AFDELING MATHEMATISCHE BESLISKUNDE BW 116/80 JANUARI
(DEPARTMENT OF OPERATIONS RESEARCH)

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

A RECURSIVE APPROACH TO THE IMPLEMENTATION OF ENUMERATIVE METHODS

Preprint

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

1980 Mathematics Subject Classification: 05A05, 68A99, 90C10

A RECURSIVE APPROACH TO THE IMPLEMENTATION OF ENUMERATIVE METHODS

J.K. LENSTRA
Mathematisch Centrum, Amsterdam

A.H.G. RINNOOY KAN
Erasmus University, Rotterdam

ABSTRACT

Algorithms for generating permutations by means of both lexicographic and minimum-change methods are presented. A recursive approach to their implementation leads to transparent procedures that are easily proved correct; moreover, they turn out to be no less efficient than previous iterative generators. Some applications of explicit enumeration to combinatorial optimization problems, exploiting the minimum-change property, are indicated. Finally, a recursive approach to implicit enumeration is discussed.

KEY WORDS & PHRASES: *enumerative methods, recursive implementation, generation of permutations, lexicographic generator, minimum-change generator, combinatorial optimization, explicit enumeration, implicit enumeration, branch-and-bound.*
NOTE: This report is not for review; it will be published in the Proceedings of the School on Analysis and Design of Algorithms in Combinatorial Optimization, Udine, Italy, September 10-21, 1979.

1. INTRODUCTION

The analysis of the inherent computational complexity of combinatorial problems indicates that for many of those problems a polynomial-time algorithm is not likely to exist. It appears that with respect to these problems we have to settle for some form of *enumeration* of the solution set whereby the feasible solutions are identified and an optimal one is obtained. For all but the smallest problems the number of feasible solutions is so large that the use of a computer for the actual computations is unavoidable. Thus, the computational performance of any enumerative method not only depends on algorithmic details but also on the computer implementation. The latter topic forms the subject of this paper.

More specifically, the paper will be devoted to a discussion of a *recursive* approach to the implementation of enumerative methods. We hope to demonstrate that such an approach leads to procedures that are elegant, easy to understand, easily programmed and easily proved correct. While these positive aspects will probably be recognized by most programmers, a familiar argument against recursive procedures suggests that none the less they require inordinate running times. Thus, ironically, many recursive approaches advocated in the literature are implemented after complicated manipulations in an iterative fashion [Barth 1968; Bitner *et al.* 1976; Gries 1975]! We will demonstrate on a simple example that with respect to efficiency a recursive implementation need certainly not be inferior to an iterative one; this remains true even if we consider a measure of efficiency that is computer and compiler independent.

The example referred to above is closely related to many combinatorial problems and involves the *generation of all permutations of a finite set.* In Section 2 we discuss various types of recursive permutation generators and present some results concerning their efficiency relative to iterative generators.

Since feasible solutions of many combinatorial problems are characterized by permutations, generators of permutations can be used in a straightforward way to solve such problems by *explicit enumeration* of all feasible solutions. We give some examples in Section 3, but it should be clear that this approach will solve only relatively small problems.

However, the advantages of a recursive approach carry through to forms of *implicit enumeration* as well. We illustrate this in Section 4 by presenting general frameworks for a popular type of implicit enumeration method known as *branch-and-bound*, in which again recursion plays a crucial role.

The material presented in this paper is adapted from [Lenstra & Rinnooy Kan 1975; Lenstra 1977].

2. GENERATION OF PERMUTATIONS

Methods for generating combinatorial configurations can often be classified as either *lexicographic* or *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. The relative advantages of minimum-change methods have been discussed in the literature: 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" [Ord-Smith 1971].

The very "cleanliness" [Lehmer 1964] of these combinatorial methods allows a proper demonstration of what we believe to be the advantages of a recursive approach to the implementation of enumerative methods.

