_1 \leftrightarrow n, T(n-1), m_2 \leftrightarrow n, \ldots, T(n-1), m_{n-1} \leftrightarrow n, T(n-1)$$

where

$$m_k = \begin{cases} n-k & \text{if } n \text{ is even and } k > 2, \\ n-1 & \text{if } n \text{ is odd or } k \le 2; \end{cases}$$

note that T(1) is empty. Figure 4(mc2) shows the resulting sequence for n = 4.

The above description leads direct to our second *minimum-change generator of permutations*.

```
procedure pm mc2 (problem,n,x); value n,x;
integer n; array x; procedure problem;
begin   real xk,xm;

        procedure even(n); value n; integer n;
        if n > 2 then
        begin   real xn; integer k,m;
                m:= n-1; xn:= xm;
                odd(m);
                for k:= m, m, m-2 step -1 until 1 do
                begin   x[n]:= xk:= x[k]; x[k]:= xn; xn:= xk; problem(x,k,n);
                        odd(m)
                end
        end     else
        begin   x[2]:= x[1]; x[1]:= xm; problem(x,1,2)
        end;

        procedure odd(n); value n; integer n;
        begin   real xn; integer k,m;
                m:= n-1; xn:= x[n]; xm:= x[m];
                even(m);
                for k:= m step -1 until 1 do
                begin   x[n]:= xk:= x[m]; x[m]:= xm:= xn; xn:= xk; problem(x,m,n);
                        even(m)
                end
        end;

        problem(x,0,0); if n ≥ 2 then
        begin   if (n÷2)×2 = n then begin xm:= x[n]; even(n) end else odd(n)
        end
end pm mc2;
```

A call "pm mc2 (problem,n,x*)" has the following effect:
if n = 1, then a call "problem(x*,0,0)" is made, and else
- a hamiltonian path in G'(n) from x* to y* is traversed, where

$$y^* = \begin{cases} (x_2^*,\ldots,x_{n-3}^*,x_{n-1}^*,x_n^*,x_{n-2}^*,x_1^*) & \text{if m is even,} \\ (x_1^*,\ldots,x_{n-2}^*,x_n^*,x_{n-1}^*) & \text{if m is odd;} \end{cases}$$

- in vertex x* a call "problem(x*,0,0)" is made;
- in each vertex x, reached by transposition of the elements in positions k and ℓ, a call "problem(x,k,ℓ)" is made.

The inductive proof is left to the reader. We have distinguished between n even and n odd, and the case n = 2 has been handled separately.

We make one final remark on minimum-change sequences of permutations. Given an undirected graph H(n) on n vertices, we define an undirected graph $G_H(n)$ on the set of n-permutations; (x,y) is an edge of $G_H(n)$ iff x can be obtained from y by a single transposition of the elements in positions k and ℓ, where (k,ℓ) is an edge of H(n). One can prove that $G_H(n)$ *contains a hamiltonian circuit iff* H(n) *contains a spanning tree.* The "only if"-part is obvious; the "if"-part follows by an inductive argument. In the Trotter-Johnson algorithm the "transposition graph" H(n) is a tree with edge set $\{(k,k+1)|k = 1,\ldots,n-1\}$; it is properly contained in the transposition graph of the Wells-Boothroyd method.

