ON TOPOLOGICAL SORTING AND OTHER RESTRICTED PERMUTATION PROBLEMS

Alan David Kalvin

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg in partial fulfilment of the requirements for the degree of Master of Science

Johannesburg 1980

ABSTRACT

This dissertation studies the problem of generating and enumerating all topological sorting arrangements of a partially ordered finite set.

Three algorithms that generate topological sortings are analysed and compared. An algorithm for enumerating topological sortings is discussed, and methods for enumerating the topological sortings of some special types of partially ordered sets are presented.

The topological sorting problem is a restricted permutation problem, that is a problem concerned with the study of permutations that satisfy some given set of restrictions. Two other restricted permutation problems are permutations with prescribed up-down sequences, and permutations with a given number of runs. Both these problems are shown to be reducible to topological sorting.

DECLARATION

I, Alan David Kalvin, declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted for any other degree or examination in any other university.

.....

30 January 1981

For my parents

CONTENTS PAGE

PREFACE

I wish to record my sincere gratitude to Dr. Yaakov L. Varol and Dr. Judy M. Bishop for supervising this dissertation. In particular, special thanks are due to Dr. Varol who introduced me to the subject of topological sorting. I am also appreciative of the assistance that I received from Dr. K. J. Danhof (from the University of Southern Illinois at Carbondale) during his visit to the University of the Witwatersrand. I am grateful to the Council for Scientific and Industrial Research for its generous financial support. Finally, I wish to thank my mother, Jean Kalvin, who devoted much of her time to typing this dissertarion.

The text of this dissertation has been prepared on a Zilog microcomputer running under the U.C.S.D. Pascal System II.0 with the aid of the U.C.S.D. Text Editor, and programs written by staff and students of the Computer Science Division of the Department of Applied Mathematics, University of the Witwatersrand.

CHAPTER 1 - BASIC CONCEPTS

1.1 Preliminaries

Let S be a set, then any subset R of S x S is called a binary relation on S. If the pair $(x,y) \in R$ we write xRy, and if $(x,y) \notin R$ we write xRy. If xRy and yRx imply x = y for all x,y in S, then R is called a precedence relation and is said to be antisymmetric. If xRy and yRz imply xRz for all x,y,z in S, R is said to be transitive; if xRx for all x in S, then R is said to be reflexive and if xRx for all x in S, then R is said to be irreflexive. If $S = (x_1, x_2, \ldots, x_n)$, then the relation R can be represented by the n x n matrix M where $M_{ij} = 1$ if $x_i R x_j$ and $M_{ij} = 0$ otherwise. M is called the adjacency matrix of R.

A partially ordered set or poset (S,R) is a set together with a binary relation R which is antisymmetric, transitive and reflexive. R is called a partial ordering on S, and x is said to precede y (in the partial ordering) if xRy. If S is finite and non-empty, (and we shall always assume so), the poset (S,R) can be represented pictorially by a Hasse diagram in which each element of S is represented by a node, and there is a falling line joining x to y if and only if xRy and there is no z such that xRz and zRy, for all x,y,z in S. For example Figure 1.1 shows the Hasse diagram of the poset (S,R) where S = (1, 2, 3, 4, 6, 12) and xRy if x divides y.

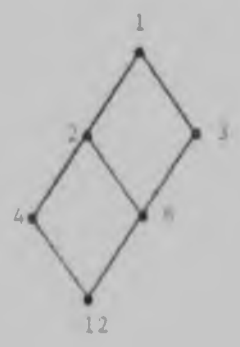


Figure 1.1 A Hasse diagram of a poset

A directed graph $G = (V,E)$ is a finite, non-empty set V together with a binary relation E on V. The elements of V and E are the vertices and edges of G respectively. We say that y is adjacent to x if $(x,y) \in E$ and the set A_x of all vertices adjacent to x is called the adjacency set for x. A sequence of edges of the form (v_1,v_2), $(v_2,v_3),\ldots,(v_{k-1},v_k)$ is called a path of length k-1 in G. As a special case, a single vertex denotes a path of length 0 from itself to itself. If there is a path from x to y then x is said to reach y. A cycle is a path of length at least 1 which begins and ends at the same vertex. If $G = (V,E)$ is a directed graph and E is a transitive precedence relation, then G will have no cycles and it is said to be acyclic. If $E' \subset E$, then the directed graph $G' = (V,E')$ is called a spanning subgraph of $G = (V,E)$.

Let $G = (V,E)$ be an acyclic directed graph. We define $G^* = (V,E^*)$ such that $(x,y) \in E^*$ if and only if y is reachable from x in G. G^* is called the (reflexive and) transitive closure of G. It follows that G^* is a poset - since E^* satisfies the required properties - and any spanning subgraph of G^* which preserves reachability can represent the poset (V,E^*). In particular, the Hasse diagram of the poset corresponds to the minimal spanning subgraph of G^* with that property. In this dissertation we shall refer to these structures, either by posets or acyclic directed graphs representing them.

1.2 The topological sorting problem

Given a partial ordering R on a finite non-empty set S of size n, the problem of topological sorting is to find a permutation $p_1p_2\cdots p_n$ of the elements of S such that i appears to the left of j if iRj. This permutation is called a solution to the topological sorting problem, or simply a topological sorting. The topological sorting problem is equivalent to each of the following:

(1) Given a partial ordering R on a finite non-empty set S, find a way to permute the rows and columns of the adjacency matrix M of R so that M becomes an upper-triangular matrix.

(2) Given a partial ordering on a finite, non-empty set, find a linear (total) ordering in which the partial ordering can be embedded.

(3) Given an acyclic directed graph G with n vertices, find a way to label these vertices with the integers $\{1,2,\ldots,n\}$ such that the label of vertex x is less than the label of vertex y for each edge (x,y) in G.

(4) Given an acyclic directed graph G, find a way to arrange the vertices in a straight line so that all arcs go from left to right.

In our study of topological sorting the reflexivity of partial orderings is of no importance, and when convenient we shall ignore this requirement. (Knuth [1968, pp. 258-9] in fact defines a partial ordering to be a binary relation that is either (a) antisymmetric, transitive and reflexive, or (b) antisymmetric, transitive and irreflexive).

Topological sorting is a process which is of potential use whenever we have a problem involving a partial ordering. Following are some examples of topological sorting:

(1) Constructing a glossary of terms

If the definition of a word x depends on that of a word y, we denote this by the relation xRy. Topological sorting of the words means arranging them into an order so that no word is used before it has been defined.

(2) Scheduling activities of a project

A project consists of a set of activities that must be performed. If activity x must be completed before activity y can commence, we write xRy. Topological sorting means the arrangement of the activities in such an order that upon initiation of each activity all its prerequisite activities have been completed. (Several sophisticated techniques such PERT, RAMPS and CPM have been developed to evaluate such activity models - see for example Horowitz and Sahni [1976]).

(3) Drawing up a university curriculum

In a university curriculum, certain courses must be taken before others since they rely on the material presented in

the prerequisites. If a course x is a prerequisite for a course y, we write xRy. Topological sorting means arranging the courses in such an order that no course lists a later course as a prerequisite.

(4) Declaration of procedures in a computer program

In a program, some procedures may contain calls to other procedures. If a procedure x is called by a procedure y, we write xRy. Topological sorting implies the arrangement of procedure declarations in such a way that there are no forward references.

There is essentially one method to do topological sorting, and it appears in the literature in a number of different forms.

Firstly, let R be a partial ordering on a finite non-empty set S. In order to sort S topologically we start by taking an element which is not preceded by any other element in the partial ordering. This element may be placed first in the output. Now we remove this element from S; the resulting set is again partially ordered, and the process can be repeated until the whole set has been sorted. The only way in which this method could fail would be if there were a non-empty partially ordered set in which every element was preceded by another; for in such a case there would be nothing to do. But if every element is preceded by another, we could construct an arbitrarily long sequence $p_1, p_2, p_3, \cdots$ in which $p_{j+1} R p_j$. Since S is finite, we must have $p_j = p_k$ for some $j < k$, but this implies that $p_k R p_{j+1}$ contradicting the antisymmetric property of the partial ordering. This form of the topological sorting method is given by Kahn [1962], Knuth [1968], Knuth and Szwarcfiter [1973] and Wirth [1976]. (It is easy to see that this is in fact the only way to sort topologically, since if $p_1 p_2 \cdots p_n$ is a topological sorting of S, then p_1 must be an element without predecessors, and p_2 must have no predecessors other than perhaps p_1, and so on).

A second way to describe this topological sorting technique is given in Aho, Hopcroft and Ullman [1974], and Horowitz and Sahni [1976]. Let G = (V,E) be a directed acyclic graph. Select a vertex with no incoming edge. This vertex may be placed first in the output. Delete the vertex from the graph along with all outgoing edges. The result-

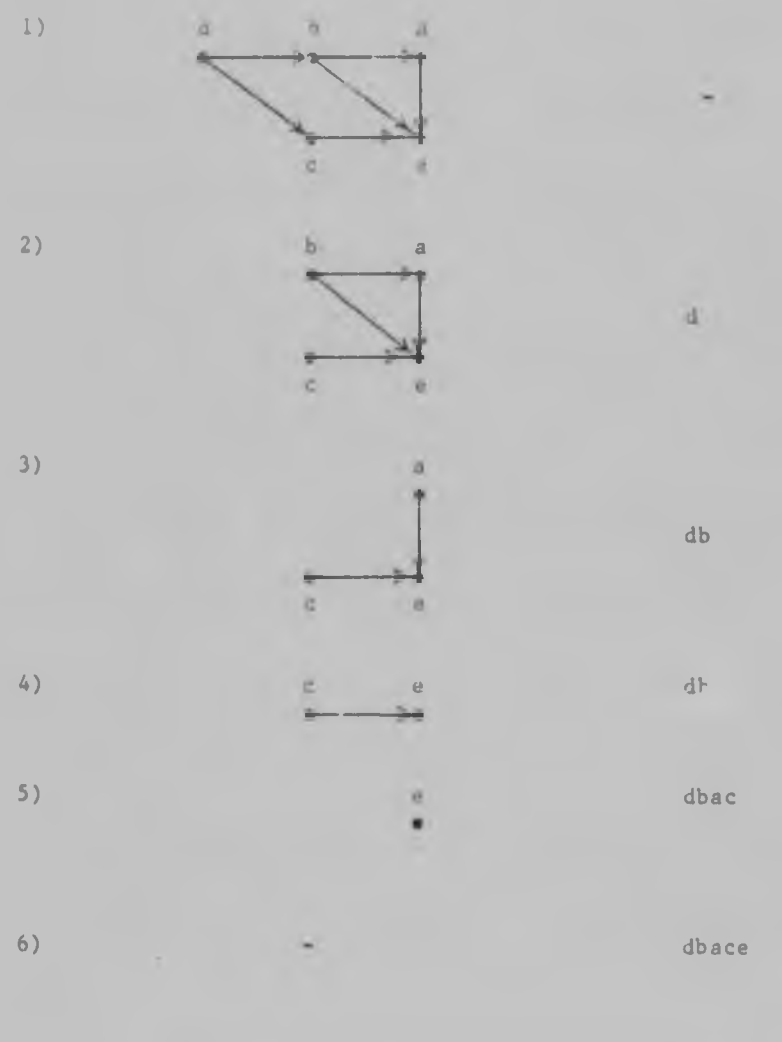


Figure 1.2 An example of topological sorting

ing graph is again acyclic and the process can be repeated until every vertex has been output. An example of this process is shown in Figure 1.2. As before, his method can fail only if there is an acyclic graph in which every vertex has an incoming edge. However this situation can never occur since it would imply that there was a cycle in the graph. A variation of this form of the topological sorting method is given by Tarjan [1974], and Reingold, Nievergelt and Deo [1977]. A depth-first spanning forest of the acyclic directed graph G = (V,E) is obtained by performing a depth-first search on G. The vertices are selected in such an order that every

vertex is selected before any of its descendants in the forest.

Another form of this method, described in terms of triangulating the adjacency matrix of a partial ordering is presented by Nijenhuis and Wilf [1975].

1.3 Scope of this dissertation

We shall be concerned primarily with the problem of finding all topological sortings of a given poset. This problem can be viewed as a restricted permutation problem, that is a problem concerned with the study of permutations that are consistent with some given set of restrictions - in the case of topological sorting these restrictions are, of course, the partial ordering imposed on the set of objects in question.

In Chapter 2 we discuss various algorithms for generating all topological sortings and the related problem of enumerating topological sortings is considered in Chapter 3.

Chapter 4 deals with two other restricted permutation problems, namely permutations with prescribed up-down sequences and permutations with a given number of runs.

In Chapter 5 we consider possible areas for future research.

CHAPTER 2 - ALGORITHMS FOR GENERATING TOPOLOGICAL SORTINGS

2.0 Introduction

The number of permutations of n objects consistent with a partial ordering is often small compared with n! - a single relation reduces the number of permissible permutations to n!/2. It is grossly inefficient to generate the topological sortings of n objects by generating all n! permutations and then testing each one for consistency. We therefore consider more efficient methods of generating topological sortings.

2.1 Backtracking algorithms

A backtracking (or tree programming) algorithm for generating all topological sortings of n objects is a procedure where at every stage we consider a subproblem of the form "find all ways to complete a given partial permutation $p_1p_2\cdots p_{i-1}$ to a topological sorting $p_1p_2\cdots p_n$". The general method is to branch on all possible choices of p_i.

Such a backtracking algorithm will have the basic structure shown in Figure 2.1. TOPSORTS(i) will generate all permutations of the set $\{1,2,\ldots,n\}$ consistent with some partial ordering which begin with $p_1p_2\cdots p_{i-1}$. On entry (and exit) to TOPSORTS(i), $S = \{1,2,\ldots,n\}-\{p_1,p_2,\ldots,p_{i-1}\}$ is the set of integers not yet considered for the current configuration being generated, and M_i consists of all integers in S which can be chosen as p_i without contradicting the restrictions imposed by the given partial ordering.