The algorithms which are to be presented in this section 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 algorithms, the positions in which it differs from its predecessor. The actual procedure corresponding to "problem" has to be defined by the user to handle each configuration in the desired way.

Previously published iterative generators usually have been organized in such a way that each call generates one configuration from its predecessor only. This necessitates continual recomputation of the information that is needed to find the next configuration in the sequence. A mechanism for performing this kind of computations efficiently has been described in [Ehrlich 1973A]. We do feel, however, that much of the clarity of essentially recursive algorithms is lost within any iterative implemenation.

For generators of various types of combinatorial configurations such as subsets, combinations and permutations, we refer to [Wells 1971; Ehrlich 1973A; Even 1973; Lenstra & Rinnooy Kan 1975; Reingold *et al.* 1977]. Permutation generators have been surveyed in [Lehmer 1964; Ord-Smith 1970, 1971; Sedgewick 1977].

In Section 2.1 a minimum-change generator of permutations is presented. It produces a sequence in which each permutation is derived from its predecessor by *transposing two adjacent elements*. Its basic principles have been discovered by Steinhaus [Gardner 1974] and were rediscovered independently in [Trotter 1962] and [Johnson 1963]. Trotter's iterative algorithm was for a number of years the fastest permutation generator. A more efficient iterative implementation has been presented in [Ehrlich 1973B]; see also [Gries 1975; Dershowitz 1975].

The lexicographic generator of permutations in Section 2.2 produces all permutations π of the set $\{1,\ldots,n\}$ in such a way that $\pi(n)\pi(n-1)\ldots\pi(1)$ is an *increasing* n-*ary number*.

In Section 2.3 our recursive generators are compared to previously published procedures.

2.1. A minimum-change generator

Given a set $\{\pi^*(1),\ldots,\pi^*(n)\}$, we define an undirected graph $G(n)$ whose vertices are given by the $n!$ n-permutations of this set; (π,ρ) is an edge of G(n) if and only if π and ρ differ only in two neighboring 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 adjacent elements.

According to Steinhaus's method, 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 $\pi^*(n)$ 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 $\pi^*(n)$ with its left-hand neighbor. After that, $\pi^*(n)$ 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)-st n-permutation, and the n-1 next ones arise from successive transpositions of $\pi^*(n)$ with its right-hand neighbor. Then $\pi^*(n)$ 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 $\pi^* = (\pi^*(1),\ldots,\pi^*(n))$ and $\rho^* = (\pi^*(2),\pi^*(1),\pi^*(3),\ldots,\pi^*(n))$ respectively; they are adjacent and thus we have found a hamiltonian circuit in G(n).

Steinhaus's method can be described more formally by a sequence S(2) of n!-1 transpositions. Denoting the transposition of $\pi^*(i)$ and the h-th element in the current permutation of $\{\pi^*(1),\ldots,\pi^*(i-1)\}$ by $i\leftrightarrow h$, we define the transposition sequence S(i) recursively by

$$S(i) = S(i+1), i\leftrightarrow h_1, S(i+1), i\leftrightarrow h_2, \ldots, S(i+1), i\leftrightarrow h_{i-1}, S(i+1)$$

where

$$h_k = \begin{cases} k & \text{if } \pi^*(i) \text{ moves rightwards,} \\ i-k & \text{if } \pi^*(i) \text{ moves leftwards,} \end{cases}$$

and S(n+1) is empty. Figure 1 and Table 1(mc) show the graphs G(n) for $n \leq 4$ 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 [Lenstra Jr. 1973].