The *lexicographic generator of permutations* below produces the configurations in such a way that $x_n x_{n-1} \ldots x_1$ is an *increasing* n-*ary number*. A slight modification leads to a more efficient *pseudo-lexicographic generator of permutations*. Figure 4(lex,plex) shows the lexicographic and pseudo-lexicographic sequences for n = 4.

```
procedure pm lex (problem,n); value n;
integer n; procedure problem;
begin   integer h; integer array x[1:n];
```

```
        procedure node(n); value n; integer n;
        if n = 1 then problem(x) else
        begin   integer k,m,xn;
                m:= n-1; xn:= x[n];
                node(m);
                for k:= m step -1 until 1 do
                begin   x[n]:= h:= x[k]; x[k]:= xn; xn:= h;
                        node(m)
                end;
                for k:= n step -1 until 2 do x[k]:= x[k-1]; x[1]:= xn
        end;

        for h:= n step -1 until 1 do x[h]:= n+1-h;
        node(n)
end pm lex;

procedure pm plex (problem,n); value n;
integer n; procedure problem;
begin   integer h; integer array x[1:n];

        procedure node(n); value n; integer n;
        if n = 1 then problem(x) else
        begin   integer k,m,xk,xn;
                m:= n-1; xn:= x[n];
                node(m);
                for k:= m step -1 until 1 do
                begin   x[n]:= xk:= x[k]; x[k]:= xn;
                        node(m);
                        x[k]:= xk
                end;
                x[n]:= xn
        end;

        for h:= n step -1 until 1 do x[h]:= n+1-h;
        node(n)
end pm plex;
```

5. COMPUTATIONAL COMPARISON

The algorithms presented in sections 2, 3 and 4 have been compared to ALGOL 60 versions of the following minimum-change algorithms:

- Ehrlich's "loopless" algorithms "ss pc1273" [12], "cb acm466" [11] and "pm acm466" [11], which generate subsets according to the binary reflected Gray code, combinations by an order-preserving method and permutations by adjacent transpositions respectively;
- Liu and Tang's algorithm "cb acm452" [24] which generates combinations by the method, based on the Gray code;
- Chase's algorithm "cb acm382" [8] for the order-preserving generation of combinations;
- Trotter's algorithm "pm acm115" [35;27] which generates permutations by adjacent transpositions;
- Boothroyd's algorithms "pm bcb6" [5] and "pm bcj30" [7;27] which are recursive and iterative generators of permutations by transpositions.

Table 1 shows the result of the comparison. The running times have been measured during one uninterrupted run on the Electrologica X8 computer of the Mathematisch Centrum; a procedure with an empty body was chosen for the actual parameter "problem". Our minimum-change algorithms turn out to be faster than corresponding previously published procedures. Although the time differences are not spectacular, a recursive approach should certainly not be rejected on grounds of computational inefficiency *a priori*.

Results like the above ones unavoidably remain computer and compiler dependent. It is of interest to note in this context that some experiments using PASCAL on the Control Data Cyber 73-28 of the SARA Computing Centre in Amsterdam showed a nineteen-fold increase in speed for the recursive "ss mc" and a fourteen-fold increase for the iterative "ss pc1273". On the other hand, the running times of the iterative generators may be reduced by up to twenty percent by a different transformation of these generators into PASCAL procedures producing all configurations at one call.

In order to develop a computer independent measure of efficiency, let us define

$$a = \lim_{n\to\infty} \frac{\text{number of array subscript evaluations}}{\text{number of generated configurations}},$$

array access being a dominant factor in this type of ALGOL 60-procedure [27]. For recursive algorithms, evaluation of a is accomplished by the solution of