2.1.1 Wells' algorithm

The algorithm given by Wells [1971], is shown in Figure 2.2. The iteration quantifier "for $p_i \in M_i$ do" (line L8) selects the elements of the set M_i in ascending numerical order and so the topological sortings will be generated in numerical (lexicographical) order.

In its original form, Wells' algorithm generates a new topological

```
procedure TOPSORTS(i):
begin if i < n+1 then
        for p_i ∈ M_i do
          begin S:= S ~ {p_i};
                TOPSORTS(i+1);
                S:= S ∪ {p_i};
          end;
      endif;
return;
```

Figure 2.1 The basic backtracking algorithm

sorting every time it is called. To process each sorting (for what ever purpose it may be required) a main program is used to call this procedure. A simple modification to the original procedure (which does not affect time/space complexities or the basic structure of the algorithm) results in the generation procedure becoming the "main program" and each topological sorting is processed as it is generated. To indicate this clearly we shall assume a macro called PROCESS which will be invoked each time a topological sorting is generated. We shall explicitly include calls of the form PROCESS(P), where P represents the topological sorting to be processed, in all the generation algorithms discussed.

In Wells' algorithm, the partial ordering is represented by the adjacency sets A_j, $j = 1,2,\ldots,n$ and B_j is the transpose of A_j (i.e. $k \in A_j \iff j \in B_k \iff jRk$). The property of reflexivity is ignored (i.e. we assume $x \notin A_x$) and we can view A_j and B_j as the sets of elements which must respectively follow and precede element j in a topological sorting.

The algorithm uses significant impasse detection to compute dynamically the set M_i of elements which can appear in position i in order to extend a partial permutation $p_1p_2\cdots p_{i-1}$ into a topological sorting $P = p_1p_2\cdots p_n$. First of all, note that element j cannot occupy positions $1,2,\ldots,|B_j|$ or positions $n-|A_j|+1, n-|A_j|+2,\ldots,n$ in a topological sorting since these positions are reserved respectively for those elements which must precede and follow j. The precomputed

```
      procedure GENERATE_W(A,B,n):
      begin S:= {1,2,...,n};
        for j:= 1 to n do
          begin Q_j:= {|B_j|+1,|B_j|+2,...,n-|A_j|} end;
        TOPSORTS_W(1);
      end;

      procedure TOPSORTS_W(i):
L1.   begin if i = n+1 then PROCESS(P);
L2.         else M_i:= { }; t:= 0;
L3.              for j ∈ S such that i ∈ Q_j and B_j ∩ S = { } do
L4.                begin if i = n-|A_j|
L5.                      then if t = 1 then return.
L6.                                    else t:= 1; M_i:= {j};
                         endif;
                   endif;
L7.                if t = 0 then M_i:= M_i ∪ {j} endif;
                 end;
L8.              for p_i ∈ M_i do
L9.                begin S:= S - {p_i};
L10.                     TOPSORTS(i+1);
L11.                     S:= S ∪ {p_i};
                   end;
          endif;
      return;
```

Figure 2.2 Wells' generating algorithm

set Q_j gives the positions not so blocked to j. Also, an element j is available for use in position i only if all elements which must precede it already exist in the partial permutation. The iteration quantifier at line L3 restricts the elements of M_i according to these observations. One further impasse detection is incorporated into this loop which constructs M_i. The test at line L4 asks if this is the last chance for placing element j; if so, it becomes the sole element of M_i (line L6). If position i is the last chance for yet

another element (test at line L5 is successful), an impasse exists. In this case, the procedure TOPSORTS_W(i) terminates, control is passed back to TOPSORTS_W(i-1), and the next element of M_{i-1} is selected as p_{i-1}.

Wells' algorithm will generate topological sortings correctly but, as we shall now show, it is not particularly efficient. Firstly, it has the limitation of being applicable only to sets of elements on which antisymmetric, transitive binary relations are defined. Given an arbitrary binary relation, the construction of the required transitive closure and/or the detection of symmetries (cycles) in the relation would require an additional procedure of $O(n^3)$ time complexity. (We shall see that the other generation algorithms discussed do not have this limitation). For a binary relation where a high proportion of the elements are related to one another, the number of topological sortings will be small, and an $O(n^3)$ closure procedure may require more time than the actual generation procedure!

Another reason for inefficiency in the algorithm is the excessive use of impasse detection; most of the work involved in testing for impasses (and in the precomputations required for this testing) is unnecessary. Consider the test at line L5. If this test is successful, then position i is the last chance for placing two distinct elements, so an impasse exists and the procedure "backtracks". If position i is the last chance for placing an element x, then at this stage, the set S of elements yet to be assigned a position in the current partial permutation consists of x and all elements which must follow x, i.e. $S = A_x \cup \{x\}$. If position i is also the last chance for placing another element y, then similarly, $S = A_y \cup \{y\}$. This implies that $x \in A_y$ and $y \in A_x$ (i.e. yRx and xRy), contradicting the antisymmetry of the partial ordering. So this situation can never exist and line L5 can be removed.

If position i is the last chance for placing some element x, lines L4, L6 and L7 are used to ensure that x becomes the sole element of M_i. Now the condition "$B_j \cap S = \{\ \}$" in the quantifier at line L3 will also ensure that element x is the sole element of M_i, since at this stage, $S = A_x \cup \{x\}$ and therefore for any $y \in S$, $y \neq x$, $y \in A_x$ and so $x \notin (B_y \cap S)$ which means that y cannot be selected as a member of M_i. So lines L4, L6 and L7 are unnecessary.

As we have noted, element j cannot occupy positions $1,2,\ldots,|B_j|$ or positions $n-|A_j|+1, n-|A_j|+2,\ldots,n$ and the condition "$i \in Q_j$" in line L3 is used to restrict the choice of elements for M_i accordingly. This condition is however, redundant. Since the condition "$B_j \cap S = \{\ \}$" ensures that element x will be the sole element of M_i when i is the last chance for placing x, it also ensures that element x will never be considered as a possible candidate for positions $n-|A_x|+1$, $n-|A_x|+2,\ldots,n$. Furthermore, "$B_j \cap S = \{\ \}$" will ensure that element x is never considered as a possible candidate for positions $1,2,\ldots,|B_x|$, since if we are forming the set M_i for any $i \in \{1,2,\ldots,|B_x|\}$, the set S, at this stage, will contain at least $|B_x|+1-i$ elements of B_x, i.e. $B_x \cap S \neq \{\ \}$. So we can simplify line L3 to

L3. for $j \in S$ such that $B_j \cap S = \{\ \}$ do

By removing the unnecessary impasse tests we have eliminated the need for the sets Q_j, $j = 1,2,\ldots,n$ and hence we do not need the adjacency sets A_j any longer.

Note that the procedure TOPSORTS_W is called most often when $i = n$, at which stage S contains exactly one element, and so it can be greatly simplified for this case. By combining this fact with the elimination of all that we have proved to be redundant in Wells' algorithm we obtain the algorithm of Figure 2.3. As Wells [1980] states, this algorithm represents a significant improvement to, and a generalization of, his algorithm. It can be seen that when using this improved algorithm, at each stage of the process of constructing a topological sorting $p_1p_2\cdots p_n$, element j is considered for selection as p_i only if $B_j \cap S = \{\ \}$, that is only if all elements that precede j in the partial ordering have already been selected. This is exactly the method discussed in Chapter 1 for finding a single topological sorting, a fact that is not surprising since, as we noted, it is the only method for finding a topological sorting. The algorithm in Figure 2.3 generates all topological sortings by using this method in conjunction with backtracking.

This algorithm is more general than Wells' algorithm in the sense that it can be applied to any arbitrary binary relation R defined on $\{1,2,\ldots,n\}$. Remember that Wells' algorithm is applicable only to binary relations on $\{1,2,\ldots,n\}$ that are antisymmetric, transitive

```
     procedure GENERATE_I(B,n):
     begin S:= {1,2,...,n};
           TOPSORTS_I(1);
     end;


     procedure TOPSORTS_I(i):
L1.  begin if i = n then p_n:= sole element of S; PROCESS(P);
L2.        else for p_i ∈ S such that B_pi ∩ S = { } do
L3.               begin S:= S - {p_i};
L4.                     TOPSORTS(i+1);
L5.                     S:= S ∪ {p_i};
                  end;
           endif;
     return;
```

Figure 2.3 An improved generating algorithm

(and reflexive) - i.e. partial orderings. The only reason that the property of transitivity is required is that the sizes of the adjacency sets A_j and their transposes B_j, $j = 1,2,\ldots,n$ are needed for detecting impasses. The algorithm in Figure 2.3, on the other hand, does not employ these impasse detection techniques. So firstly, any binary relation R whose transitive closure is a partial ordering can be used in this algorithm instead of the partial ordering itself.

Secondly, if R is a binary relation whose transitive closure R* is not a partial ordering, then R contains one or more cycles. Suppose for the moment that R contains exactly one cycle; this means there is some integer $k > 1$ and a subset $\{c_1,c_2,\ldots,c_k\}$ of $\{1,2,\ldots,n\}$ such that $c_1Rc_2, c_2Rc_3, \ldots, c_{k-1}Rc_k$ and c_kRc_1, and therefore $c_1R^*c_k$ and $c_kR^*c_1$ and so there is no possible way to sort $\{1,2,\ldots,n\}$ topologically. Now using the algorithm in Figure 2.3, when TOPSORTS_I(i) is called for $i = n+1-k$, we have $S = \{c_1,c_2,\ldots,c_k\}$. Since none of the c_i's satisfies the requirement to be selected as p_{n+1-k}, the loop of lines L2, L3 and L4 is never executed, so TOPSORTS_I(i) is never called for $i = n+2-k, n+3-k, \ldots, n$ and the algorithm correctly terminates without generating any permutations. Similarly if R con-

tains more than one cycle then TOPSORTS_I(i) will not be called for i > n+1-m, where m is the total number of elements lying on cycles.

It is interesting to note that if the condition "$B_{p_i} \cap S = \{\ \}$" is removed from the iteration quantifier in line L2 of this algorithm, what we are left with is simply an algorithm which generates, in increasing lexicographical order, all the n! permutations of the set $\{1,2,\ldots,n\}$. Wells [1971] in fact presents this permutation generation algorithm, and although he does not apply it to the topological sorting problem, he makes the point that this basic permutation generation algorithm can be readily modified to generate permutations subject to various restrictions by including the required conditions in the iteration quantifier.

2.1.2 The Knuth-Szwarcfiter algorithm

Knuth and Szwarcfiter [1974] present a generating algorithm that has the identical basic structure of the algorithm in Figure 2.3. Like the algorithm in Figure 2.3 it generates all topological sortings by combining backtracking techniques with the method given in Chapter 1 for finding a single topological sorting. There is however one important difference between the two algorithms. Consider the stage in the process of generating a topological sorting where the partial permutation $p_1p_2\cdots p_{i-1}$ has been constructed, and $S = \{1,2,\ldots,n\} - \{p_1,p_2,\ldots,p_{i-1}\}$ contains those elements not yet selected for this current configuration. In order to find all possible choices for p_i, the algorithm of Figure 2.3 searches through S selecting those elements that have no predecessors in the partial ordering other than $\{p_1,p_2,\ldots,p_{i-1}\}$. The Knuth-Szwarcfiter algorithm, on the other hand, maintains a list M that contains, at each stage, precisely those elements that have no predecessors in the partial ordering other than $\{p_1,p_2,\ldots,p_{i-1}\}$. By maintaining this list M, the Knuth-Szwarcfiter algorithm avoids the need to make a brute-force search through S at every stage of the generation process in order to find all the possible choices for p_i, and as a result it is on the order of n times faster than the algorithm in Figure 2.3.

The Knuth-Szwarcfiter algorithm is shown in Figure 2.4. Procedure TOPSORTS_KS(i) assumes that for all elements y in $S = \{1,2,\ldots,n\} - \{p_1,p_2,\ldots,p_{i-1}\}$ the current value of a global variable COUNT[y] is

```
procedure GENERATE_KS(R,n):
begin S:= {1,2,...,n};
      M:= {y|there is no x in S such that xRy};
      TOPSORTS_KS(1);
end;


procedure TOPSORTS_KS(i):
begin if i = n then P[n] = sole element of M; PROCESS(P);
        else for j ∈ M do
               begin erase all pairs of the form jRk;
                     P[i]:= j;
                     TOPSORTS_KS(i+1);
                     retrieve all pairs of the form jRk;
               end;
      endif;
return;
```

Figure 2.4 The Knuth-Szwarcfiter generating algorithm

the number of ordered pairs xRy for x in S, that is COUNT[y] is the number of predecessors of y other than $\{p_1, p_2, \ldots, p_{i-1}\}$ and the list M contains precisely those elements y in S such that COUNT[y] = 0.

The execution of TOPSORTS_KS(i+1) may cause temporary changes to the contents of M and array COUNT, but both are restored to their entry values upon exit. The array P contains each topological sorting as it is generated, that is P[i] = p_i for i = 1,2,...,n.

When the operation "erase all pairs of the form jRk" is executed, COUNT[k] is decreased by 1 for each element k such that jRk, and those elements for which COUNT[k] drops to zero are added to the list M. Conversely, when the operation "retrieve all pairs of the form jRk" is executed, COUNT[k] is increased by 1 for each element k such that jRk, and those elements for which COUNT[k] is now 1 are removed from M.

As in the case of the algorithm of Figure 2.3, the Knuth-Szwarcfiter algorithm is applicable to any binary relation R defined on the set

{1,2,...,n}. Knuth and Szwarcfiter [1974] give an efficient machine-oriented version of their algorithm, in which the recursion is eliminated.

2.1.2.1 Time complexity

Suppose a binary relation R defined on {1,2,...,n} consists of m pairs of the form iRj. For each topological sorting generated, the procedure TOPSORTS_KS is called at most n times, and in the process at most all m of the pairs iRj are erased and subsequently retrieved. So the algorithm takes at most O(m+n) time per topological sorting generated.

Now let us briefly look at the behaviour of m. The maximum value that m can be is $(n^2-n)/2$; this happens when the given poset is a chain, and in this case $O(n^2)$ time will be required to generate the single permutation that will be a topological sorting.