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

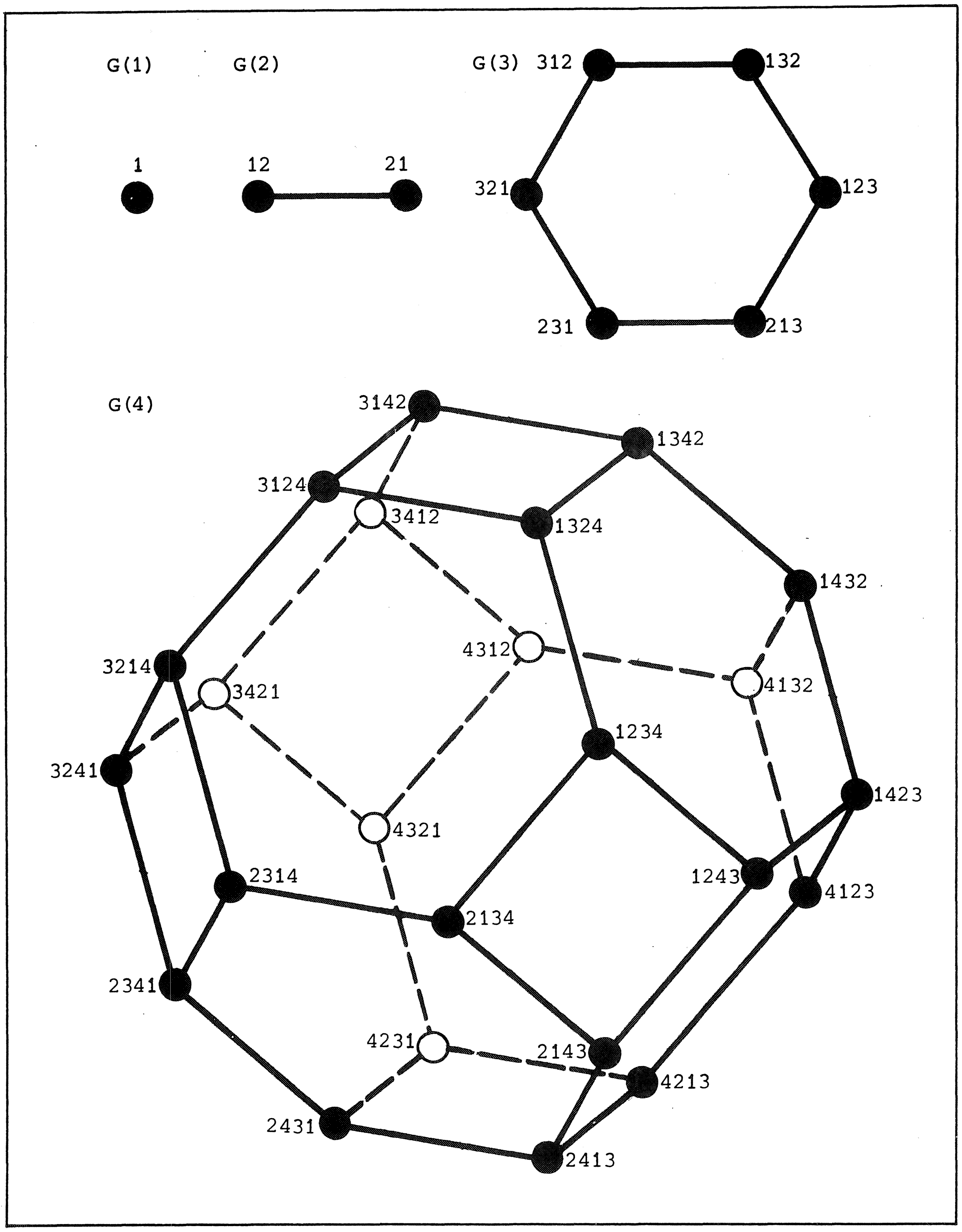

Figure 1 Graphs G(n).