configurations	algorithm	reference	time	a	restrictions
SUBSETS					$n \geq 1$
$n = 15$	ss lex	h.l., section 2	51.6	2	
	ss mc	h.l., section 2	36.7	$1\frac{1}{2}$	
	ss pc1273	Ehrlich [12]	51.7	$\geq 4, \leq 10$	
LATTICE-POINTS					$n \geq 1$, $\ell_k \leq u_k$
$n = 15$	lp lex	h.l., section 2	89.5	5	
$\ell_k = 0$, $u_k = 1$	lp mc	h.l., section 2	50.2	$2\frac{1}{4}$	
$n = 8$	lp lex	h.l., section 2	154.3	7.87	
$\ell_k = 1$, $u_k = k$	lp mc	h.l., section 2	81.5	2.80	
$n = 8$	lp lex	h.l., section 2	57.6	1	
$\ell_k = 1$, $u_k = n+1-k$	lp mc	h.l., section 2	35.5	1	
COMBINATIONS					$n \geq 1$, $0 \leq m \leq n$
$n = 15$	cb lex	h.l., section 3	7.6	$4\frac{1}{2}$	
$m = n/3$	cb mc	h.l., section 3	3.6	2	
	cb acm452	Liu & Tang [24]	7.5	≥ 6	
	cb acm382	Chase [8]	8.8	≥ 6	
	cb acm466	Ehrlich [11]	6.9	$\geq 8, \leq 16$	$1 \leq m \leq n-1$
$n = 15$	cb lex	h.l., section 3	7.7	$4\frac{1}{2}$	
$m = 2n/3$	cb mc	h.l., section 3	4.7	2	
	cb acm452	Liu & Tang [24]	7.8	≥ 6	
	cb acm382	Chase [8]	8.7	≥ 6	
	cb acm466	Ehrlich [11]	7.1	$\geq 8, \leq 16$	$1 \leq m \leq n-1$
PERMUTATIONS					$n \geq 1$
$n = 8$	pm lex	h.l., section 4	92.4	6.44	
	pm plex	h.l., section 4	82.5	5.44	
	pm mc1	h.l., section 4	42.9	3	
	pm acm115	Trotter [35;27]	91.3	≥ 7	$n \geq 2$
	pm acm466	Ehrlich [11]	58.1	3	$n \geq 3$, $n \neq 4$
	pm mc2	h.l., section 4	54.3	3.35	
	pm bcb6	Boothroyd [5]	103.3	6.72	
	pm bcj30	Boothroyd [7;27]	83.6	>3.16	$n \geq 5$

Table 1 Comparison of various generators.

time: running time in seconds; a: average array access (in the limit).

recursive expressions. For all iterative algorithms except Ehrlich's ones, only lower bounds can be given; it is not clear if finite limits exist.

6. EXPLICIT ENUMERATION

The generators can be used to solve many combinatorial optimization problems through enumeration and evaluation of all feasible solutions. Needless to say, only very small problems can be solved by such a brute force approach, even if the minimum-change property of the generators is exploited. However, they can be applied to validate more complicated solution methods by checking their results on small problems.

More specifically, the procedures "ss mc" and "lp mc" can be used to solve *integer programming problems*. Krol [19] reports that a lexicographic method that he is curiously unable to describe, is superior to several implicit enumeration algorithms. A more sophisticated approach to this type of problem arises in the context of cutting-plane algorithms [15]. It involves a complete enumeration of the vertices on a facet of the integer lattice that contains (or is likely to contain) a feasible integer point. If such a point x^* is indeed found, the cut $cx \geq cx^*+1$ can be added to the lp-tableau; else, we can cut off the enumerated facet.

Explicit enumeration of permutations $x = (x_1,\dots,x_n)$ can be used to solve *sequencing problems* P of the form $\min_x z_P(x)$. An example is the *quadratic assignment problem* (QAP):

$$z_{QAP}(x) = \sum_{i=1}^{i=n} \sum_{j=1}^{j=n} c_{x_i x_j} d_{ij}$$

where c and d are non-negative nxn-matrices. If we take $d_{ij} = 1$ for $i > j$, $d_{ij} = 0$ otherwise, we obtain the *acyclic subgraph problem* (ASP) [22]. Analogously, the choice $d_{12} = d_{23} = \dots = d_{n-1,n} = d_{n1} = 1$, $d_{ij} = 0$ otherwise, leads to the well-known *travelling salesman problem*, that is called *symmetric* if $c_{ij} = c_{ji}$ for all i,j.