If the given poset is completely unordered, m is at its minimum value of zero, and the n! topological sortings will each be generated in O(n) time.

In general, for topological sorting problems where t, the number of solutions, is relatively close to n!, m tends to be small; and for problems where t is very much smaller than n!, m tends to be large. So in general, the higher the ratio of t to n!, the more efficiently the solutions are generated.

It should be stressed that this is a very general rule and it is not always true, because a poset may be represented by any spanning subgraph (i.e. binary relation) that preserves reachability (see p.2), and so the value of m may vary for different instances of the same topological sorting problem.

2.2 A non-backtracking algorithm

For a topological sorting problem on the set {1,2,...,n} for which the permutation 12...n is a solution, Varol and Rotem [1977] give an algorithm that will generate all other solutions by application of a series of adjacent transpositions and cyclic right-rotations.

In general, the permutation 12...n is not necessarily a solution to a topological sorting problem on n elements, and therefore a renaming process is used to transform the problem into an equivalent problem having the permutation 12...n among its solutions. If R is a partial ordering defined on a set of n elements, and $p_1p_2\cdots p_n$ is any permutation consistent with R, then renaming the elements with their locations, that is $p_i \Rightarrow i$, and translating the relation R accordingly, that is $p_iRp_j \Rightarrow iRj$, gives the required transformation. We call this topological sorting that is renamed 12...n the <u>starting solution</u> for the given topological sorting problem. The starting solution for a topological sorting problem may be obtained by an application of the method for finding one topological sorting described in Chapter 1.

Given a permutation of $\{1,2,\ldots,n\}$ consistent with the partial ordering R, we say that an element i is <u>blocked</u> (from moving rightwards) if iRj for j the right-hand neighbour of i, or if i is the right-most element in the permutation. If $p_1p_2\cdots p_n$ is a permutation consistent with R, then the transposition of the adjacent elements p_i and p_{i+1} will violate the precedence imposed by R if and only if p_iRp_{i+1} and therefore transposing an element p_i with its right-hand neighbour p_{i+1} will yield a new topological sorting if and only if p_i is not blocked. This principle is used to generate all topological sortings, given the starting solution $p_1p_2\cdots p_n$ = 12...n as follows.

Suppose we have a topological sorting of the n-1 elements $2,3,\ldots,n$. From it we can construct all the topological sortings of $1,2,\ldots,n$ in which the elements $2,3,\ldots,n$ remain fixed relative to one another by placing element 1 on the left-hand side of the topological sorting of $2,3,\ldots,n$ and then moving 1 rightwards by a series of adjacent transpositions, until it is blocked. A cyclic right-rotation is then performed on the subsequence consisting of 1 and all the elements to its left, resulting in 1 being placed back on the left at its starting position. The next topological sorting of $2,3,\ldots,n$ is formed, element 1 is moved rightwards again until it is blocked and a cyclic right-rotation is performed, and so forth.

All topological sortings of $2,3,\ldots,n$ are formed in the same way using each of the topological sortings of $3,4,\ldots,n$ through which to

```
procedure GENERATE_V(R,n):
begin compute one topological sorting p1p2...pn;
      translate R so that each piRpj becomes iRj;
      TOPSORTS_V;
end;


procedure TOPSORTS_V:
begin for i:= 1 to n do P[i]:= LOC[i]:= i;
      PROCESS(P);
      i:= 1;
      while i < n do
        begin if i is not blocked
                 then j:= right-hand neighbour of i;
                      transpose i and j;
                      LOC[i]:= LOC[i]+1; i:= 1;
                      PROCESS(P);
                 else ROTATE(i);
                      LOC[i]:= i; i:= i+1;
              endif;
        end;
return;
```

Figure 2.5 Varol's generating algorithm

move element 2. This process is repeated for the topological sortings of 3,4,...,n, the topological sortings of 4,5,...,n and so on up to the single topological sorting of the element n.

This algorithm is shown in Figure 2.5 The array P initially contains the permutation 12...n, and the array LOC is used to give, at each stage, the location of element k, for all $k \leq i$.

We assume that we have a primitive operation "ROTATE(i)" which performs a cyclic right-rotation of the elements P[i],P[i+1],....., P[LOC[i]] = i. In other words, "ROTATE(i)" equivalent to

```
m:= LOC[i];
while m > i do
  begin P[m]:= P[m-1];
        m:= m-1;
  end;
P[i]:= i;
```

This operation can be performed very ef[illegible] on some computers, although, of course, it will generall[illegible] more efficient than a transposition of adjacent elements, si[illegible] such a transposition is equivalent to a cyclic right-rotation of a sequence of two elements.

Consider finding all permutations of the set S = {a,b,c,d,e} subject to the constraints aRd, bRd, and eRa. One such permutation is becad. By using this permutation as the starting solution, we reformulate the problem as: Starting with the permutation 12345, generate all other permutations consistent with the partial ordering R characterized by the pairs 1R5, 2R4 and 4R5. The sequence of permutati[illegible] generated is shown in Figure 2.6. The starred permutations are redundant solutions produced as a result of the cyclic right-rotations and are not passed through the PROCESS marco.

12345	becad	21435	ebacd
21345	ebcad	24135	eabcd
23145	ecbad	24315	eacbd
23415	ecabd	*12435	
*12345		*12435	
13245	bcead	12453	beadc
31245	cbead	21435	ebadc
32145	cebad	24153	eabdc
32415	ceabd	*12453	
*13245		*12453	
*12345		*12345	
12435	beacd	*12345	

Figure 2.6 Solutions generated by Varol's algorithm

2.2. Time complexity

Firstly, finding a starting solution, and translating the partial ordering accordingly, will clearly take $O(m+n)$ time, for m the number of pairs iRj.

Now suppose we apply the algorithm to a topological sorting problem on the set $\{1,2,\ldots,n\}$ for which there are t solutions, and that in the process of generating these solutions, r cyclic right-rotations are executed. From Figure 2.5 it can be easily seen that the total time required by the algorithm will be $c_1 r + c_2 t$, for some constants c_1 and c_2. Varol and Rotem [1977] show that $r < nt$, by noting that there can be at most n-1 successive rotations after the generation of each topological sorting, and in fact this worst-case situation happens only after the generation of the final topological sorting. Therefore $c_1 r + c_2 t < t(c_1 n + c_2)$ and hence the algorithm takes at most $O(n)$ time per solution generated (except for the starting solution).

We shall now consider a more detailed approach to analysing this algorithm in order to find a more refined upper bound on r, the number of rotations.

Lemma 2.1 $\sum_{i=1}^{n} i! = a_n \cdot n!$ where $1 \leq a_n \leq 3/2$.

Proof: For $n > 1$, let $S_n = \sum_{i=1}^{n} i! = a_n \cdot n!$

Then $S_n = S_{n-1} + n!$

$= a_{n-1} \cdot (n-1)! + n!$

$= (\frac{a_{n-1}}{n} + 1) \cdot n!$

So $a_n = a_{n-1} \cdot \frac{1}{n} + 1$.

By inspection $a_1 = 1$, $a_2 = a_3 = 3/2$ and $a_4 = 11/8 < a_3$.
For $n > 4$, assume that $a_{n-1} < a_{n-2}$. Then

$$a_{n-1} \cdot \frac{1}{n} + 1 < a_{n-1} \cdot \frac{1}{n-1} + 1 < a_{n-2} \cdot \frac{1}{n-1} + 1$$

that is $a_0 < a_{n-1}$, and therefore by induction the result follows.

(Q.E.D.)

During the execution of the algorithm, each time that an element i is blocked from moving rightwards, a cyclic right-rotation is executed by an invocation of ROTATE (i), and we call such a rotation an i-rotation. Since an element i is moved rightwards and subsequently rotated back to position i once for every arrangement of the elements $\{i+1, i+2, \ldots, n\}$ which is consistent with the given partial ordering, the total number r_i of i-rotations is equal to the number of topological sortings of $\{i+1, i+2, \ldots, n\}$. Therefore the total number of rotations executed in the process of generating all topological sortings of $\{1, 2, \ldots, n\}$ is

$$r = \sum_{i=1}^{n-1} r_i$$

$$= \sum_{i=1}^{n-1} (\text{number of topological sortings of } \{i+1, i+2, \ldots, n\})$$

The smallest possible value that r_i, the number of topological sortings of $\{i+1, i+2, \ldots, n\}$, can be is r_{i+1}, which will occur in the case when $-1Rj$ for all $j = i+2, i+3, \ldots, n$, that is if element i+1 must precede, in the partial ordering R, every higher numbered element. The largest possible value that r_i can be is $(n-i)r_{i+1}$ which will occur when $i+1Rj$ for all $j = i+2, i+3, \ldots, n$ that is if element i+1 can be placed on the right of every higher numbered element without contradicting the restrictions of R. Therefore we have the conditions:

1) $r_i/(n-i) \leq r_{i+1} \leq r_i$ for $i = 0, 1, 2, \ldots, n-2$

2) $r_{n-1} = 1$

where $r_0 = t$ is the total number of topological sortings generated by the algorithm. The largest possible of r, let us call it r_{max}, occurs for:

1) $r_i = t$ for $i < n-k$

2) $r_i = (n-i)!$ for $i \geq n-k$

where k is the integer satisfying $k! \leq t < (k+1)!$

<u>if $k \leq n-1$</u>

$$r_{max} = \sum_{i=1}^{n-k-1} t + \sum_{i=n-k}^{n-1} (n-i)!$$

$$= \sum_{i=1}^{n-k-1} t + \sum_{i=1}^{k} i!$$

$$= (n-k-1)t + a_k \cdot k! \quad \text{by Lemma 2.1}$$

So the maximum number of rotations per solution generated is

$$y = (n-k-1) + a_k \cdot k!/t \leq n-k-1 + a_k \leq n-k + 1/2 \qquad (2.1)$$

<u>if $k = n-1, n$</u>

(1) $k = n-1$ when $t = n!/2$, that is when there is exactly one pair iRj in the partial ordering.

(2) $k = n$ when $t = n!$, that is when the partial ordering is empty.

In either case, $$r_{max} = \sum_{i=1}^{n-1} (n-i)! = \sum_{i=1}^{n-1} i!$$

$$= a_{n-1} \cdot (n-1)! \quad \text{by Lemma 2.1}$$

So the maximum number of rotations per solution generated is

$$y = \begin{cases} a_{n-1} \cdot \frac{1}{n} & \text{for } k = n-1 \\ a_{n-1} \cdot \frac{1}{n} & \text{for } k = n \end{cases} \qquad (2.2)$$

From (2.1) and (2.2) we have firstly that the number of rotations per solution is strictly less that n and so, as we have already seen, the algorithm requires O(n) time per solution generated.

Secondly, as t approaches n!, k approaches n, so the maximum number of rotations per solution decreases, and consequently the total time per solution decreases. In fact for the two highest values that t can assume, namely n!/2 and n!, the maximum number of rotations per solution - and therefore the total time per solution - is O(1/n).

So it follows that the algorithm generates solutions most efficiently for problems where t is relatively close to n!, that is when t/n!, the ratio of topological sortings to the total number of permutations, is large; and it generates solutions least efficiently for problems where t/n! is small.

So far we have analysed Varol's algorithm assuming the existence of a primitive operation ROTATE which performs a cyclic right-rotation on a sequence of adjacent elements in constant time. We now show that if no such primitive operation exists, and ROTATE is implemented by a procedure that requires time proportional to the length of the sequence being rotated, then the order of complexity of the algorithm does not change.

Suppose that the ROTATE operation is implemented by the procedure

```
procedure ROTATE(i):
begin m:= LOC[i];
      while m > i do
         begin P[m]:= P[m-1];
               m:= m-1;
         end;
      P[i]:= i;
end;
```

This procedure needs 2d(1 + LOC[i]-i) time, for some constant d, to perform a cyclic right-rotation on the sequence of elements P[i], P[i+1],...,P[LOC[i]].

When ROTATE is about to be called for the j-th time, element i_j is in position LOC[i_j] of array P, having moved there from its leftmost position P[i_j] as the result of (LOC[i_j] - i_j) transpositions. Each topological sorting solution (except the starting solution) is generated from the previous one as the result of a transposition, so the total number of transpositions executed is t-1, and therefore:

$$\sum_{j=1}^{r} (LOC[i_j]-i_j) = t-1$$

So the total time required for the execution of all r rotations is $2d(r+t-1)$. Now the algorithm needs $c_1 r + c_2 t = O(r+t)$ time when a primitive ROTATE operation is assumed, and therefore when the non-primitive ROTATE procedure is used the total running time increases by $(2d-1)r + 2d(t-1)$ to $[(c_1+2d-1)r + (c_2+2d)t - 2d]$ which is still $O(r+t)$.

2.2.2 Rotations and the starting solution

When Varol's algorithm is used to generate all topological sortings of a given poset, the total number of rotations executed (and so the total running time, since it is an $O(r+t)$ algorithm) depends on the starting solution chosen, that is the particular topological sorting which is renamed 12...n.

For example, given the restrictions aRd, bRd and eRa on the set S = {a,b,c,d,e}, if becad is used for the starting solution 9 rotations will be executed (see Figure 2.6). The number of rotations can vary between 7 (if cbead is chosen as the starting solution) and 14 (if ebadc is chosen). Another example is the poset shown in Figure 2.7 which consists of a chain of n-1 elements and one isolated element. There are n+1 topological sortings of this poset. If the topological sorting $yx_1x_2\cdots x_{n-1}$ is used for the starting solution, then the number of rotations is n-1 (which means less than one per solution generated), and the total time required to generate all the solutions is $O(n)$. If, on the other hand, $x_1x_2\cdots x_{n-1}y$ is used for the starting solution, $(n^2-n)/2$ rotations are executed and $O(n^2)$ time is needed to generate all solutions.

The problem of determining which topological sorting to use as the starting solution so that the minimum number of rotations are executed, received a good deal of attention from this writer, but not much success was obtained in finding a complete solution. The following is one of the more promising approaches considered.