TABLE 1. PERMUTATION SEQUENCES

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

```
procedure pm mc (problem,n,pi); value n,pi;
integer n; array pi; procedure problem;
begin   real pin; integer k,q; boolean array r[1:n];

        procedure rite(i); value i; integer i;
        if i < n then
        begin   boolean rj; real pii; integer ti,j;
                pii:= pi[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+i;
                        pi[k-1]:= pi[k]; pi[k]:= pii; problem(pi,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   pi[k-1]:= pi[k]; pi[k]:= pin; problem(pi,k-1)
                end
        end;

        procedure left(i); value i; integer i;
        if i < n then
        begin   boolean rj; real pii; integer ti,j;
                pii:= pi[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;
                        pi[k+1]:= pi[k]; pi[k]:= pii; problem(pi,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   pi[k+1]:= pi[k]; pi[k]:= pin; problem(pi,k)
                end;
                q:= 1
        end;

        pin:= pi[n]; q:= 0; for k:= 2 step 1 until n do r[k]:= false;
        problem(pi,0); if n > 1 then left(2)
end pm mc.
```

A call "pm mc (problem,n,π^*)" has the following effect:

if n = 1, then a call "problem(π^*,0)" is made; else

- a hamiltonian path in G(n) from π^* to $\rho^* = (\pi^*(2),\pi^*(1),\pi^*(3),\ldots,\pi^*(n))$ is traversed;

- in vertex π^* a call "problem(π^*,0)" is made;
- in each vertex π, reached by transposition of the elements in positions k and k+1, a call "problem(π,k)" is made.

The latter two assertions are clear from inspection. To prove the first one, we note that a call "rite(i)" ("left(i)") performs a series of i-1 transpositions of $\pi^*(i)$ with its right (left) neighbor, where the predicate r(i) indicates which direction has to be chosen. By induction on i we can show that a call "rite(i)" or "left(i)" generates all permutations in which the current order of $\pi^*(1),\ldots,\pi^*(i-1)$ is preserved, only transposing adjacent elements, whereas just before such a call and immediately after its execution, π and q have the following property:

the indices $(i,\ldots,n)$ can be rearranged as $(j_1,\ldots,j_q,j_{q+i},\ldots,j_n)$ with $j_1 > \ldots > j_q$, $j_{q+i} < \ldots < j_n$, such that $\pi(k) = \pi^*(j_k)$ for $k = 1,\ldots,q,q+i,\ldots,n$.

The first assertion now corresponds to the effect of a call "left(2)", which indeed activates the whole process. This completes the proof.

Using the integer q to determine the place of the transpositions is easier and more efficient than keeping track of the inverse permutation for that purpose, as is done in [Ehrlich 1973A; Ehrlich 1973B].

In order to add to the transparency and efficiency of the procedure, two simple constructions have been applied. First, we have distinguished between the leftward and rightward moves of the elements by means of two procedures calling themselves and one another. Further, the deepest level of the recursion has been written out explicitly. This device 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 n-th element, which is involved in (n-1)/n of the transpositions.

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; (π,ρ) is an edge of $G_H(n)$ if and only if π can be obtained from ρ by a single transposition of the elements in positions k and ℓ, where (k,ℓ) is an edge of H(n). One [Lenstra Jr. 1973] can prove that $G_H(n)$ *contains a hamiltonian circuit if and only if* H(n) *contains a spanning tree*. The "only if"-part is obvious; the "if"-part follows by an inductive argument. Note that the *transposition graph* H(n) of Steinhaus's method is a tree with edge set $\{(k,k+1) \mid k = 1,\ldots,n-1\}$.

2.2. A lexicographic generator

As mentioned before, the permutations π of the set $\{1,\ldots,n\}$ are ordered lexicographically when $\pi(n)\pi(n-1)\ldots\pi(1)$ is an increasing n-ary number. Table 1(lex) shows the sequence for $n = 4$.

Our *lexicographic generator of permutations* is given below. At each level of the recursion exactly one component of π is defined, and at the bottom a call "problem(π)" is made.

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

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

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

A call "pm lex (problem,n)" has the following effect:

- all permutations π of $\{1,\ldots,n\}$ are generated in lexicographic order;
- for each permutation π a call "problem(π)" is made.