If we define the *reflection* of x by $\bar{x} = (x_n,\dots,x_1)$, it is obvious that $z_{ASP}(\bar{x}) = \sum_{i\neq j} c_{ij} - z_{ASP}(x)$ for the ASP and $z_{TSP}(\bar{x}) = z_{TSP}(x)$ for the symmetric TSP. It follows that for these two problems it suffices to enumerate a *reflection-free* set of permutations. Further, since $z_{TSP}((x_{k+1},\ldots,x_n,x_1,\ldots,x_k)) = z_{TSP}(x)$ for any k, we may fix one of the components of x when solving a TSP. The $(n-1)!/2$ solutions to a symmetric TSP are the hamiltonian circuits in a complete undirected graph; they are called *rosary permutations* [17;28;32].

In the Trotter-Johnson algorithm, discussed in section 4, the elements x_1^* and x_2^* are transposed half-way. If a permutation x is generated before this transposition, then its reflection $\bar{x}$ occurs thereafter. Hence the first $n!/2$ permutations form a reflection-free set (cf. [18]). Generally, the $n!/(m-1)!$ permutations preserving the original order of $x_1^*,\ldots,x_{m-1}^*$, can be generated by a simple adaptation of "pm mc1":

```
procedure pp mc1 (problem,n,m,x); ...;
begin   ...
        ...; if n ≥ m then left(m)
end pp mc1;
```

The above sequencing problems may now be solved by calls "pm mc1 (qap,n,x)", "pp mc1 (asp,n,3,x)" and "pp mc1 (tsp,n-1,if symmetric then 3 else 2,x)", where "qap", "asp" and "tsp" are procedures which compute the cost changes occurring in these problems.

Several suboptimal approaches to combinatorial optimization problems involve the systematic exploration of a neighbourhood of some given solution, starting anew from improved solutions until no further improvement is found and a local optimum has been obtained [29].

For instance, a solution x to the TSP is called m-*opt* if it is impossible to obtain a better solution by replacing m of its links (x_i, x_{i+1}) by a different set of m links [23]. A 3-opt method, derived from "cb mc" by replacing the general recursion mechanism by a set of three nested for-loops and inserting the appropriate statements instead of the "problem"-calls, proved to be more efficient than the algorithm presented by Lin [23].

Analogously, one can obtain efficient suboptimal algorithms for the QAP and the ASP. The approach might be applicable also to other types of difficult optimization problems, e.g. in the area of machine scheduling.

7. IMPLICIT ENUMERATION

The lexicographic procedures presented in sections 2, 3 and 4 can easily be adapted to be used for implicit enumeration purposes by adding a lower bound calculation on all possible completions of a partial configuration. In the early fifties, Lehmer used such an approach to solve the linear assignment problem (!) [34]; similarly, the enumeration scheme of "pm plex" has been applied to the travelling salesman problem [2]. The fact that our recursive generators coupled with a simple lower bound may well outperform sophisticated implicit enumeration algorithms that suffer from a large computational overhead (see [31]) underlines the applicability of recursive programming to implicit enumeration methods of the *branch-and-bound* type in general. We shall present an ALGOL-like description of branch-and-bound procedures, indicating in which case a recursive approach might suitably be used. For a specification of the necessary properties of the elements which constitute a branch-and-bound procedure, we refer to the axiomatic framework in [25] and its correction in [30]. Some examples of these methods have been surveyed in [20].

Suppose then, that given a *set* X *of feasible solutions* and a *criterion function* $c\colon X \to \mathbb{R}$, we want to find an $x^* \in X$ such that $c(x^*) = \min_{x\in X} c(x)$. A branch-and-bound procedure to find such an *optimal solution* can be characterized as follows.