Given a poset of n elements, we associate with each element j a number, called the blocking factor of j, which is equal to the number

Figure 2.7

of elements which must follow j in the partial ordering. The L-value of a topological sorting $p_1p_2\cdots p_n$ is defined to be the radix n number $b_1b_2\cdots b_n$, that is

$$b_1b_2\cdots b_n = \sum_{i=0}^{n-1} b_{n-i}\cdot n^i,$$

where b_i is the blocking factor of p_i.

For example, given the partial ordering bRa, bRc, bRd and dRc on {a,b,c,d}, then the topological sorting badc has an L-value of 3100 (base 4).

Now for the starting solution, we choose that topological sorting $p_1p_2\cdots p_n$ with the lowest L-value. To construct this topological sorting, at each stage, when the partial permutation $p_1p_2\cdots p_{i-1}$ has been constructed, we choose for p_i that element of M_i (the set of valid choices) which has the lowest blocking factor. The motivation for choosing the starting solution in this way is as follows.

Firstly, a rotation is executed by Varol's algorithm every time that an element is blocked from moving rightwards, and so we wish to find a starting solution that results in the smallest number of blockings. Secondly, note that in generating all t topological sortings of a given poset:

(1) The number of transpositions executed is t-1, no matter which topological sorting is used as the starting solution.

(2) In between every transposition of element i+1 with its right-hand neighbour, element i is repeatedly transposed with its right-hand neighbour until it is blocked.

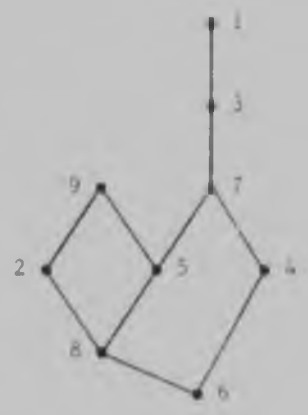
So by choosing as the starting solution that topological sorting in which the elements least likely to be blocked are as far to the left as possible, we are attempting to keep the number of transpositions (and therefore solutions generated) between successive rotations as high as possible, and consequently the number of blockings (and therefore rotations) as low as possible.

The writer found for most posets considered that this method correctly selected the starting solution which resulted in fewest rotations. Also in many of the cases, the number of rotations produced by the topological sortings (when used as the starting solution) was, more or less, an increasing function of their L-values.

As an example, in Figure 2.8(a) is the Hasse diagram of the partial ordering given by Knuth [1968, p.259] to explain the principle of topological sorting. Figure 2.8(b) shows the actual ordered pairs that constitute the partial ordering R, and Figure 2.9 shows all topological sortings (listed in increasing L-value), their L-values, and the number of rotations they produce when used as the starting solution. (Note how the number of rotations tends to increase with increasing L-value).

This method for choosing the starting solution has been discussed primarily to provide some guideline for possible future research. The results obtained by the writer are not conclusive, since the testing was by no means comprehensive and the posets considered were chosen rather arbitrarily. (In the following section we discuss the problems involved in defining a good sample of posets to use for empirical tests).

Note also that to generate all topological sortings of a poset using Varol's algorithm, the partial ordering can be represented by any binary relation (i.e. the set of restrictions) whose transitive closure is equal to the partial ordering. Now to find the lowest L-valued topological sorting, the partial ordering itself is needed in order to compute the blocking factors, and this means the execution



(a)

1R3	4R6
1R4	
1R5	5R6
1R6	5R8
1R7	
1R8	7R4
	7R5
2R6	7R6
2R8	7R8
	8R6
3R4	
3R5	9R2
3R6	9R5
3R7	9R6
3R8	9R8

(b)

Figure 2.5 A partial ordering

of an $O(n^3)$ transitive closure procedure. So finding the lowest L-valued topological sorting and using it for the starting solution may not always result in a reduction of running time, since for a topological sorting problem having, for example, $O(n)$ solutions Varol's algorithm requires only $O(n^2)$ time no matter which solution is used as the starting solution.

2.3 Conclusion

Three topological sorting algorithms have been discussed. The two backtracking algorithms are essentially the same - we have shown that Wells' algorithm is just a less efficient version of the Knuth-Szwarcfiter algorithm.

The Knuth-Szwarcfiter algorithm requires $O(m+n)$ time for each solution generated, and in general the solutions are generated most efficiently when t, the number of solutions is close to n!.

TOPOLOGICAL SORTING	L-VALUE (BASE 9)	NUMBER OF ROTATIONS PRODUCED
921374586	426541210	32
921375486	426542110	33
921375846	426542110	33
912374586	462541210	43
912375486	462542110	44
912375846	462542110	44
913274586	465241210	51
913275486	465242110	52
913275846	465242110	52
913742586	465412210	55
913745286	465412210	55
913724586	465421210	56
913754286	465421210	56
913725486	465422110	57
913752486	465422110	57
913725846	465422110	57
913752846	465422110	57
192374586	642541210	61
192375486	642542110	62
192375846	642542110	62
193274586	645241210	69
193275486	645242110	70
193275846	645242110	70
193742586	645412210	73
193745286	645412210	73
193724586	645421210	74
193754286	645421210	74
193725486	645422110	75
193752486	645422110	75
193725846	645422110	75
193752846	645422110	75
137492586	654142210	73
137495286	654142210	73
139274586	654241210	76
139275486	654242110	77
139275846	654242110	77
137942586	654412210	79
137945286	654412210	79
139742586	654412210	80
139745286	654412210	80
137924586	654421210	80
137954286	654421210	80
139724586	654421210	81
137925486	654422110	81
139754286	654421210	81
137952486	654422110	81
137925846	654422110	81
137952846	654422110	81
139725486	654422110	82
139752486	654422110	82
139725846	654422110	82
139752846	654422110	82

Figure 2.9 Comparing starting solutions

Varol's algorithm requires O(m+n) time to generate the starting solution, and O(n) time for every other solution. The solutions are always generated most efficiently for problems where t is close to n!, and for t = n!/2 or n! the time per solution is O(1/n).

To maintain some perspective we must note that our analysis does not give much idea about the values of the constants of proportionality for these order-of-magnitude measures of time complexity. Suppose we are concerned with the practical details of solving topological sorting problems for which a reasonably large proportion of all n! permutations are solutions. Since the function n! grows extremely quickly, real-world time considerations will restrict us to dealing with those problems for which n is small, and in such cases the constants of proportionality will indeed be significant. Nevertheless, most practical problems involving topological sorting, like those we discussed in Chapter 1, are concerned with finding only one, and not all possible topological sortings.

A possible way of comparing the algorithms is by obtaining empirical results, that is by implementing the algorithms on a real computer and comparing their running times. We face a number of problems using this approach. Firstly, whatever results we may obtain will be of limited significance, since they will be dependent, to some extent, on the specific implementation. For example, while we have shown that the order of complexity of Varol's algorithm is not affected by whether or not the ROTATE operation is primitive, when the algorithm is implemented, the actual running time will certainly depend on whether or not the computer used has hardware rotation capabilities. (In his survey of permutation generation algorithms, Sedgewick [1977] points out the problem that the comparing of algorithms by empirical testing can really become a comparison of compilers, programmers and computers, not algorithms).

A second problem encountered with empirical tests is how to select a good sample of partial orderings for input to the programs. The problem of defining what constitutes an average partial ordering and a good distribution of partial orderings is not simple. Even assuming (optimistically) that we manage to overcome this difficulty, we immediately face another problem. Remember that for the Knuth-Szwarcfiter and Varol's algorithms, a partial ordering may be represented

by each of the binary relations whose transitive closure is equal to the partial ordering. For example a chain of length n may be represented in n! different ways, and depending which representation we choose to use, m, the number of pairs xRy can vary between n and $(n^2+n)/2$. Since O(m+n) time is required for each solution generated by the Knuth-Szwarcfiter algorithm, and for the starting solution generated by Varol's algorithm, we have the situation where a program may have many, and possibly vastly differing, running times for the same topological sorting problem, depending on the representation of the partial ordering.

Yet another problem with empirical tests, one that we have mentioned previously in this section, is that of topological sorting problems where the the number of solutions is .atively close to n!. It is reasonable to assume that a good sample of partial orderings will contain problems like this. This means the empirical testing will be limited to problems for which n is very small, and so whatever conclusions may be reached will not be very general. (For n > 25, the time required to generate n! permutations, at the rate of one per microsecond, is greater than the age of the earth [Sedgewick 1977]).

CHAPTER 3 - ENUMERATION OF TOPOLOGICAL SORTINGS

3.0 Introduction

There is no simple formula to calculate the number of permutations consistent with an arbitrary poset, and the enumeration must be done in other ways. There are some classes of posets however, for which formulae can be derived.

3.1 Some special cases

a) Trees: If T is a directed graph which is a tree, then N(T) the number of permutations consistent with T can be calculated by the following expression which appears in Szwarcfiter and Wilson [1978]:

$$N(T) = n! / \prod_{x \in T} (|T(x)|)$$

where n is the number of vertices, the product is over all vertices x in T, T(x) is the subtree rooted at x, and |T(x)| is its number of vertices. Basically, the reason why this formula solves the problem is that if x is a vertex in the tree T, and y is another vertex belonging to a subtree of x, then x must necessarily precede y in any of the permutations. Hence, the total number of permutations in which x precedes every vertex belonging to any of its subtrees, is $n!/|T(x)|$. By considering all vertices of the tree, we obtain the above formula for N(T). There are, for example

$$8!/8.6.4.1.1.1.1.1 = 210$$

permutations of $\langle 1, 2, \ldots, 8 \rangle$ consistent with the poset of Figure 3.1.

This result for trees can easily be extended to forests. If G is a forest of trees $T_1, T_2, \ldots, T_k$, we construct the tree T with root x so that $T_1, T_2, \ldots, T_k$ are the subtrees of x, and apply the formula to T. In fact, the number of topological sortings of the forest G will be equal to the number of topological sortings of the tree T rooted at x, constructed from G, since corresponding to every topological sorting $p_1 p_2 \ldots p_n$ of G is the topological sorting $x p_1 p_2 \ldots p_n$ of T.

Figure 3.1 A tree

b) Zig-zag posets: A partial ordering on the set $\{1,2,\ldots,n\}$ which can be represented by n-1 pairs, each of the form iRi+1 or i+1Ri for $i = 1,2,\ldots,n-1$, is called a zig-zag poset, since its Hasse diagram has a characteristic zig-zag shape. The enumeration of the topological sortings of zig-zag posets is discussed in Section 4.1.3.

c) Graphs corresponding to Young tableaux: A graph that corresponds to a Young tableau is an acyclic directed graph G, where the vertices can be arranged into an array of left-justified rows of decreasing length in such away that:

(1) for each vertex v, if b is the vertex below v then (v,b) is an edge in G, and if r is the vertex to the right of v then (v,r) is an edge in G; and

(2) no other edges exist.

In this kind of graph, the hook of a vertex is defined to be the vertex plus the vertices that lie below and to its right. The hook length of a vertex is the number of vertices in the hook of this vertex. For example, the directed graph in Figure 3.2 corresponds to a Young tableau, and the vertex at the top left-hand corner has a hook length of 9. The following theorem is from Knuth [1973]:

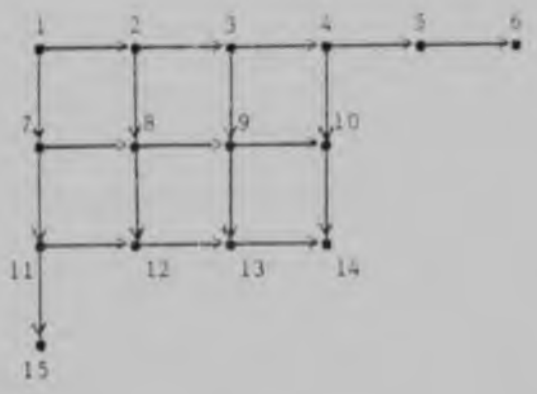


Figure 3.2 A graph corresponding to a Young tableau

Theorem 3.1 If G is a graph with n vertices corresponding to a Young tableau, then the number of ways to topologically sort the partial ordering represented by G, is n! divided by the product of the hook lengths.

From this theorem we can calculate the number of permutations consistent with the poset in Figure 3.2 as

$$15!/9.7.6.5.4.1.6.4.3.2.5.3.2.1.1 = 80080.$$

Referring to this theorem, Knuth states: "It would be nice if there were some simple formula which would generalize [the theorem] to the case of an arbitrary directed graph; but not all graphs have such pleasant properties as the graphs corresponding to tableaux." [Knuth 1973, p.65]. He goes on to say that while there are certain other classes of directed graphs, such as trees, which have a simple solution to this enumeration problem, it can also be shown that some directed graphs have no simple formula. For example, the number of solutions to a topological sorting problem on n elements is not always a divisor of n! For instance, the 4-element poset:

has the five topological sortings: 1234, 1243, 2134, 2143 and 2413.

A lack of "pleasant" properties in arbitrary acyclic directed graphs compels us to consider an alternative, and more complicated enumeration method.

3.2 Wells' enumeration algorithm

Wells [1971] presents an algorithm to enumerate topological sortings which is based on three easily established facts. Let (S,R) be a poset of n elements. Then for $T \subset S$, denote by $N(T)$ the number of permutations of elements in T which are consistent with the restrictions imposed by the poset. The three facts are:

(a) If V and W are disjoint subsets of S and xRy for all $x \in V$, $y \in W$, then $N(V \cup W) = N(V).N(W)$.

(b) If K1 and K2 are disjoint subsets of S, and $x \not R y$, $y \not R x$ for $x \in K1$, $y \in K2$, then $N(K1 \cup K2) = N(K1).N(K2).\binom{|K1|+|K2|}{|K1|}$.

(c) If T is a subset of S and $t \in T$, then $N(T) = \sum (N(V).N(W))$, where the summation is over all partitions of $T - \{t\}$ into two sets V and W such that $V \cap W = \{\}$, $\{x \in T | xRt\} \subset V$ and $\{x \in T | tRx\} \subset W$.

Statements (a) and (b) follow from the product rule for independent events. The binomial coefficient in (b) gives the number of ways that a permutation of the elements in K1 and a permutation of the elements in K2 can be combined to yield a legitimate permutation of the elements in $K1 \cup K2$. Assertion (c) is a straightforward application of the sum rule for mutually exclusive events and of statement (a).