To prove the first assertion, let us assume that, given a permutation π, a call "node(ℓ)" is made. It is easily checked that just before the ℓ calls "node(ℓ-1)" on the next level of the recursion, the then current permutation ρ is given by

$$\begin{aligned}\rho &= (\rho(1),\dots,\rho(k-1),\rho(k),\rho(k+1),\dots,\rho(\ell-1),\rho(\ell),\rho(\ell+1),\dots,\rho(n)) \\ &= (\pi(1),\dots,\pi(k-1),\pi(k+1),\pi(k+2),\dots,\pi(\ell),\pi(k),\pi(\ell+1),\dots,\pi(n)),\end{aligned}$$

for $k = \ell,\ell-1,\dots,1$. By induction on ℓ it can be shown that a call "node(ℓ)" generates all permutations π in which $\pi(\ell+1),\dots,\pi(n)$ remain unchanged, in increasing order, whereas just before such a call and immediately after its execution, π satisfies $\pi(1) > \pi(2) > \dots > \pi(\ell)$. The observation that the effect of a call "node(n)" corresponds to the first assertion completes the proof.

2.3. Computational experience

The algorithms presented in Sections 2.1 and 2.2 have been compared to ALGOL 60 versions of two minimum-change generators, mentioned in the introduction to Section 2.

Table 2 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 algorithm turns 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 instead of ALGOL 60 on the Electrologica X8 showed a nineteen-fold increase in speed for a recursive subset generator and a fourteen-fold increase for an iterative one. 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 [Ord-Smith 1971]. For recursive algorithms, evaluation of a is accomplished by the solution of recursive expressions. For Trotter's iterative algorithm only a lower bound can be given; it is not clear if a finite limit exists.

TABLE 2. COMPARISON OF FOUR PERMUTATION GENERATORS

generator	restrictions	time	a
[Trotter 1962; Ord-Smith 1971]	$n \geq 2$	91.3	≥ 7
[Ehrlich 1973B]	$n \geq 3$, $n \neq 4$	58.1	3
pm mc	$n \geq 1$	42.9	3
pm lex	$n \geq 1$	92.4	6.44

time : CPU seconds on an Electrologica X8 for $n = 8$.

a : average array access (in the limit).

3. EXPLICIT ENUMERATION

Generators of combinatorial configurations 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.

An an illustration we will show how generators of permutations can be used to solve sequencing problems P of the form

$$\min_\pi\{f_P(\pi)\}$$

where π runs over all permutations of $\{1,\ldots,n\}$. This formulation includes several well-known combinatorial optimization problems. Recall that the criterion function of the *quadratic assignment problem* (QAP) is given by

$$f_{QAP}(\pi) = \sum_{i=1}^n \sum_{j=1}^n c_{\pi(i)\pi(j)} d_{ij}$$

where (c_{ij}) and (d_{ij}) are nonnegative $n\times n$-matrices. If we take $d_{ij} = 1$ for $i > j$, $d_{ij} = 0$ otherwise, we obtain the *acyclic subgraph problem* (ASP).

Analogously, the choice $d_{12} = d_{23} = \dots = d_{n-1,n} = d_{n1} = 1$, $d_{ij} = 0$ otherwise, leads to the *traveling salesman problem* (TSP), that is called *symmetric* if $c_{ij} = c_{ji}$ for all i,j.

If we define the *reflection* of π by $\bar{\pi} = (\pi(n),\dots,\pi(1))$, it is obvious that $f_{ASP}(\bar{\pi}) = \sum_{i \neq j} c_{ij} - f_{ASP}(\pi)$ for the ASP and $f_{TSP}(\bar{\pi}) = f_{TSP}(\pi)$ for the symmetric TSP. It follows that for these two problems it suffices to enumerate a *reflection-free* set of permutations. Further, since $f_{TSP}((\pi(k+1),\dots,\pi(n),\pi(1),\dots,\pi(k))) = f_{TSP}(\pi)$ for any k, we may fix one of the components of π 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* [Harada 1971; Read 1972; Roy 1973].