- Throughout the execution of the procedure, the *best solution* x^* *found so far* provides an *upperbound* $c(x^*)$ on the value of the optimal solution.
- A *branching rule* b associates to $Y \subset X$ a family $b(Y)$ of subsets such that $\cup_{Y'\in b(Y)} Y' = Y$; the subsets Y' are the *descendants* of the *parent* subset Y. This rule only has to be defined on a set $\mathcal{X}$ with $X \in \mathcal{X}$ and $b(Y) \subset \mathcal{X}$ for any $Y \in \mathcal{X}$.
- A *bounding rule* $\mathrm{lb}\colon \mathcal{X} \to \mathbb{R}$ provides a *lower bound* $\mathrm{lb}(Y) \leq c(x)$ for all $x \in Y \in \mathcal{X}$. *Elimination* of Y occurs if $\mathrm{lb}(Y) \geq c(x^*)$.
- A *predicate* $\xi\colon \mathcal{X} \to \{\underline{\text{true}},\underline{\text{false}}\}$ indicates if during the examination of Y (e.g. during the calculation of $\mathrm{lb}(Y)$) a feasible solution $x(Y)$ is generated which has to be evaluated. *Improvement* of x^* occurs if $c(x^*) > c(x(Y))$.
- A *search strategy* chooses a subset from the collection of generated subsets which have so far neither been eliminated nor led to branching.

It turns out that, of the three search disciplines that have been used most frequently, only two are suitable for recursive implementation. To illustrate this point, we shall now present three general procedures:

- "bb jumptrack" implements a *frontier search* where a subset with minimal lower bound is selected for examination;
- "bb backtrack1" implements a *depth-first search* where the descendants of a parent subset are examined in an arbitrary order; this type of tree search is known as *newest active node search*;
- "bb backtrack2" implements a *depth-first search* where the descendants are chosen in order of non-decreasing lower bounds; this type is sometimes called *restricted flooding*.

During the tree search, the parameters na and nb count the numbers of subsets that are eliminated and that lead to branching respectively. We define the operation ":f∈" in the statement "s:f∈ S" to mean that s:= s* with f(s*) = $\min_{s \in S}$ f(s); hence, ":∈" indicates an arbitrary choice.

```
procedure bb jumptrack (X,c,x*,$\mathcal{b}$,lb,$\xi$,na,nb);
begin   local $\mathcal{Y}$,$\mathcal{Y}'$,$\mathcal{B} \subset \mathcal{X}$, Y,Y' $\in \mathcal{X}$, LB: $\mathcal{X} \to \mathbb{R}$;
        na:= nb:= 0; $\mathcal{Y}$:= $\emptyset$;
        LB(X):= lb(X); if $\xi$(X) then x*:c∈ {x*,x(X)};
        if LB(X) $\geq$ c(x*) then na:= 1 else $\mathcal{Y}$:= {X};
        while $\mathcal{Y} \neq \emptyset$ do
        begin   Y:LB∈ $\mathcal{Y}$;
                nb:= nb+1; $\mathcal{B}$:= $\mathcal{b}$(Y); $\mathcal{Y}$:= ($\mathcal{Y}$-{Y})$\cup\mathcal{B}$;
                while $\mathcal{B} \neq \emptyset$ do
                begin   Y':∈ $\mathcal{B}$; $\mathcal{B}$:= $\mathcal{B}$-{Y'};
                        LB(Y'):= lb(Y'); if $\xi$(Y') then x*:c∈ {x*,x(Y')}
                end;
                $\mathcal{Y}'$:= {Y'|Y' $\in \mathcal{Y}$, LB(Y') $\geq$ c(x*)};
                na:= na+|$\mathcal{Y}'$|; $\mathcal{Y}$:= $\mathcal{Y}$-$\mathcal{Y}'$
        end
end bb jumptrack;
```

```
procedure bb backtrack1 ($X,c,x^*,\mathcal{b},lb,\xi,na,nb$);
begin   local $Y' \in \mathcal{X}$;

        procedure node($Y$);
        begin   local $\mathcal{B} \subset \mathcal{X}$, $LB \in \mathbb{R}$;
                $LB := lb(Y)$; if $\xi(Y)$ then $x^* :c\in \{x^*, x(Y)\}$;
                if $LB \geq c(x^*)$ then $na := na+1$ else
                begin   $nb := nb+1$; $\mathcal{B} := \mathcal{b}(Y)$;
                        while $\mathcal{B} \neq \emptyset$ do
                        begin   $Y' :\in \mathcal{B}$; $\mathcal{B} := \mathcal{B} - \{Y'\}$;
                                if $LB < c(x^*)$ then node($Y'$)
                        end
                end
        end;

        $na := nb := 0$;
        node($X$)
end bb backtrack1;

procedure bb backtrack2 ($X,c,x^*,\mathcal{b},lb,\xi,na,nb$);
begin   local $\mathcal{B} \subset \mathcal{X}$, $Y' \in \mathcal{X}$, $LB: \mathcal{X} \to \mathbb{R}$;

        procedure node($Y$);
        begin   local $\mathcal{Y} \subset \mathcal{X}$;
                $nb := nb+1$; $\mathcal{Y} := \mathcal{B} := \mathcal{b}(Y)$;
                while $\mathcal{B} \neq \emptyset$ do
                begin   $Y' :\in \mathcal{B}$; $\mathcal{B} := \mathcal{B} - \{Y'\}$;
                        $LB(Y') := lb(Y')$; if $\xi(Y')$ then $x^* :c\in \{x^*, x(Y')\}$
                end;
                while $\mathcal{Y} \neq \emptyset$ do
                begin   $Y' :LB\in \mathcal{Y}$; $\mathcal{Y} := \mathcal{Y} - \{Y'\}$;
                        if $LB(Y') \geq c(x^*)$ then $na := na+1$ else node($Y'$)
                end
        end;

        $na := nb := 0$;
        $LB(X) := lb(X)$; if $\xi(X)$ then $x^* :c\in \{x^*, x(X)\}$;
        if $LB(X) \geq c(x^*)$ then $na := 1$ else node($X$)
end bb backtrack2;
```