The idea of the algorithm is to split the problem into simpler and simpler parts, recursively. First of all, the poset, treated as an undirected graph, is tested for connectivity. Each component of the problem independently yields an enumeration; these results are eventually combined using (b). To resolve a component, an element t, called the cleavage vertex, is chosen on which an analysis using statement (c) is fashioned. This analysis produces a set of pairs of

simpler problems and each such problem is solved in a recursive manner, the results being combined by the formula of (c). (We assume that for T = empty set, N(T) = 1).

An example of the process is given in Figure 3.3 for the set S = {1,2,...,10}, and the partial ordering represented by the ordered pairs:

1R3	6R5
2R3	6R9
3R4	7R6
3R6	8R7
4R5	9R10

This poset consists of [illegible]e component (represented by the Hasse diagram shown in the [illegible]. the top of Figure 3.3), so the first step is to resolve the [illegible]t into a set of simpler problems using (c). With t = [illegible]ircled vertex), there are two pairs of problems to consider, i.e. [illegible]ere are two partitions which satisfy the conditions stated in (c). Each pair is shown in two solid boxes at the end of the first level links; the two members of a pair are labelled V and W. Each of these members is them analysed for connectedness - the components are shown separated by a dotted line and are labelled K1, K2, etc. Each component is then further resolved, the branching process being terminated at a primitive problem, i.e. a chain or an empty set, either of which yields exactly [illegible] consistent permutation.

The numbers shown underneath each box in the diagram give the number of permutations consistent with the poset illustrated within the box. They are computed from the formula of (b) when combining components and from the formula of (c) when combining problem pairs. For instance, the calculations $2.1.\binom{4+0}{4}$ and 20.6 + 30.3 yield respectively the numbers 30 and 210 of the diagram.

The enumeration algorithm is shown in Figures 3.4 and 3.5. The basic recursive procedure COUNT of Figure 3.4 returns the number of topological sortings of the poset (S,R). (Note that the algorithm will not work correctly if the binary relation R is not transitively closed). The principal variables are the sets V and W which describe the resolved problem pairs, and the set K which describes the compo-

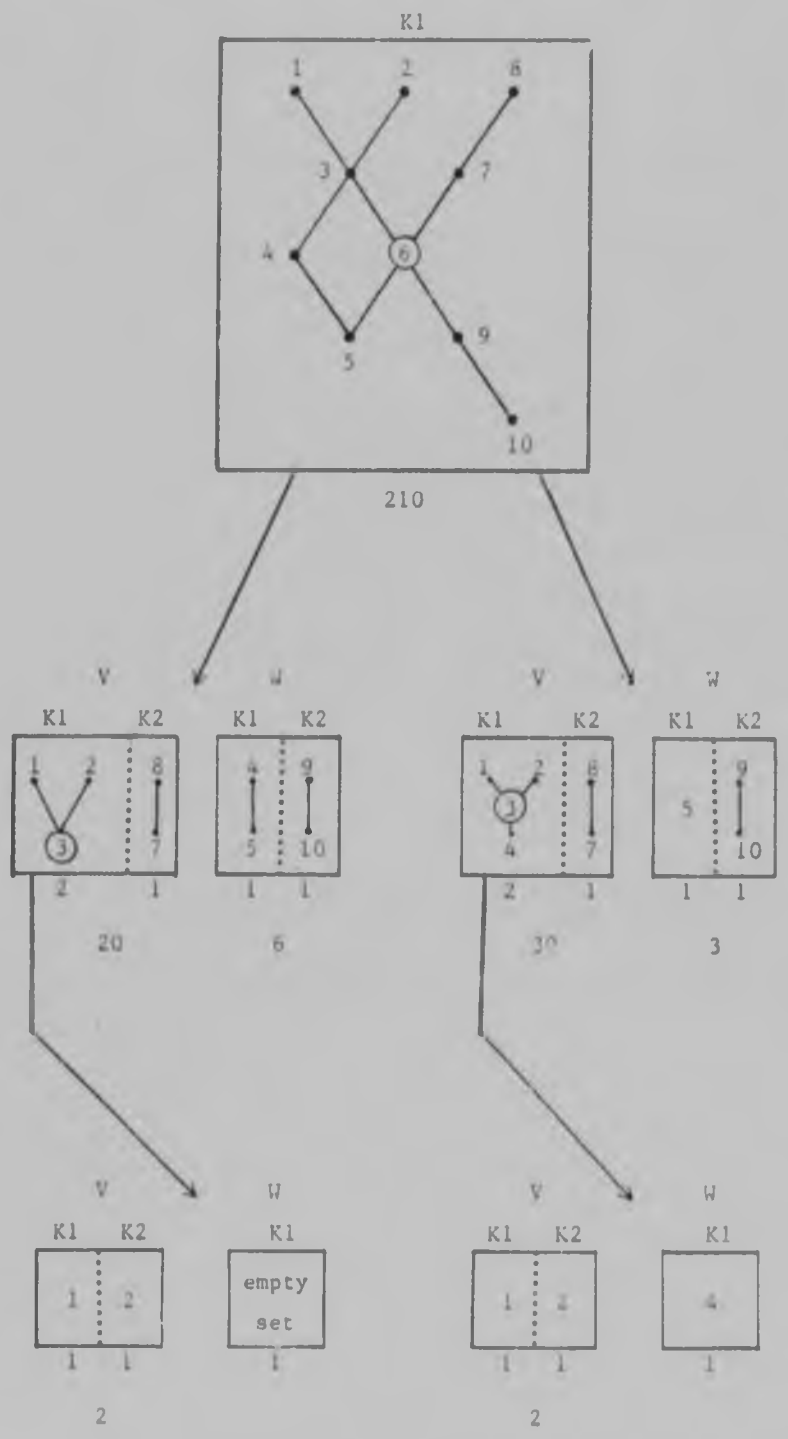


Figure 3.3 An example of the recursive procedure for enumerating topological sortings

```
procedure COUNT(S,R):
begin amnt:= 1; J:= K:= { };
      while J ≠ S do
          begin NEXTCOMP(S, J, K);
                if K is a chain then c:= 1;
                   else repeat NEXTPAIR(K, V, W);
                               n_V:= COUNT(V, R);
                               n_W:= COUNT(W, R);
                               c:= c + n_V.n_W;
                        until  all the pairs (V,W) of K have
                               been generated by NEXTPAIR;
                endif;
                amnt:= amnt.c.( (|J|+|K|) / |K| );
          end;
return amnt;
```

Figure 3.4 Main procedure for the topological sorting enumeration algorithm

nent currently being resolved. The initial problem may consist of several components - possible values for K (in the example of Figure 3.3 there was one component only). Each of these components splits into several pairs of smaller problems given by values of V and W, each of these problems has components and so forth. The recursion terminates when a chain is discovered (an empty component is considered to be a trivial chain).

The variables c, n_V and n_W register various accumulated tallies at each level of recursion. The quantity c which ultimately gives the number of permutations for the active component K, is the sum of products of fact (c) on page 33. The quantities n_V and n_W ultimately give the number of permutations for the pair of problems indicated by V and W respectively. The quantity amnt is computed by applying the formula from fact (b) while contributions accumulate from distinct components. The set J (formed within the NEXTCOMP procedure) cumulates the vertices of the components already considered in the subproblem currently being resolved.

The NEXTCOMP procedure shown in Figure 3.5 generates a new component K of the poset S each time it is called. At each stage the set S ~ J contains those vertices that do not belong to the components already generated. A vertex j is chosen from this set and the component K containing j is then generated. The set K′ consists of recently discovered vertices of the component, and the set K* consists of all vertices connected to any vertex of K′. This process continues until K′ is empty, that is until no new vertices are found.

The procedure NEXTPAIR of Figure 3.5 produces the subproblem pairs (V,W) resulting from component K by generating a new pair every time it is called. The vertex on which the cleavage [fact(c)] is based is chosen as that vertex for which the number of unrelated vertices (i.e. the vertices which neither precede nor succeed the vertex in

```
procedure NEXTCOMP(S, J, K):
begin J:= J ∪ K;
      choose an element j ∈ S ~ J;
      K:= K′:= {j};
      while K′ ≠ { } do
          begin K*:= K′ ∪ {p ∈ S|pRk or kRp for some k ∈ K′};
                K′:= K* ~ K;
                K := K ∪ K′;
          end;
return;
```

```
procedure NEXTPAIR(K, V, W):
begin if this is the first call for this component K
         then CLEAVAGE(K, H, t) endif;
      G:= NEXTSUB(H);
      V:= {p ∈ K|pRt} ∪ G;
      W:= (K ~ {t}) ~ V;
return;
```

Figure 3.5 Auxiliary procedures for the enumeration algorithm

the partial ordering) is a minimum. (This vertex t, and the set H of vertices unrelated to it are computed by the procedure CLEAVAGE, which is invoked during the first call to NEXTPAIR for component K). Those vertices of K which precede t are put into V, and those which succeed t into W. Each legitimate way in which the vertices of H may then be divided between V and W yields a problem pair. This is achieved using the procedure NEXTSUB which generates all the subsets of H closed under the partial ordering. (A subset G of H is closed under the partial ordering R if, for all x and y, $y \in G$ and xRy, implies $x \in G$). The closed subset G is included with V, and W becomes the relative (to $K \sim \{t\}$) complement of V.

Like the procedures NEXTCOMP and NEXTPAIR, NEXTSUB produces a new solution each time it is called. To generate all the closed subsets of H, Wells' algorithm computes every possible subset of H and then tests if it satisfies the required conditions. If the set H has m elements this brute-force method requires the generation and testing of 2^m subsets for each component K. A more efficient method to generate closed subsets is given by Waite [1967].

3.2.1 The cleavage vertex

A curious anomaly in the highly detailed presentation by Wells of his enumeration algorithm is the total absence of an explanation for the method of selecting as the cleavage vertex, that vertex t for which the size of the corresponding set H of unrelated vertices is a minimum. (The algorithm will, of course, work correctly for any choice of cleavage vertex).

The most probable reason that this method of selecting the cleavage vertex t is used, is that it is an attempt to make the algorithm efficient by limiting the number of subproblems generated at each level of recursion.

When the algorithm splits a poset (S,R) of size n into subproblem pairs, each pair (V,W) will have a combined size of n-1 (the cleavage vertex having been eliminated). So the total size of all the subproblems is proportional to the number of subproblem pairs.

Now the number of subproblem pairs (V,W) is equal to the number of

subsets of H closed under the partial ordering R, where H is the set of vertices unrelated to the cleavage vertex t.

In some cases, for example the poset in Figure 3.5(a), the number of closed subsets of H is an increasing function of the size of H, but in general this may not be true. Nevertheless, as we shall see, Wells' method of choosing t so that the corresponding set H has minimum size, is a reasonable way of limiting the number of closed subsets (and hence the number of subproblem pairs) generated at each level of recursion.

Firstly, the number of closed subsets of H must be limited, to some degree, by the size of H, since if $|H| = m$ then the number of closed subsets of H must lie between m+1 and 2^m, these bounds corresponding respectively to no ordering and total ordering of the elements of H. So, on average, it is likely that a large set has more closed subsets that a small set, and if $j < \log_2 k$, then any set of size j will always have fewer closed subsets than a set of size k.

Secondly, it should be noted that H is not an arbitrary set, but in fact a partially ordered subset of the original poset (S,R) of the enumeration problem. The significance of this can be seen from the following lemma.

Lemma 3.1 Let (S,R) be a poset, and H, $H' \subset S$ where $H' = H \cup \{z\}$ for some $z \in S$. If p subsets of H are closed under R, then the number p' of subsets of H' that are closed under R satisfies $p+1 \leq p' \leq 2p$.

Proof: The subsets of H' closed under R which do not contain the element z will be precisely the p closed subsets of H. Also the set H' is a closed subset of itself, and so $p+1 \leq p'$. Furthermore, for each subset C of H' which contains z, $C - \{z\}$ will be a closed subset of H. Since there are p such subsets, there can be at most p closed subsets of H' containing z and so $p' \leq 2p$. (Q.E.D.)

It follows from this lemma that if vertices t and t' are unrelated to the sets of vertices H and H' respectively, where $H \subset H'$, then the policy of choosing t over t' as the cleavage vertex will always result in fewer subproblem pairs.

Therefore it can be seen that Wells' method of cleavage vertex selection is a useful one - it restricts, to some degree, the number of subproblems (and therefore the sum of the sizes of the subproblems) at each level of recursion, and also, perhaps the most decisive factor in favour of this method, it is simple and relatively cheap to use.

(a)

cleavage vertex t	set H of unrelated vertices	number of closed subsets/ subproblem pairs	total subproblem size
1	-	1	6
2	{7}	2	12
3	{6, 7}	4	24
4	{5, 6, 7}	8	48
5	{4, 6, 7}	8	48
6	{3, 4, 5, 7}	10	60
7	{2, 3, 4, 5, 6}	11	66

(b)

Figure 3.5 (a) A poset; (b) the relationship between cleavage vertex and the number of subproblems pairs

Figure 3.6

The number of closed subsets of a set H does not, however, depend solely on the size of H, but on the degree of ordering among its elements as well. Consider the poset in Figure 3.6.

Here, u_i is unrelated to 2n vertices for $1 \leq i \leq n$,
w_i is unrelated to n+1 vertices for $1 \leq i \leq n$,
y is unrelated to n vertices, and
z is unrelated to n vertices.