In the minimum-change generator of permutations, discussed in Section 2.1, the elements $\pi^*(1)$ and $\pi^*(2)$ are transposed half-way. If a permutation π is generated before this transposition, then its reflection $\bar{\pi}$ occurs thereafter. Hence the first $n!/2$ permutations form a relection-free set (*cf.* [Roy 1973]). Generally, the $n!/m!$ permutations preserving the original order of $\pi^*(1),\dots,\pi^*(m)$ can be generated by a simple adaptation of "pm mc":

```
procedure pp mc (problem,n,m,pi); ...;
begin   ...
        ...; if n > m then left(m+1)
end pp mc1.
```

The above sequencing problems may now be solved by calls

pm mc (qap,n,π),

pp mc (asp,n,2,π),

pp mc (tsp,n-1,<u>if</u> symmetric <u>then</u> 2 <u>else</u> 1,π),

where "qap", "asp" and "tsp" are procedures which compute the changes occurring in the criterion functions of these problems.

4. IMPLICIT ENUMERATION

The permutation generator presented in Section 2.2 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 (!) [Tompkins 1956]. 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 [Rinnooy Kan *et al.* 1975] underlines the applicability of recursive programming to implicit enumeration methods of the *branch-and-bound* type in general.

In this section we present a quasi-ALGOL description of branch-and-bound procedures, indicating in which case a recursive approach might be suitable. For a formal characterization of branch-and-bound procedures, we refer to the axiomatic framework in [Mitten 1970] and its correction in [Rinnooy Kan 1976]; see also [Agin 1966; Balas 1968] for analyses of the case in which the set of feasible solutions is finite and [Kohler & Steiglitz 1974] for the case of permutation problems. Some standard examples of branch-and-bound methods have been surveyed in [Lawler & Wood 1966].

Suppose then, that a *set* X *of feasible solutions* and a *criterion function* $f: X \to \mathbb{R}$ are given, and define the *set* X^* *of optimal solutions* by

$$X^* = \{x^* | x^* \in X, f(x^*) = \min\{f(x) | x \in X\}\}.$$

A branch-and-bound procedure to find an element of X^* can be characterized as follows.

- Throughout the execution of the procedure, the *best solution* x^* *found so far* provides an *upper bound* $f(x^*)$ on the value of the optimal solution.
- A *branching rule* b associates to $Y \subset X$ a family $b(Y)$ of subsets such that $\cup_{Y' \in b(Y)} Y' \cap X^* = Y \cap X^*$; the subsets Y' are the *descendants* of the *parent* subset Y. This rule only has to be defined on a class $\mathcal{X}$ with $X \in \mathcal{X}$ and $b(Y) \subset \mathcal{X}$ for any $Y \in \mathcal{X}$.
- A *bounding rule* lb: $\mathcal{X} \to \mathbb{R}$ provides a *lower bound* $lb(Y) \le f(x)$ for all $x \in Y \in \mathcal{X}$. *Elimination* of Y occurs if $lb(Y) \ge f(x^*)$.
- A *predicate* $\xi: \mathcal{X} \to \{\underline{true},\underline{false}\}$ indicates if during the examination of Y (*e.g.* during the calculation of lb(Y)) a feasible solution x(Y) is generated which has to be evaluated. *Improvement* of x^* occurs if $f(x^*) > f(x(Y))$.