Anyone familiar with branch-and-bound will have noticed that the above descriptions only provide a minimal algorithmic framework. Numerous problem-dependent variations may be included in an actual procedure. For instance, elimination of Y may be possible already during the calculation of lb(Y) or may be based on dominance rules or feasibility considerations. In a minor (and in our experience quite successful) variation on "bb backtrack1", the subsets Y' are not chosen arbitrarily but according to some heuristic, e.g. preliminary lower bounds lb'(Y'). Many similar variations are possible and need not be discussed here.

From our experience with branch-and-bound we may conclude, however, that again the recursive approach produces transparent and elegant procedures, in which much administrative work is taken over by the compiler without a noticeable negative effect on overall efficiency. Even in the larger area of implicit enumeration, a recursive approach merits serious consideration.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the valuable help and suggestions from J.D. Alanen, P. van Emde Boas, B.J. Lageweg, H.W. Lenstra, Jr., I. Pohl and the Editor.

REFERENCES

1. Barron, D.W. *Recursive Techniques in Programming.* Macdonald, London, 1968.

2. Barth, W. Ein ALGOL 60 Programm zur Lösung des Traveling Salesman Problems. *Ablauf- und Planungsforschung 9* (1968), 99-105.

3. Bitner, J.R., Ehrlich, G., and Reingold, E.M. Efficient generation of the binary reflected Gray code and its applications. Department of Computer Science, University of Illinois at Urbana-Champaign, 1975.