The sets $H_y = \{u_1, u_2, \ldots, u_n\}$ and $H_z = \{w_1, w_2, \ldots, w_n\}$ of vertices unrelated to y and z respectively are the same size, but while the u_i's are all unrelated to one another, the w_i's are totally ordered (i.e. they form a chain). This difference results in vastly differing subproblem pairs depending on which of y or z is chosen as the cleavage vertex. (The algorithm resolves ties like this by selecting the vertex first encountered in the search for the cleavage vertex).

y as the cleavage vertex: Each of the 2^n subsets of the set $H_y = \{u_1, u_2, \ldots, u_n\}$ will be closed under the partial ordering, in other words, there will be 2^n pairs of subproblems. For each subproblem pair (V,W), the set W will contain vertex z as well as the component $\{w_1, w_2, \ldots, w_n\}$, since these vertices must always follow y in the partial ordering. Within the 2^n pairs will be $2^n-(n+1)$ non-primitive components which will have to be split up further - these non-primitive components will each be of the form:

for some subset $\{u_i,\ldots,u_j\}$ of H_y.

z as the cleavage vertex: The set $H_z = \{w_1,w_2,\ldots,w_n\}$ forms a chain and so there are n+1 closed subsets, namely $\{\ \}$, $\{w_1\}$, $\{w_1,w_2\}$,..., $\{w_1,w_2,\ldots,w_n\}$. Also, each subproblem will consist of primitive components (i.e. chains) only, and so no further recursive applications of the algorithm are necessary.

The difference between choosing y and z is quite significant - 2^n subproblem pairs containing a total of $2^n-(n+1)$ non-primitive components, compared with n+1 pairs having primitive components only.

example illustrates quite clearly that the number of closed
ts of a set H depends not only on the size of H, but the degree
ing among its elements. In fact, if we modify the previous
so that the number u_i's is $m \neq n$ (i.e. y and z are no longer unrelated to the same number of vertices), then $H_y = \{u_1,u_2,\ldots,u_m\}$ will have more closed subsets than will $H_z = \{w_1,w_2,\ldots,w_n\}$ for m even as small as $\log_2 n + 1$.

So far we have discussed the problems of finding the cleavage vertex that results in the fewest number of subproblem pairs. However, splitting a problem into a minimum number of subproblem pairs (and therefore obtaining a minimum for the total size of the subproblems combined) does not necessarily lead to the most efficient algorithm. Consider the poset in Figure 3.7.

Figure 3.7

Vertex 7 is unrelated to just one vertex, 6, and vertex 5 is unrelated to the three vertices 1, 4 and 6, and so it follows from Lemma 3.1 that more subproblem pairs are generated when the problem is split around vertex 5 than when it is split around vertex 7. If 5 is the cleavage vertex, there are six subproblem pairs, and if 7 is the cleavage vertex, there are two subproblem pairs, as shown in Figures 3.8 and 3.9 respectively.

The six pairs obtained by splitting around vertex 5 are composed of primitive components only, with the exception of the V-shaped component:

which appears twice, and so a further level of recursion is required to resolve the problem completely.

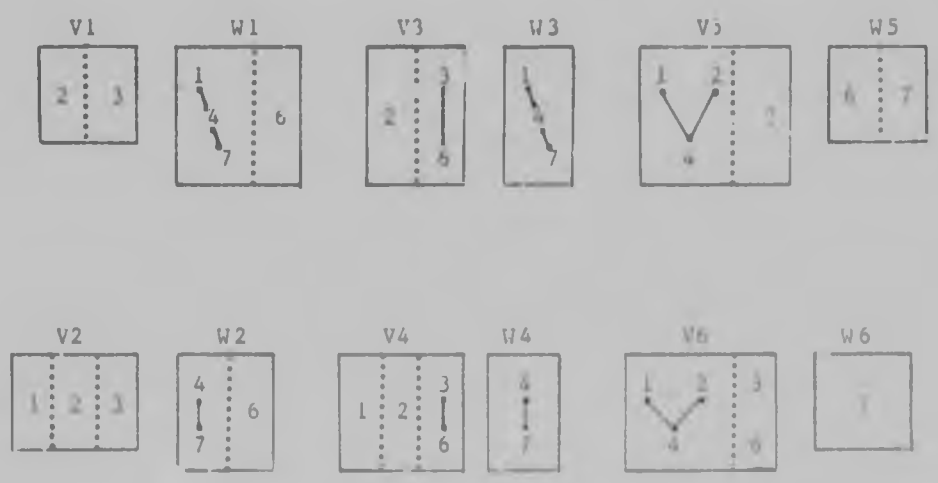


Figure 3.8 The subproblem pairs resulting when vertex 5 is the cleavage vertex

The two subproblem pairs obtained by splitting around vertex 7 contain two non-primitive components, and must be resolved further. Another level of recursion (using 2 as the cleavage vertex) results in ten more subproblems composed of primitive components only, with the exception of the V-shaped component:

which appears twice, necessitating a third level of recursion to resolve the problem completely.

Comparing the two different choices for the cleavage vertex, it is apparent that while splitting around vertex 7 results in fewer subproblem pairs, it is not necessarily the better choice; an extra level of recursion is required, and the original problem is broken down into a total of 39 primitive components, compared with a total of 25 when vertex 5 is chosen as the cleavage vertex.

The six subproblem pairs obtained by splitting around vertex 5 are more balanced than the two pairs obtained by splitting around vertex 7, in the sense that the elements are more evenly divided between the two subproblems in each subproblem pair (V,W). For 5 as the cleavage vertex, there are three pairs of subproblems of sizes 2 and 4, two pairs of sizes 3 and 3, and one pair of sizes 5 and 1; so the average sizes of the pair (V.W) are 2.83 and 3.17. For 7 as the

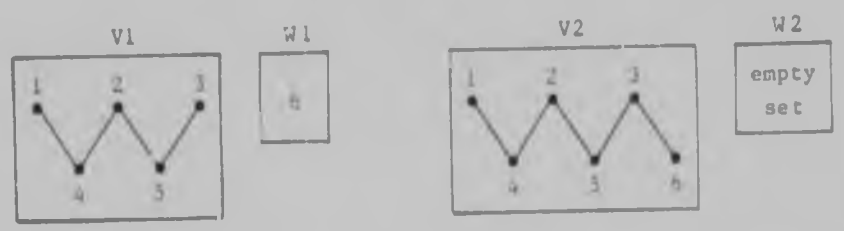


Figure 3.9 The subproblem pairs resulting when vertex 7 is the cleavage vertex

cleavage vertex, there is one pair of subproblems of sizes 5 and 1, and one pair of sizes 6 and 0; so the average sizes of the pair (V,W) are 5.5 and 0.5.

The reason for this difference in balance can be seen by looking at the sets of vertices related to 5 and 7; vertex 7 is related to five other vertices, all of which preceed 7 in the partial ordering, while vertex 5, although related to three other vertices only, precedes one of them, and follows the other two in the partial ordering - a more balanced situation.

In general, if a poset contains among its elements three distinct elements x, y and z where xRy, yRz and xRz, and if y is chosen as the cleavage vertex, then the pair xRz will not feature in any of the subproblems. (Since y is the cleavage vertex, it does not appear in any subproblem and so, obviously, neither will the pairs xRy and yRz or any other pair containing y). This pair xRz does not appear in any subproblem, since the condition that x must precede z is now implicit in the fact that in each pair (V,W) of subproblems $x \in V$, and $z \in W$. So the more equally that the vertices related to the cleavage vertex are divided between preceding and following this vertex in the partial ordering, the more pairs of the form xRz will disappear from the subproblems, and therefore the simpler these subproblems will be.

Note that in the previous example, a pair of subproblems of sizes 5 and 1 arises for either choice of cleavage vertex. Because of the balancing, the subproblem of size 5 obtained with 5 as the cleavage vertex is, however, a much simpler problem than the size 5 subproblem obtained with 7 as the cleavage vertex. The former subproblem consists of two components, one primitive and the other a simple V-shape, while the latter has one component, which on further resolution results in four subproblem pairs composed of a total of 13 primitive components.

In conclusion, we have discussed how different choices of cleavage vertex can result in vastly differing sets of subproblem pairs, and how the efficiency of the enumeration algorithm is affected by properties such as the size of the set H of vertices unrelated to the cleavage vertex t, the partial ordering on this set, and the measure

of balance between the vertices preceding and following the cleavage vertex in the partial ordering. The question that now comes to mind is "what criteria should be used to select the cleavage vertex to achieve maximum efficiency?" In attempting to answer this question we face a number of problems.

Firstly, the fact that an arbitrary ordering lacks what Knuth refers to as "pleasant properties" makes a rigorous analytical evaluation of the problem of cleavage vertex choice extremely difficult, if not impossible. This difficulty is the most probable reason why Wells offers no explanation for his selection method, since it is highly unlikely that this selection method would be given as part of the enumeration algorithm if it offered no advantage over some simpler method. For example the cleavage vertex could be chosen (i) randomly, (ii) as the highest numbered vertex, or (iii) in some other way.

Secondly, Wells' selection method is relatively cheap to use, and another method that results in a better choice of cleavage vertex will not necessarily result in a more efficient algorithm, since the gains in using this better choice of cleavage vertex may very well be outweighed by the work involved in searching for it.

3.2.2 Repeated enumeration of a component

When a problem is split into subproblem pairs, every component of every subproblem is enumerated separately, these results eventually being combined to give the required solution to the original problem. A particular component which appears in a number of different subproblems will be enumerated repeatedly, once for each of these subproblems.

That a component appears repeatedly in different subproblems is an inherent feature of the algorithm, since in every subproblem pair (V,W), the vertices that precede the cleavage vertex will always be in the set V, and the vertices that follow the cleavage vertex will always be in the set W. For example, consider the poset of Figure 3.10(a). If vertex 6, which is unrelated to the fewest other vertices, is chosen as the cleavage vertex, the component of Figure 3.10(b) will appear in each of the eight subproblem pairs.

(a) (b)

Figure 3.10

Repetition of components can be seen as well in the other examples we have discussed. For example, in the poset of Figure 3.6, if y is the cleavage vertex, then the chain $w_1 R w_2$, $w_2 R w_3, \ldots\ldots, w_{n-1} R w_n$ appears in the set W for each of the 2^n subproblem pairs (V,W). Even though this chain is a primitive component that is not resolved further, at least O(n) time is needed to test that it is primitive, which means that at least $O(n.2^n)$ time will be spent on unnecessary repetitions of this test.

A simple way to eliminate the repeated enumeration of components, and so to improve algorithm efficiency, is to allocate O(m) storage cells to "remember" the components that have already been evaluated (where m is the number of closed subsets of the original poset (S,R) of the enumeration problem). This modification can result in a very significant improvement in the speed of the algorithm.

CHAPTER 4 - TWO PROBLEMS REDUCIBLE TO TOPOLOGICAL SORTING

4.0 Introduction

In Chapter 1 we saw that topological sorting can be viewed as a restricted permutation problem, that is a problem concerning the study of permutations of n objects that are consistent with a given set of restrictions. In this chapter we look at two other restricted permutation problems, and we develop methods to solve them by reducing them to topological sorting.

4.1 Permutations with prescribed up-down sequences

A permutation $P = p_1p_2\cdots p_n$ of $\{1, 2, \ldots, n\}$ is said to have a rising sequence (up sequence) if $p_i < p_{i+1} < \cdots < p_{i+k}$, and a falling sequence (down sequence) if $p_i > p_{i+1} > \cdots > p_{i+k}$. Denoting rises and falls by 1 and 0 respectively, the ups and downs are described by a succession of n-1 1's and 0's, called the U-D sequence of the permutation $p_1p_2\cdots p_n$, and the permutation is called a solution to the U-D sequence. For example, the U-D sequence of 72583614 is 0, 1, 1, 0, 1, 0, 1 and 81273645 is another solution to this sequence.

A U-D sequence $s_1, s_2, \ldots, s_{n-1}$ defines a set of permutations of the form $p_1p_2\cdots p_n$ for which the constraints that $p_i < p_{i+1}$ if $s_i = 1$ and $p_{i+1} < p_i$ if $s_i = 0$ are satisfied. So the U-D sequence represents a partial ordering " < " (i.e. the usual relation of "less than" on the real numbers) of the positions $p_1, p_2, \ldots, p_n$ according to their contents. (The topological sorting problem on the other hand, involves a partial ordering of the elements of a set). The Hasse diagrams of these partial orderings have a characteristic zigzag shape. For example, the sequence 0, 1, 1, 0, 1, 0, 1 defines the partial ordering shown in Figure 4.1.

The "ups" of the U-D sequence become the "downs" in the Hasse diagrams and vice versa. This may seem an unfortunate convention, going against the usual notion of up and down, but it is perfectly consistent with the concepts of up and down in mathematical, as opposed to botanical, trees.

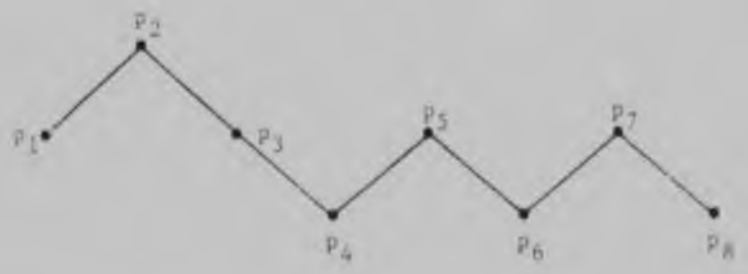


Figure 4.1 A "zig-zag" Hasse diagram

Given an U-D sequence $s_1, s_2, \ldots, s_{n-1}$, the solutions of the form $p_1 p_2 \ldots p_n$ can be found by transforming the problem into an equivalent topological sorting problem. For notational convenience, let us denote temporarily p_i by $p(i)$. Finding a permutation $p(1)p(2)\ldots p(n)$ consistent with the partial ordering on the $p(i)$'s defined by the U-D sequence, amounts to finding a permutation $a(1)a(2)\ldots a(n)$ of the set $\{1, 2, \ldots, n\}$ satisfying

$$p(a(1)) < p(a(2)) < \ldots < p(a(n))$$

that is $p(a(i)) = i$ for $i = 1, 2, \ldots, n$, or $p(i) = a^{-1}(i)$ where $a^{-1}(j) = k$ if $a(k) = j$. Therefore the permutation $a^{-1}(1)a^{-1}(2)\ldots a^{-1}(n) = p(1)p(2)\ldots p(n)$ will have the required U-D sequence.