- 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, two are suitable for recursive implementation. To illustrate this point, we shall now present three general procedures:

- "bb jumptrack" implements a *breadth-first search* where a subset with minimal lower bound is selected for examination; this type of tree search is known as *frontier search*;
- "bb backtrack1" implements a *depth-first search* where the descendants of a parent subset are examined in an arbitrary order; this type is known as *newest active node search*;
- "bb backtrack2" implements a *depth-first search* where the descendants are chosen in order of nondecreasing 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 ":z$\in$" in the statement "s:z$\in$ S" to mean that s:= s^* with $z(s^*) = \min\{z(s) | s \in S\}$; hence, ":$\in$" indicates an arbitrary choice.

```
procedure bb jumptrack (X,f,$x^*$,$\mathcal{b}$,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 $x^*$:f$\in$ {$x^*$,x(X)};
        if LB(X) $\geq$ f($x^*$) then na:= 1 else $\mathcal{Y}$:= {X};
        while $\mathcal{Y} \neq \emptyset$ do
        begin   Y:LB$\in$ $\mathcal{Y}$;
                nb:= nb+1; $\mathcal{B}$:= $\mathcal{b}$(Y); $\mathcal{Y}$:= ($\mathcal{Y}$-{Y})$\cup\mathcal{B}$;
                while $\mathcal{B} \neq \emptyset$ do
                begin   Y':$\in$ $\mathcal{B}$; $\mathcal{B}$:= $\mathcal{B}$-{Y'};
                        LB(Y'):= lb(Y'); if $\xi$(Y') then $x^*$:f$\in$ {$x^*$,x(Y')}
                end;
                $\mathcal{Y}'$:= {Y'|Y' $\in \mathcal{Y}$, LB(Y') $\geq$ f($x^*$)};
                na:= na+|$\mathcal{Y}'$|; $\mathcal{Y}$:= $\mathcal{Y}$-$\mathcal{Y}'$
        end
end bb jumptrack.
```

```
procedure bb backtrack1 ($X,f,x^*,b,lb,\xi,na,nb$);
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 $x^*:f\in \{x^*,x(Y)\}$;
                if $LB \ge f(x^*)$ then na:= na+1 else
                begin   nb:= nb+1; $\mathcal{B}:= b(Y)$;
                        while $\mathcal{B} \ne \emptyset$ do
                        begin   $Y':\in \mathcal{B}$; $\mathcal{B}:= \mathcal{B}-\{Y'\}$;
                                if $LB < f(x^*)$ then node(Y')
                        end
                end
        end;

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

procedure bb backtrack2 ($X,f,x^*,b,lb,\xi,na,nb$);
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; $\mathcal{Y}:= \mathcal{B}:= b(Y)$;
                while $\mathcal{B} \ne \emptyset$ do
                begin   $Y':\in \mathcal{B}$; $\mathcal{B}:= \mathcal{B}-\{Y'\}$;
                        LB(Y'):= lb(Y'); if $\xi(Y')$ then $x^*:f\in \{x^*,x(Y')\}$
                end;
                while $\mathcal{Y} \ne \emptyset$ do
                begin   $Y':LB\in \mathcal{Y}$; $\mathcal{Y}:= \mathcal{Y}-\{Y'\}$;
                        if $LB(Y') \ge f(x^*)$ then na:= na+1 else node(Y')
                end
        end;
        na:= nb:= 0;
        LB(X):= lb(X); if $\xi$(X) then $x^*$:f$\in$ {$x^*$,x(X)};
        if LB(X) $\geq$ f($x^*$) then na:= 1 else node(X)
end bb backtrack2.
```

Anyone familiar with branch-and-bound will have noticed that the above descriptions provide only 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 due to *elimination criteria* based on dominance rules or feasibility considerations. In a minor (and in our experience quite successful) variation on "bb backtrack1", the descendants Y' of a parent subset Y are not chosen arbitrarily, but according to some heuristic, *e.g.* preliminary lower bounds lb'(Y') with lb(Y) $\leq$ lb'(Y') $\leq$ lb(Y'). Many similar variations are possible but do not affect the basic mechanisms outlined above.

A main characteristic of many branch-and-bound procedures is the unpredictability of their computational behavior. Their worst-case performance may be close to explicit enumeration, and no satisfying analyses of average-case behavior have been presented up to now [Karp 1976; Lenstra & Rinnooy Kan 1978]. Extensive computational experience seems to be the only way to test their quality. Branch-and-bound should not be used before one feels sure that the complexity of the problem is such that no better approach can be found. However, this is often the case, and methods of branch-and-bound are widely used for solving combinatorial optimization problems.

