Abstract:

A structured recursive procedure for generating all solutions of the topological sorting problem is presented, together with an efficient machine-oriented, non-recursive translation.

To appear in the Information Processing Letters.

A Structured Program to Generate All Topological Sorting Arrangements

by

Donald E. Knuth
and
Jayme L. Szwarcfiter

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 61
May, 1974

Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.

bibliographical details

KNUTH, Donald E.

A structured program to generate all topological sorting arrangements. [By] Donald E. Knuth [and] Jayme L. Szwarcfiter.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1974.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 61).

12"

Added entries

SZWARCFITER, Jayme Luiz
UNIVERSITY OF NEWCASTLE UPON TYNE. Computing Laboratory. Technical Report Series. 61

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th): 511.6
U.D.C.: 519.1

Suggested keywords

COMBINATORIAL PROBLEMS
DATA STRUCTURES
PROGRAMMING LANGUAGES

Abstract

A structured recursive procedure for generating all solutions of the topological problem is presented, together with an efficient machine-oriented, non-recursive translation.

About the Author

Professor Knuth is from the Computer Science Department, Stanford University. This research has been supported in part by the National Science Foundation, grant GJ 36473X, and by the Office of Naval Research, Contract NR 044-402.

Mr. Szwarcfiter is from the Universidade Federal do Rio de Janeiro, and currently working for the Ph.D. at the Computing Laboratory, University of Newcastle upon Tyne, supported by the Conselho Nacional de Pesquises, Brazil.

A STRUCTURED PROGRAM TO GENERATE ALL TOPOLOGICAL SORTING ARRANGEMENTS

An algorithm for topological sorting was presented by Knuth [4] as an example of typical interaction between linked and sequential forms of data representation. The purpose of the present note is to extend the algorithm so that it generates all solutions of the topological sorting problem: the extended algorithm serves as an instructive example of several important general issues related to backtracking, procedures for changing recursion into iteration, manipulation of data structures, and the creation of well-structured programs.

Given a number n and a set of integer pairs (i,j), where $1 \le i,j \le n$, the problem of topological sorting is to find a permutation $x_1 x_2 \ldots x_n$ of $\{1,2,\ldots,n\}$ such that i appears to the left of j for all pairs (i,j) that have been input. It is convenient to denote input pairs by writing the relation " $i \prec j$ " and saying " i precedes j ".

The topological sorting problem is essentially equivalent to arranging the vertices of a directed graph into a straight line so that all arcs go from left to right. It is well known that such an arrangement is possible if and only if there are no oriented cycles in the graph, i.e., if and only if no $k \geq 1$ relations of the form

$$i_1 < i_2, i_2 < i_3, \ldots, i_k < i_1$$

exist in the input. The problem in mathematical terms is to embed a given partial order into a linear (total) order.

A natural way to solve this problem is to let x_1 be an element having no predecessors, then to erase all relations of the form $x_1 < j$ and to let x_2 be an element $\neq x_1$ with no predecessors in the system as it now exists, then to erase all relations of the form $x_2 < j$, etc. It is not difficult to verify (cf. [4]) that this method will always succeed unless there is an oriented cycle in the input. Moreover, in a sense it is the only way to proceed, since x_1 must be an element without predecessors, and x_2 must be without predecessors when all relations $x_1 < j$ are deleted, etc. This observation leads naturally to an algorithm that finds all solutions to the topological sorting problem; it is a typical example of a "backtrack" procedure [2,3], where at every stage we consider a subproblem of the form "Find all ways to complete a given partial permutation $x_1 x_2 \ldots x_k$ to a topological sort $x_1 x_2 \ldots x_n$." The general method is to branch on all possible choices of x_{k+1}.

A central problem in backtrack applications is to find a suitable way to arrange the data so that it is easy to sequence through the possible choices of x_{k+1}; in this case we need an efficient way to

discover the set of all elements $\neq \{x_1,\ldots,x_k\}$ which have no predecessors other than $x_1,\ldots,x_k$, and to maintain this knowledge efficiently as we move from one subproblem to another. It is not satisfactory merely to make a brute-force search through all n possibilities for x_{k+1}, since this typically makes the program on the order of n times slower than a method which avoids such searching.