For example, the sequence 1, 0, 0 defines the permutations of the form $p_1 p_2 p_3 p_4$ satisfying $p_1 < p_2$, $p_2 > p_3$, $p_3 > p_4$. This partial ordering can be extended to the linear orderings

$$p_4 < p_3 < p_1 < p_2$$
$$p_4 < p_1 < p_3 < p_2$$
$$p_1 < p_4 < p_3 < p_2$$

giving as possible values for $a(1)a(2)a(3)a(4)$ the permutations

4312
4132
1432

Therefore the inverses of these permutations, namely

3421
2431
1432

are the solutions to the U-D sequence 1, 0, 0.

Based on this concept, we have the following method to generate all solutions to a given U-D sequence $s_1, s_2, \ldots, s_{n-1}$.

(i) Construct the partial ordering R on the set of integers $\{1, 2, \ldots, n\}$ such that iRi+1 if $s_i = 1$
and i+1Ri if $s_i = 0$.

(ii) Find all topological sorting solutions.

(iii) Construct the inverses of the topological sorting solutions, giving the required solutions to the U-D sequence.

We now consider two algorithms to generate U-D sequence solutions based on two of the topological sorting algorithms that were discussed in Chapter 2.

4.1.1 An algorithm based on the Knuth-Szwarcfiter algorithm

The Knuth-Szwarcfiter algorithm for generating topological sortings given in Figure 2.4 requires two minor modifications in order to be transformed into the algorithm to generate U-D sequence solutions shown in Figure 4.2. Firstly, the input to the algorithm of Figure 4.2 is not a partial ordering, but a U-D sequence contained in array SEQ, and the partial ordering used in this algorithm is constructed from this U-D sequence. The second modification to the original topological sorting algorithm, is the replacement of the statement

```
P[i]:= j;
```

appearing in Figure 2.4, with the statement

```
P[j]:= i;
```

The change resulting from this replacement is that the array P, instead of containing each topological sorting solution as it is generated, will now contain the inverse of each such solution. So the U-D sequence solutions will be generated directly from the partial ordering, and the intermediate step of explicitly computing the topological sortings is not necessary.

From Chapter 2 we know that for a partial ordering represented by m pairs iRj on a set of n elements, the Knuth-Szwarcfiter algorithm needs O(m+n) time to generate each solution. Now a U-D sequence of length n-1 always defines a partial ordering which we represent by n-1 pairs, and therefore the algorithm of Figure 4.2 requires O(n) time per solution generated.

```
procedure GENERATE_KS(SEQ,n):
begin S:= {1,2,...,n};
      construct the partial ordering R on S such that
      for i:= 1,2,...,n iRi+1 if SEQ[i] = 1
                    and i+1Ri if SEQ[i] = 0;
      M:= {y|there is no x such that xRy);
      UPDOWN_KS(1);
end;


procedure UPDOWN_KS(i):
begin if i = n then P[n]:= sole element of M; PROCESS(P);
        else for j ∈ M do
               begin erase all pairs of the form jRk;
                     P[j]:= i;
                     UPDOWN_KS(i+1);
                     retrieve all pairs of the form jRk;
               end;
     endif;
return;
```

Figure 4.2 An algorithm to generate U-D sequence solutions

4.1.2 An algorithm based on Varol's algorithm

The algorithm based on the Knuth-Szwarcfiter algorithm computes U-D solutions directly from the partial ordering, skipping out the step of computing explicitly the corresponding topological sortings. For any algorithm based on Varol's algorithm this step must be included, since each topological sorting solution generated (with the exception of the starting solution) is constructed from the previous solution.

An algorithm based on Varol's algorithm is given in Figure 4.3. The array P is used to store topological sorting solutions as they are generated, the array Q is used to store the corresponding U-D solutions, and the array SEQ contains the U-D sequence $s_1, s_2, \ldots, s_{n-1}$ as its first n-1 entries and 1 as its n-th entry.

Recall that Varol's algorithm requires an initial topological sorting solution (the starting solution) which is renamed 12...n. Making use of the particular nature of the type of partial ordering defined by a U-D sequence, we have the following method to compute a starting solution in linear time. (For an arbitrary partial ordering no such linear method exists).

Consider the sequence $S = s_1, s_2, \ldots, s_n$ where $s_1, s_2, \ldots, s_{n-1}$ is the given U-D sequence, and $s_n = 1$.

(1) Scan the sequence S from left to right, and output i for each $s_i = 1$ as it is encountered.

(2) Scan the sequence S from right to left, and output i for each $s_i = 0$ as it is encountered.

For example, given the U-D sequence 1, 0, 0, 1, 0 we get the sequence S = 1, 0, 0, 1, 0, 1. Scanning S from left to right, we encounter 1's in positions 1, 4, 6 and so we output 146; then scanning S from right to left, we encounter 0's in positions 5, 3, 2 and so we output 532. The complete output is therefore the permutation 146532.

This permutation is a solution to the topological sorting problem 1R2, 3R2, 4R3, 4R5, 6R5 defined by the U-D sequence 1, 0, 0, 1, 0 and its inverse, 165243 is a solution to this U-D sequence.

```
procedure GENERATE_V(SEQ,n):
begin STARTING_SOLUTION(H);
      for i:= 1 to n do Q[H[i]]:= i;
      construct the partial ordering R on {1,2,...,n) such that
      for i:= 1,2,...,n Q[i] R Q[i+1] if SEQ[i] = 1
                    and Q[i+1] R Q[i] if SEQ[i] = 0;
      UPDOWN_V;
end;


procedure UPDOWN_V:
begin for i:= 1 to n do P[i]:= LOC[i]:= i;
      PROCESS(Q);
      i:= 1;
      while i < n do
        begin if i is not blocked
                  then j:= right-hand neighbour of i;
                       transpose i and j;
                       LOC[i]:= LOC[i]+1; i:= 1;
                       CONVERT(P,H,Q);
                       PROCESS(Q);
                  else ROTATE(i);
                       LOC[i]:= i; i:= i+1;
              endif;
        end;
return;
```

Figure 4.3 An algorithm to generate U-D sequence solutions

It is easy to see that this method of finding a starting solution to the topological sorting problem defined by the U-D sequence $s_1, s_2, \ldots, s_{n-1}$ is valid. For $i = 1,2,\ldots,n-1$, if $s_i = 1$, we have the pair $iRi+1$ and in the permutation constructed, i indeed appears to the left of $i+1$, and similarly if $s_i = 0$. (Note that it is not important that $s_n = 1$, the method is still valid if $s_n = 0$). Although this method has been described in terms of two scans of the s_i sequence one scan only is necessary, as shown in the STARTING_SOLUTION procedure given in Figure 4.4.

```
procedure STARTING_SOLUTION(H):
begin up:= 1; down:= n;
      for i:= 1 to n do
        begin if SEQ[i] = 1 then H[up]:= i;
                                 up:= up+1;
                            else H[down]:= i;
                                 down:= down-1;
              endif;
        end;
return;
```

Figure 4.4 Procedure to find a starting solution

The permutation constructed by the procedure STARTING_SOLUTION to be used as the starting solution is stored in array H. It is convenient to view array H as a mapping of 12...n onto the permutation constructed. The inverse mapping H^{-1} represents the renaming function (i.e. it maps the starting solution onto 12...n). Array Q, (which is later used to store the U-D sequence solutions) is used temporarily in procedure GENERATE_V to represent this renaming function H^{-1} for the purpose of computing the set of pairs iRj of the renamed partial ordering.

The procedure UPDOWN_V is simply the same as procedure TOPSORTS_V of Varol's generating algorithm in Figure 2.5, with the addition of a call to a procedure CONVERT appearing just before the second call to the PROCESS macro. The procedure CONVERT will convert a renamed topological sorting solution to the corresponding U-D sequence solution by (i) translating the renamed topological sorting back to its original name, and then (ii) inverting this permutation. That is, if the array P contains the renamed topological sorting solution, the procedure CONVERT computes array Q such that

$$Q[i]:= (H[P[i]])^{-1} \quad \text{for } i = 1,\ldots,n$$

or equivalently,

$$Q[i] = j \quad \text{if} \quad H[P[i]] = j, \quad j = 1,2,\ldots,n.$$

A call to procedure CONVERT is not required prior to the first call to PROCESS, since at this stage array Q contains the inverse of the starting solution which is in array H, that is Q already contains the U-D sequence solution corresponding to the starting topological sorting solution.

Consider for example, the U-D sequence 0, 1, 0, 0, 1. This sequence defines the partial ordering

2R1	5R4
2R3	5R6
4R3	

The procedure STARTING_SOLUTION constructs the array H containing the permutation 256431 to be used as the starting solution to the topological sorting. This permutation is renamed 123456 and the partial ordering is renamed giving

$H^{-1}(2)$ R $H^{-1}(1)$	$H^{-1}(5)$ R $H^{-1}(4)$
$H^{-1}(2)$ R $H^{-1}(3)$	$H^{-1}(5)$ R $H^{-1}(6)$
$H^{-1}(4)$ R $H^{-1}(3)$	

that is

1R6	2R4
1R5	2R3
4R5	

All solutions to the renamed topological sorting problem are now generated. For example, a particular solution is 216435; translated by the mapping H to its original name we get 521463, and inverting this permutation gives the U-D sequence solution 326415.

The computation of U-D sequence solutions by procedure CONVERT requires the recomputation of all n entries of array Q for each solution generated, however successive solutions usually differ from each other in a couple of positions only. (For instance, in the previous example, the first two U-D solutions generated are 615423 and 625413 which differ from one another in two positions only). Modifying the procedure UPDOWN_V of Figure 4.3 to obviate the constant recomputation of array Q results in the improved procedure UPDOWN_I shown in Figure 4.5.

```
procedure UPDOWN_I:
begin for i:= 1 to n do P[i]:= LOC[i]:= i;
      PROCESS(Q);
      i:= 1;
      while i < n do
        begin if i is not blocked
                 then j:= right-hand neighbour of i;
                      transpose i and j;
                      Q[H[j]]:= LOC[i]; Q[H[i]]:= LOC[i]+1;
                      LOC[i]:= LOC[i]+1; i:= 1;
                      PROCESS(Q);
                 else ROTATE(i);
                      k:= LOC[i];
                      while k ≥ i do
                        begin Q[H[P[k]]:= k;
                              k:= k-1;
                        end;
                      LOC[i]:= i, i:= i+1;
              endif;
        end;
return;
```

Figure 4.5 Improved version of the procedure UPDOWN_V

The procedure UPDOWN_V of Figure 4.3 has been altered so that whenever the contents of array P are changed, the corresponding changes to array Q are made. Firstly, when a topological sorting is computed from the preceding one by a simple transposition of a pair of elements of array P, the corresponding U-D sequence solution is computed by a transposition of elements of Q. This is achieved using the statements

```
Q[H[j]]:= LOC[i];
Q[H[i]]:= LOC[i]+1;
```

in Figure 4.5 which replace the call to procedure CONVERT.

Eliminating this procedure results in a saving of O(n) time per solution generated, but there are now extra overheads. In order to keep Q "up-to-date" with P at every step of the algorithm, whenever cyclic right-rotations are performed on P between the generation of successive topological sortings, the corresponding operations must be performed on array Q. This is achieved by the "while" loop that follows the call to procedure ROTATE in Figure 4.5. Rotating the elements of P requires a simple cyclic right-rotation of the elements in positions i,i+1,...,LOC[i], an operation which can be performed very efficiently on some computers. The corresponding operations on the elements H[P[i]], H[P[i+1]],....,H[P[LOC[i]] of array Q may not be as simple, since these elements are not necessarily contiguous.

Let us now compare the times complexities of the procedures UPDOWN_V (Figure 4.3) and UPDOWN_I (Figure 4.5). Suppose that a U-D sequence of length n-1 defines a topological sorting problem which has t solutions, and that r cyclic right-rotations of elements of array P are executed in the process of generating these solutions. Both of the procedures UPDOWN_V and UPDOWN_I will require O(r+t) time to generate these topological sortings since, as we saw in Chapter 2, this is the time complexity of Varol's generating algorithm.

To convert these t topological sortings into U-D sequence solutions, the procedure UPDOWN_V requires t-1 calls to the O(n) CONVERT procedure giving a total conversion time of O(nt).

The procedure UPDOWN_I generates every topological sorting solution (except the starting solution) from the previous one by transposing elements of P, and so the corresponding t-1 transpositions of elements of Q require 2c(t-1) time, for some constant c. The operations performed on array Q corresponding to the rotations of elements of array P are exactly those operations that would be performed on P if ROTATE were not a primitive operation, but a procedure requiring time proportional to the sequences being rotated, and from Chapter 2 we know that the total time required by ROTATE if it is not primitive is 2d(r+t-1). So the total time required by procedure UPDOWN_I for all operations on Q, and therefore its total conversion time is $2c(t-1) + 2d(r+t-1) = 2dr + 2(c+d)t - 2(c+d)$ which is O(r+t).

From Chapter 2 we know that $r < nt$ and so UPDOWN_V has time complexity $O(r+t) + O(nt) = O(nt)$, and UPDOWN_I has time complexity $O(r+t) + O(r+t) = O(r+t)$. So clearly UPDOWN_I is the better procedure, since it is at least as fast, if not faster than UPDOWN_V.

The method used in procedure STARTING_SOLUTION in Figure 4.4 to construct a starting topological sorting solution in linear time is by no means unique. An interesting alternative method is as follows:

As before, consider the sequence $S = s_1, s_2, \ldots, s_n$ where $s_1, s_2, \ldots, s_{n-1}$ is the given U-D sequence and $s_n = 1$.

(1) Partition the sequence S into subsequences $C_1, C_2, \ldots, C_r$ by placing a vertical line at either end of S, and also after every $s_i = 1$. For each subsequence C_m, $m = 1,2,\ldots,r$ perform step (2).

(2) Output $k, k-1, \ldots, j+1, j$ where $C_m = s_j, s_{j+1}, \ldots, s_{k-1}, s_k$.