From our experience with the implementation of branch-and-bound algorithms we may conclude that again the recursive approach produces transparent procedures, in which much administrative work is taken over by the compiler without a noticeable negative effect on overall efficiency.

ACKNOWLEDGMENTS

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

REFERENCES

N. AGIN (1966) Optimum seeking with branch-and-bound. *Management Sci.* 13, B176-185.

E. BALAS (1968) A note on the branch-and-bound principle. *Operations Res.* 16,442-445,886.

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

J.R. BITNER, G. EHRLICH, E.M. REINGOLD (1976) Efficient generation of the binary reflected Gray code and its applications. *Comm. ACM* 19,517-521.

N. DERSHOWITZ (1975) A simplified loop-free algorithm for generating permutations. *BIT* 15,158-164.

G. EHRLICH (1973A) Loopless algorithms for generating permutations, combinations and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20,500-513.

G. EHRLICH (1973B) Algorithm 466, Four combinatorial algorithms. *Comm. ACM* 16,690-691.

S. EVEN (1973) *Algorithmic Combinatorics*, Macmillan, London.

M. GARDNER (1974) Some new and dramatic demonstrations of number theorems with playing cards. *Sci. Amer.* 231,122-125.

D. GRIES (1975) Recursion as a programming tool. Technical Report 234, Department of Computer Science, Cornell University, Ithaca.

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

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

R.M. KARP (1976) The probabilistic analysis of some combinatorial search algorithms. In: J.F. TRAUB (ed.) (1976) *Algorithms and Complexity: New Directions and Recent Results*, Academic Press, New York, 1-19.

W.H. KOHLER, K. STEIGLITZ (1974) Characterization and theoretical properties of branch-and-bound algorithms for permutation problems. *J. Assoc. Comput. Mach.* 21,140-156.

E.L. LAWLER, D.E. WOOD (1966) Branch-and-bound methods: a survey. *Operations Res.* 14,699-719.

D.H. LEHMER (1964) The machine tool of combinatorics. In: E.F. BECKENBACH (ed.) (1964) *Applied Combinatorial Mathematics*, Wiley, New York, 5-31.

H.W. LENSTRA, JR. (1973) Private communications.

J.K. LENSTRA (1977) *Sequencing by Enumerative Methods*, Mathematical Centre Tracts 69, Mathematisch Centrum, Amsterdam.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1975) A recursive approach to the generation of combinatorial configurations. Report BW50, Mathematisch Centrum, Amsterdam.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1978) On the expected performance of branch-and-bound algorithms. *Operations Res.* 26,347-349.

L.G. MITTEN (1970) Branch-and-bound methods: general formulation and properties. *Operations Res.* 18,24-34.

R.J. ORD-SMITH (1970) Generation of permutation sequences: part 1. *Comput. J.* 13,152-155.

R.J. ORD-SMITH (1971) Generation of permutation sequences: part 2. *Comput. J.* 14,136-139.

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

E.M. REINGOLD, J. NIEVERGELT, N. DEO (1977) *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, N.J.

A.H.G. RINNOOY KAN (1976) On Mitten's axioms for branch-and-bound. *Operations Res.* 24,1176-1178.

A.H.G. RINNOOY KAN, B.J. LAGEWEG, J.K. LENSTRA (1975) Minimizing total costs in one-machine scheduling. *Operations Res.* 23,908-927.

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

R. SEDGEWICK (1977) Permutation generation methods. *Comput. Surveys* 9, 137-164,314.

C. TOMPKINS (1956) Machine attacks on problems whose variables are permutations. *Proc. Sympos. Appl. Math.* 6, Amer. Math. Soc., Providence, 195-211.

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

M.B. WELLS (1971) *Elements of Combinatorial Computing*, Pergamon, Oxford.