The following procedure, written in an ALGOL-like language using "abstract" data structures (cf. Hoare [1]), shows how to solve the problem with only order n additional units of storage. The procedure is to be invoked by a main driver program of the form "read and prepare the input; alltopsorts(0)".

```
procedure alltopsorts(k); integer k; value k;
   comment This procedure will output all topological sorting
   arrangements which begin with a sequence x_1...x_k that has already
   been output. Let R = {1,2,...,n}\{x_1,...,x_k} be the set of all
   vertices not yet output; the procedure assumes that, for all yεR,
   the current value of global variable count[y] is the number of
   relations z ≺ y for zεR, and that there is a linear list D
   containing precisely those elements yεR such that count[y] = 0.
   The execution of this procedure may cause temporary changes to the
   contents of D and the count array, but both will be restored to
   their entry values upon exit;
   begin integer q, base;
      if D not empty then
      begin base := rightmost element of D;
         repeat set q to rightmost element of D
            and delete it from D;
            erase all relations of the form q ≺ j;
            output q in column k+1;
            if k = n-1 then newline;
            alltopsorts(k+1);
            retrieve all relations of the form q ≺ j;
            insert q at the left of D;
         until rightmost element of D = base;
      end
end
```

The operations of erasing and retrieving relations will respectively decrease and increase appropriate entries of the count array, and they will also respectively insert and delete elements at the right of list D .

Thus, if we suppose that D contains $y_1 \cdots y_r$ on entry to alltopsorts, for $r \geq 1$, the procedure will set base := y_r , then q := y_r . Then it will decrease count[j] by 1 for each variable j such that $y_r < j$ was input; and if $z_1, \ldots, z_s$ are the values of j whose count drops to zero at this time, D will be changed to $y_1 \cdots y_{r-1} z_1 \cdots z_s$. After outputting y_r , and doing alltopsorts beginning with $x_1 \ldots x_k y_r$, the count array is restored to its initial condition and D is changed to $y_r y_1 \cdots y_{r-1}$. The same process will occur again with $q = y_{r-1}, y_{r-2}, \ldots, y_1$, until finally all sortings will have been produced and D will again be $y_1 \cdots y_r$; then exit from alltopsorts occurs. These remarks amount to a proof by recursion induction that the algorithm is correct, since termination is an obvious consequence of the fact that D contains at most n-k entries.

Note that D operates as an output-restricted deque, since all deletions from D occur at its right and all insertions occur at its ends. Therefore we can represent D as a list with one-way linking. (It may be of interest to note that the authors' original algorithm took base and q from the left of D while inserting $z_1 \ldots z_s$ and q at the right; this made D an input-restricted deque, so that two-way linking was originally needed. Thus, a slight perturbation of the algorithm removed the need for one link.) Our program below uses an array link[0:n] to hold the pointers for D , which will be a circular list; a simple variable D points to the leftmost element, and link[D] points to the rightmost.

The erasure and retrieval operations are not difficult but they require some comment. It is clear that a natural way to represent the input relations for this purpose is to have a list for each i of all j such that $i \prec j$. If there are m input relations in all, entering in random order, we can handle this as in [4] by having integer arrays top[1:n], suc[1:m], next[1:m], such that top[i] is the index p of the first relation for i, and (if $p \neq 0$) suc[p] is the corresponding j value and next[p] is the index of the next such relation. The erasing operation is now easily programmed.

In order to convert the recursion to iteration, we will need a stack for the local variables q and base. (It is not necessary to save return addresses on the stack, since return is always to one place when $k > 0$ and to another when $k = 0$; furthermore it is unnecessary to save k on the stack since it is easily updated across calls.) This suggests that we introduce arrays q[1:n] and base[1:n]. However, since the count entries are zero for all items q on the stack, it is possible to save n locations by keeping an implicit "q stack" in the count array; thus, a simple variable t contains the value of q at the level just outside the current one, and count[t] holds the value on the next level, etc.

There are two more refinements which will speed up the program. First, we can test for D empty before actually calling the procedure, thereby maintaining D as a nonempty list throughout the process; this is a big advantage, since empty circular lists are always very awkward. Secondly, we can realize that the procedure alltopsorts is called most often when $k = n-1$, and it can be greatly simplified in that case.