For example given the U-D sequence 0,0,1,1,0,0,0 we have S = 0,0,1, 1,0,0,0,1 which is partitioned as follows:

$$|0,0,1|1|0,0,0,1|.$$

That is,

$$C_1 = 0,0,1 \quad = s_1, s_2, s_3$$
$$C_2 = 1 \quad = s_4$$
$$C_3 = 0,0,0,1 \quad = s_5, s_6, s_7, s_8$$

Starting with C_1 we output 321, then going on to C_2 we output 4, and finally reaching C_3, we output 8765, and so the complete output is the permutation 32148765. This permutation is a solution to the topological sorting problem defined by the U-D sequence 0,0,1,1,0,0,0.

A procedure for this method is shown in Figure 4.6. We can show that this method will always construct a solution to the topological sorting problem defined by the given U-D sequence, by using the same argument we used to prove that the previous method worked correctly.

```
procedure ALTERNATIVE_STARTING_SOLUTION(H):
begin j:= 1;
      for i:= 1 to n do
        begin if SEQ[i] = 1 then
                    k:= i;
                    while j ≤ i do
                      begin H[j]:= k; j:= j+1; k:= k-1; end;
              endif;
        end;
return;
```

Figure 4.6 A second method to construct a starting solution

The interesting point about this second method is that the topological sorting solution constructed is the smallest solution (in lexicographical ordering) and it is an *involution*, that is a permutation which is its own inverse. Since this permutation is an involution, it is also a solution to the U-D sequence defining the topological sorting problem, and so this method can be viewed as a constructive proof that every U-D sequence has as a solution at least one involution.

We can easily see that $p_1p_2\cdots p_n$, the permutation constructed by this method is an involution, since for each subsequence $C_m = s_j, s_{j+1},\cdots,s_{k-1},s_k$ we have that

$$\begin{aligned} p_k &= j \\ p_{k-1} &= j+1 \\ &\vdots \\ p_{j+1} &= k-1 \\ p_j &= k \end{aligned}$$

That is $p_a = b \iff p_b = a$ for $a,b = 1,2,\ldots,n$. Further, it can be seen that the permutation will be the smallest topological sorting solution (in lexicographical ordering), by noting that at each stage in the construction of the permutation $p_1p_2\cdots p_n$, the choice for p_i

is the smallest possible integer that can be selected without contradicting the restrictions of the partial ordering.

4.1.3 Enumeration of U-D sequence solutions

The number of permutations having a given U-D sequence can be computed by applying Wells' enumerating algorithm (Chapter 3) to the topological sorting problem defined by the U-D sequence. More direct enumeration techniques are given by Niven [1968] and Foulkes [1976]. The following result, which is from Niven [1968] can be used to design an algorithm to enumerate U-D sequence solutions.

Let $S = s_1, s_2, \ldots, s_{n-1}$ be a U-D sequence. Let $k_1, k_2, \ldots, k_r$ be the subscripts of those s_i's having value 0 where $k_1 < k_2 < \ldots < k_r$. If S consists of a sequence of 1's then r = 0. Then N(S), the number of solutions to the U-D sequence S equals the determinant of order r+1 with the binomial coefficient $\binom{k_i}{k_{j-1}}$ at the intersection of row i and column j (i,j = 1,2,...,r) where $k_0 = 0$, $k_{r+1} = n$ and $\binom{m}{t} = 0$ if $m < t$. For example, if S = 1, 0, 1, 1, 0, 1, 1, 1 then $k_1 = 2$, $k_2 = 5$, $k_3 = 9$ and

$$N(S) = \begin{vmatrix} \binom{2}{0} & \binom{2}{2} & \binom{2}{5} \\ \binom{5}{0} & \binom{5}{2} & \binom{5}{5} \\ \binom{9}{0} & \binom{9}{2} & \binom{9}{5} \end{vmatrix} = \begin{vmatrix} 1 & 1 & 0 \\ 1 & 10 & 1 \\ 1 & 36 & 126 \end{vmatrix} = 1099$$

4.2 Permutations with a given number of runs

If we place a vertical line at both ends of a permutation $p_1 p_2 \cdots p_n$ and also between p_j and p_{j+1} whenever $p_j > p_{j+1}$, the segments in between pairs of lines are called runs. For example, the permutation

|269|7|48|13|

has 4 runs.

Now a permutation with k runs has a U-D sequence with k-1 0's. So the problem of generating all permutations of $\{1,2,\ldots,n\}$ having k runs can be solved by finding all $\binom{n-1}{k-1}$ U-D sequences which have k-1 0's (and n-k 1's), and then generating all the solutions to each of these U-D sequences by reducing them to topological sorting problems. A far more efficient algorithm can be designed by making use of the following symmetry.

Lemma 4.1 If $a_1a_2\ldots a_n$ is a permutation of $\{1,2,\ldots n\}$ having the U-D sequence $S = s_1,s_2,\ldots,s_{n-1}$ and if $b_i = n+1-a_{n+1-i}$ for $i = 1,2,\ldots,n$ then the permutation $b_1b_2\ldots b_n$ has the reverse U-D sequence $S^* = s_{n-1},s_{n-2},\ldots s_1$.

Proof: Let $T = t_1,t_2,\ldots,t_{n-1}$ denote the U-D sequence of the permutation $b_1b_2\ldots b_n$. If $s_i = 1$, then $a_i < a_{i+1}$, so $b_{n-i} < b_{n+1-i}$ and therefore $t_{n-i} = 1$. Similarly if $s_i = 0$ then $t_{n-i} = 0$. Hence $T = S^* = s_{n-1},s_{n-2},\ldots,s_1$. (Q.E.D.)

So if the solutions to a U-D sequence $S = s_1,s_2,\ldots,s_{n-1}$ have been generated, by reversing each solution, and replacing every element i by n+1-i, the solutions to the reverse U-D sequence $S^* = s_{n-1},s_{n-2},\ldots,s_1$ are obtained.

For example given the solutions

3421
2431
1432

to the U-D sequence 1,0,0 reversing each solution and replacing each i by 5-i gives

4312
4213
3214

which are the solutions to the U-D sequence 0,0,1.

It is easy to see that it is always cheaper to generate the solutions to S* directly from the solutions to S in this way, than by

using any U-D sequence solution generating algorithm. Consider any such generating algorithm, and assume that an array P is used to store each solution as it is generated. The algorithm must contain assignment statements of the form

P[i]:= j;

By inserting after each such statement, the statement

R[n+1-i]:= n+1-j;

the solutions to the reverse U-D sequence are obtained at a cost of executing these assignment statements only which is obviously cheaper than executing the whole algorithm again.

By dividing all the $\binom{n-1}{k-1}$ U-D sequences of length n-1 having k-1 0's and n-k 1's into pairs (S, S*) where S* is the reverse of S we therefore have the following algorithm to generate all permutations of {1, 2, ..., n} which have k runs:

(1) For each pair (S, S*), generate the U-D sequence solutions to S using a U-D sequence solution generating algorithm.

(2) If S $\neq$ S*, then generate the solutions to S* by computing them directly from the solutions to S. (If S = S*, then obviously the solutions to S are the solutions to S* as well, and must not be generated a second time).

4.2.1 Enumeration of permutations with a given number of runs

The problem of generating all permutations with a given number of runs can be transformed into a set of U-D sequence solution problems, and these in turn can be transformed into a set of topological sorting problems. So permutations with a given number of runs can be enumerated by application of Wells' enumerating algorithm for topological sorting. As for the case of enumerating U-D sequence solutions, a more direct enumeration technique exists.

Let <n,k> denote the number of permutations of {1, 2, ..., n} having k runs. These numbers <n,k>, known as Eulerian numbers, can be com-

puted from the following formula that appears in Knuth [1973, p.37]:

$$\langle n,k\rangle = k^n - (k-1)^n\binom{n+1}{1} + (k-2)^n\binom{n+1}{2} - \ldots + (-1)^k 0^n\binom{n+1}{k}$$

$$= \sum_{j=0}^{n} (-1)^j (k-j)^n \binom{n+1}{j}, \qquad n \geq 0,\ k \geq 0.$$

CHAPTER 5 - PROBLEMS FOR FURTHER RESEARCH

Some specific problems related to the topics covered in this dissertation that may be considered for further research are:

1) Designing new generating algorithms

If the Knuth-Szwarcfiter and Varol's algorithms for generating topological sortings are modified by removing from them all references to partial orderings, we are left with two algorithms that generate all n! permutations of n objects.

The Knuth-Szwarcfiter algorithm in fact becomes the standard backtracking permutation generation algorithm (see Wells [1971]), and Varol's algorithm becomes the permutation generation algorithm attributed to M. Hall Jnr. (see Ord-Smith [1970]).

New algorithms for generating topological sortings may possibly be designed by modifying other permutation generation algorithms so that they generate only those permutations consistent with a given poset. (During the past twenty years, over thirty different permutation generation algorithms have been published - see for example Sedgewick [1977]).

2) Improving Varol's generating algorithm

Varol's algorithm contains a measure of arbitrariness in that the choice of starting solution is not completely specified - any one of the topological sortings may be used. In Section 2.2.2 we showed how different starting solutions may result in different running times for the same problem, and we considered a possible method of selecting the starting solution, but no conclusive results were obtained.

A possible area of research is to firstly find a way of choosing the starting solution that will always (or perhaps on average) give us maximum efficiency, and secondly to determine whether more time is saved by using the best starting solution, than is used in finding it.

3) Designing new enumerating algorithms

As far as this writer knows, Wells' algorithm (Chapter 3) is the only topological sorting enumerating algorithm that exists. A new enumerating algorithm may possibly be designed by making use of Varol's algorithm as follows.

There is a simple method to construct the final topological sorting generated by Varol's algorithm, which is:

> Starting with the permutation 12....n, for i = n-1,n-2,....,1 move element i rightwards until it is blocked, that is until either iRj for j the right-hand neighbour of i, or until i is the right-most element in the permutation.

The permutation obtained in this way is obviously the last one generated by the algorithm, since every element less than n is blocked, and so at this point the algorithm will execute n-1 successive rotations and terminate.

For example, consider the restrictions 1R5, 2R4 and 4R5 on the set {1,2,3,4,5}. (The order in which the topological sortings of this poset are generated by Varol's algorithm is shown in Figure 2.6). Starting with 12345, we first attempt to move element 4 rightwards, but it is blocked by 5 and so nothing changes. Element 3 is then moved all the way to the right giving 12453. Since 2R4, element 2 cannot move further right and so nothing changes, and then finally element 1 is moved rightwards until it is blocked by 5, and we get the permutation 24153, which is indeed the last topological sorting generated by Varol's algorithm.

Therefore the enumeration of topological sortings of a poset can be transformed into the problem of indexing the solutions generated by Varol's algorithm, that is the problem of computing the integer k, if the k-th solution generated by the algorithm is given, since the index of the final solution is equal to the total number of topological sortings generated.

It should be noted that the problem of indexing the solutions generated by Varol's algorithm is not a trivial one. Further, because of

the nature of the algorithm, whereby each solution generated is constructed from the previous one, it is possible that the cost of indexing the final solution may not be significantly cheaper than actually generating the solutions themselves.

4) Improving Wells' enumerating algorithm

In Chapter 3 we saw that the method used in Wells' algorithm for choosing the cleavage vertex does not guarantee maximum efficiency. A possible research problem is to find a method for choosing the cleavage vertex which will give maximum efficiency, and to find out also, if the time required by this method for finding the best cleavage vertex would justify our using it instead of the original method.

5) Samples of posets

In Section 2.3 we discussed the problem of how to define a good sample of posets, and the related problem of how these posets should be characterized. Solving these problems would be extremely useful, as it would make empirical testing of any algorithms concerned with topological sorting more meaningful.

6) Other restricted permutation problems

In Chapter 4 we discussed two restricted permutation problems that can be reduced to topological sorting. An area for further research would be to find out whether there are other restricted permutation problems that can also be transformed into the topological sorting problem.

LIST OF REFERENCES

AHO, A.V., HOPCROFT, J.E. and ULLMAN, J.D. [1974]. The Design and Analysis of Computer Algorithms. Reading, Mass.: Addison-Wesley.

FOULKES, H.O. [1976]. "Enumeration of Permutations with Prescribed Up-Down and Inversion Sequences". Discrete Mathematics 15: 235-252.

HOROWITZ, E. and SAHNI, S. [1976]. Fundamentals of Data Structures. Woodland Hills, Calif.: Computer Science Press Inc.

KAHN, A.B. [1962]. "Topological Sorting of Large Networks". Communications of the ACM 5(11): 558-562.

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

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

KNUTH, D.E. and SZWARCFITER, J.L. [1974]. "A Structured Program to Generate all Topological Arrangements". Information Processing Letters 2: 153-157.

NIJENHUIS, A. and WILF, H.S. [1975]. Combinatorial Algorithms. New York, N.Y.: Academic Press Inc.

NIVEN, I. [1968]. "A Combinatorial Problem of Finite Sequences". Nieuw Archief voor Wiskunde 3: 116-123.

ORD-SMITH, R.J. [1970]. "Generation of Permutation Sequences: Part 1". The Computer Journal 13(2): 152-155.

REINGOLD, E.M., NIEVERGELT, J. and DEO, N. [1977]. Combinatorial Algorithms: Theory and Practice. Englewood Cliffs, N.J.: Prentice-Hall Inc.

SEDGEWICK, R. [1977]. "Permutation Generation Methods". Computing Surveys 9(2): 137-164.

SZWARCFITER, J.L. and WILSON, L.B. [1978]. "Some Properties of Ternary Trees". The Computer Journal 21(1): 66-72.

TARJAN, R. [1974]. "Finding Dominators in Directed Graphs". SIAM Journal on Computing 3(1): 62-89.

VAROL, Y.L. and ROTEM, D. [1977]. "Permutation Generation, Indexing and Topological Sorting". Applied Mathematics Preprint No. 3, Series A. Johannesburg: University of the Witwatersrand.

WAITE, W.M. [1967]. "An Efficient Procedure for the Generating of Closed Subsets". Communications of the ACM 10(3): 169-171.

WELLS, M.B. [1971]. Elements of Combinatorial Computing. Elmsford, N.Y.: Pergamon Press.

WELLS, M.B. [1980]. Personal communication.

WIRTH, N. [1976]. Algorithms + Data Structures = Programs. Englewood Cliffs, N.J.: Prentice-Hall Inc.