4. Boothroyd, J. Algorithm 246, Graycode. *Comm. ACM 7* (Dec. 1964), 701.

5. Boothroyd, J. Algorithm 6, Perm. *Comput. Bull. 9* (Dec. 1965), 104.

6. Boothroyd, J. Algorithm 29, Permutation of the elements of a vector. *Comput. J. 10* (Nov. 1967), 311.

7. Boothroyd, J. Algorithm 30, Fast permutation of the elements of a vector. *Comput. J. 10* (Nov. 1967), 311-312.

8. Chase, P.J. Algorithm 382, Combinations of *M* out of *N* objects. *Comm. ACM 13* (June 1970), 368.

9. Chase, P.J. Transposition graphs. *SIAM J. Comput. 2* (June 1973), 128-133.

10. Ehrlich, G. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM 20* (July 1973), 500-513.

11. Ehrlich, G. Algorithm 466, Four combinatorial algorithms. *Comm. ACM 16* (Nov. 1973), 690-691.

12. Ehrlich, G. Reflected binary Gray code. Private communication. Dec. 1973.

13. Even, S. *Algorithmic Combinatorics*. Macmillan, New York etc., 1973.

14. Gardner, M. The curious properties of the Gray code and how it can be used to solve puzzles. *Sci. Amer. 227* (Aug. 1972), 106-109.

15. Garfinkel, R.S., and Nemhauser, G.L. *Integer Programming*. Wiley, New York etc., 1972.

16. Gilbert, E.N. Gray codes and paths on the *n*-cube. *Bell System Tech. J. 37* (May 1958), 815-826.

17. Harada, K. Generation of rosary permutations expressed in hamiltonian circuits. *Comm. ACM 14* (June 1971), 373-379.

18. Johnson, S.M. Generation of permutations by adjacent transposition. *Math. Comp. 17* (July 1963), 282-285.

19. Krol, G. *Het Gemillimeterde Hoofd*. Querido, Amsterdam, 1967, 133-138.

20. Lawler, E.L., and Wood, D.E. Branch-and-bound methods: a survey. *Operations Res. 14* (July 1966), 699-719.

21. Lehmer, D.H. The machine tools of combinatorics. In: Beckenbach, E.F. (Ed.). *Applied Combinatorial Mathematics*. Wiley, New York, 1964, 5-31.

22. Lenstra Jr., H.W. The acyclic subgraph problem. Report BW 26/73, Mathematisch Centrum, Amsterdam, 1973.

23. Lin, S. Computer solutions of the traveling salesman problem. *Bell System Tech. J. 44* (Dec. 1965), 2245-2269.

24. Liu, C.N., and Tang, D.T. Algorithm 452, Enumerating combinations of *m* out of *n* objects. *Comm. ACM 16* (Aug. 1973), 485.

25. Mitten, L.G. Branch-and-bound methods: general formulation and properties.

Operations Res. 18 (Jan. 1970), 24-34.

26. Ord-Smith, R.J. Generation of permutation sequences: part 1. *Comput. J. 13* (May 1970), 152-155.

27. Ord-Smith, R.J. Generation of permutation sequences: part 2. *Comput. J. 14* (May 1971), 136-139.

28. Read, R.C. A note on the generation of rosary permutations. *Comm. ACM 15* (Aug. 1972), 775.

29. Reiter, S., and Sherman, G. Discrete optimizing. *J. SIAM 13* (Sep. 1965), 864-889.

30. Rinnooy Kan, A.H.G. On Mitten's axioms for branch-and-bound. Working Paper W/74/45/03, Graduate School of Management, Delft, 1974.

31. Rinnooy Kan, A.H.G., Lageweg, B.J., and Lenstra, J.K. Minimizing total costs in one-machine scheduling. *Operations Res.*, to appear.

32. Roy, M.K. Reflection-free permutations, rosary permutations, and adjacent transposition algorithms. *Comm. ACM 16* (May 1973), 312-313.

33. Tang, D.T., and Liu, C.N. Distance-2 cyclic chaining of constant-weight codes. *IEEE Trans. Computers C-22* (Feb. 1973), 176-180.

34. Tompkins, C. Machine attacks on problems whose variables are permutations. In: *Proc. Sympos. Appl. Math. 6*, Amer. Math. Soc., Providence, 1956.

35. Trotter, H.F. Algorithm 115, Perm. *Comm. ACM 5* (Aug. 1962), 434-435.

36. Wells, M.B. Generation of permutations by transposition. *Math. Comp. 15* (Apr. 1961), 192-195.

37. Wells, M.B. *Elements of Combinatorial Computing.* Pergamon, Oxford etc., 1971.