Putting these observations together yields the following efficient machine-oriented program.

```
procedure generate all topological sorting arrangements (m,n);
   integer m,n; value m,n; comment  m relations on n elements;
begin integer array count,top[1:n];
   integer array suc,next[1:m];
   integer array link,base[0:n];
   integer q,k,i,j,p,t,D,D1;
  read and prepare the input:
   for j := 1 step 1 until n do count[j] := top[j] := 0;
   for k := 1 step 1 until m do
   begin read(i,j);
      suc[k] := j; next[k] := top[i];
      top[i] := k; count[j] := count[j]+1;
   end;
   link[0] := D := 0;
   for j := 1 step 1 until n do
   begin if count[j] = 0 then
      begin link[D] := j; D := j end
   end;
   if D = 0 then go to done else link[D] := link[0];
   k := 0; t := 0;
 alltopsorts: if k = n-1 then begin print(D) in column:(n);
      newline end else
   begin base[k] := link[D];
      repeat set q to rightmost and delete:
         q := link[D]; D1 := link[q];
        erase relations beginning with q:
         p := top[q];
         while p ≠ 0 do
            begin j := suc[p]; count[j] := count[j]-1;
               if count[j] = 0 then
                  begin if D = q then D := j else link[j] := D1; D1 := j end;
               p := next[p];
            end;
         link[D] := D1;
```

```
        print(q) in column:(k+1);
      recursive call:
        if D1 = q then begin comment input contains an oriented
           cycle and D is empty; go to done end;
        count[q] := t; t := q;
        k := k+1; go to alltopsorts;
      return when k positive: k := k-1;
        q := t; t := count[q]; count[q] := 0;
      retrieve relations beginning with q:
        p := top[q];
        while p ≠ 0 do
           begin j := suc[p]; count[j] := count[j]+1; p := next[p];
           end;
      insert q at left: comment link[q] is still set properly;
           link[D] := q; D := q;
    until link[D] = base[k];
  end;
  if k > 0 then go to return when k positive;
done: end.
```

The example input

$$1 < 3 \ , \ 2 < 1 \ , \ 2 < 4 \ , \ 4 < 3 \ , \ 4 < 5$$

will cause this program to print the five solutions

```
2 1 4 3 5
      5 3
  4 5 1 3
    1 3 5
      5 3
```

Note that redundant printing has been suppressed. Alternatively, of course, the statement ' print(q) in column:(k+1) ' could have been replaced by ' buffer[k+1] := q ', and 'newline' replaced by 'printline(buffer)'.

This program is efficient in the sense that its inner loops are reasonably fast, it uses only $O(m+n)$ words of memory, and it takes

at most $O(m+n)$ units of time per output. However, there may be tremendous amounts of output, since the number of topological sortings for large n is often very large. For example, when there are no input relations at all, all $n!$ permutations are produced. (Note that the number of digits printed in this worst case is not $n \cdot n!$, but $n + n(n-1) + n(n-1)(n-2) + \ldots + n! = \lfloor n!e \rfloor - 1$, about e per permutation on the average.)

The volume of output and the running time can be reduced to $O(n \cdot 2^n)$, if $O(2^n)$ more memory cells are allotted, by modifying the recursive procedure so that it "remembers" similar situations. We can add a new global variable corresponding to the current value of the set $\{x_1, \ldots, x_k\}$, and have the procedure alltopsorts remember which sets it has seen before, and where this occurred in the output. Whenever a set is repeated, the output can now be replaced by a simple cross-reference to the appropriate line. Thus, the above example would reduce to

```
1:  2 1 4 3 5
2:        5 3
3:    4 5 1 3
4:      1 ..  see line 1
```

Adding memory to recursive procedures is, of course, a standard way to speed them up. Compare the exponential running time of

```
integer procedure F(n); if n ≤ 1 then n else F(n-1) + F(n-2)
```

to the linear running time of

```
for n := 0 step 1 until N do F[n] := if n ≤ 1 then n else F[n-1] + F[n-2].
```

Our program contains the statement "go to return when k positive", which jumps into the repeat loop! Some people consider this disgraceful; the detailed discussion in [5] points out that this go to is not really harmful, because it has been obtained by straightforward modification of the original recursive program that was easily proved correct. The complete documentation of a program should include its abstract form and a discussion of the essentially mechanical transformations which led to the final optimized form. It is possible to remove this go to, but there would be no advantage in this case. (See [5] for further examples and discussion.)

References

[1] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, London and New York, 1972.

[2] Robert W. Floyd, "Nondeterministic algorithms," J.ACM 14 (1967), 636-644.

[3] Solomon W. Golomb and Leonard D. Baumert, "Backtrack programming," J.ACM 12 (1965), 516-524.

[4] Donald E. Knuth, Fundamental Algorithms, The Art of Computer Programming 1 (Addison-Wesley, 1968; second edition, 1973), 634 pp.

[5] Donald E. Knuth, "Structured programming with go to statements," in preparation.