BIT 10 (1970), 76–94

A TRANSITIVE CLOSURE ALGORITHM

PAUL PURDOM JR.

Abstract.

An algorithm is given for computing the transitive closure of a directed graph in a time no greater than $a_1N_1n+a_2n^2$ for large n where a_1 and a_2 are constants depending on the computer used to execute the algorithm, n is the number of nodes in the graph and N_1 is the number of arcs (not counting those arcs which are part of a cycle and not counting those arcs which can be removed without changing the transitive closure). For graphs where each arc is selected at random with probability p, the average time to compute the transitive closure is no greater than $\min\{a_1pn^3+a_2n^2, \frac{1}{2}a_1n^2p^{-2}+a_2n^2\}$ for large n. The algorithm will compute the transitive closure of an undirected graph in a time no greater than a_2n^2 for large n. The method uses about n^2+n bits and $5n$ words of storage (where each word can hold $n+2$ values).

1. Introduction.

The transitive closure T of a directed graph G is a directed graph such that there is an arc in T going from node i to node j if and only if there is a path in G going from node i to node j. The transitive closure of a node i is the set of nodes on paths starting from node i. For example the transitive closure of node k in Figure 1 is the set of nodes $\{g,l,j,k,h\}$. It is often useful to specify a graph G with nodes $1,2,\ldots,n$ by an $n\times n$ incidence matrix M with elements m_{ij} defined by

$$m_{ij} = \begin{cases} \textbf{true} \text{ if } G \text{ has an arc from node } i \text{ to node } j\,, \\ \textbf{false} \text{ otherwise}\,. \end{cases}$$

It has long been known that the incidence matrix M of a graph can be used to compute the incidence matrix T of the transitive closure of the graph with the equation

$$T = \sum_{1\leq i\leq n} M^i$$

where M and T are boolean matrices. It takes about n^4 operations to compute T this way. Warshall [1] has a method to compute the transitive closure which takes between n^2 and n^3 operations. His algorithm to con-

Received January 30, 1969; revised September 12, 1969.

vert the incidence matrix M of a graph into the incidence matrix of the transitive closure of G is equivalent to the following:

W1. For $1 \leqq k \leqq n$ do the remaining steps.
W2. For each i such that $1 \leqq i \leqq n$ and $M[i,k]$ is true do step W3.
W3. For $1 \leqq j \leqq n$ set $M[i,j] \leftarrow M[i,j]$ *OR* $M[k,j]$.

A method for computing transitive closure using lists is given by Thorelli [2]. His method, however, will in many cases take about n^4 operations if the transitive closure has about n^2 arcs (with minor changes his algorithm can be done in n^3 steps.)

There are many algorithms which require the computing of transitive closure. The reader is referred to Weber and Wirth [3] and Lynch [4] for some practical problems in the field of syntactic analysis where it is necessary to find the transitive closure of a graph with one or two hundred nodes.

The algorithm in this paper is designed for computing the transitive closure of a graph with a moderately large number of nodes (the graph should, however, fit in the computer storage; this requires about n^2 bits of memory for a graph with n nodes). In section 2 it is shown that the maximum running time for the algorithm is proportional to n^3, but there are cases (such as sparse graphs where the number of arcs is no more than a constant times the number of nodes, random graphs where each possible arc is selected with fixed probability, and undirected graphs) when the running time increases only as n^2. In section 4 the method is compared with Warshall's algorithm and cases are given where the method in this paper will be faster for large graphs.

The concepts of path equivalence and partial ordering are particularly important to understanding the algorithm. Two distinct nodes x and y are path equivalent if there is both a path from x to y and a path from y to x. Also each node is path equivalent to itself. For any pair of nodes x and y, there is a path from any node path equivalent to x to any node path equivalent to y if and only if there is a path from x to y. In the following the term equivalent always refers to path equivalence. A directed graph is a partial ordering if and only if the graph has no cycles. Thus if no pair of distinct nodes in the graph are equivalent, the graph is a partial ordering. If the graph is a partial ordering it is possible to find a consistent linear ordering of the nodes [5]. This means that the nodes $1, 2, \ldots, n$ can be renumbered as $i_1, i_2, \ldots, i_n$ in such a way that if there is an arc from x to y then i_x precedes i_y.

The algorithm consists of four parts. The first part finds all the classes of nodes which are equivalent and replaces each class by a single node.

The nodes for the classes are connected to each other according to whether or not they contain nodes which are connected in the original graph. Figure 1 shows a graph, and figure 2 shows the results of replacing each

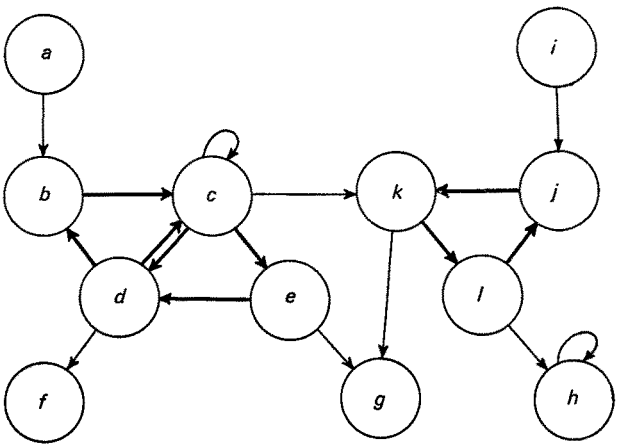

Figure 1. A directed graph with 12 nodes and 18 arcs. Arcs which connect pairs of nodes in the same path equivalence class are shown as dark arrows. Arcs which connect pairs of nodes in different equivalence classes and arcs which connect nodes to themselves are shown as light arrows.

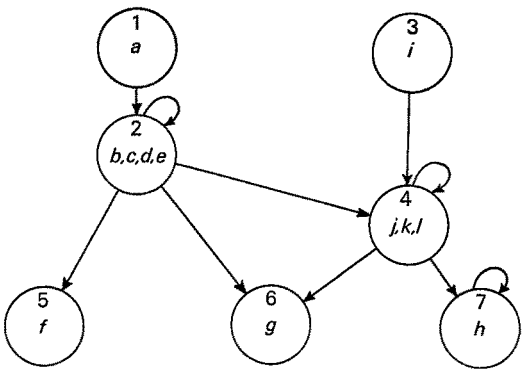

Figure 2. The graph from Figure 1 after the path equivalence classes have been replaced by single nodes. Part 1 of the algorithm combines those nodes which are members of the same path equivalence class into a single node. Thus nodes *b*, *c*, *d*, and *e* are now represented as a single node as are nodes *j*, *k*, and *l*. Part 2 of the algorithm finds a linear ordering of the nodes such that if there is an arc from one node to another, the second node has a higher number than the first. The linear ordering found by the algorithm is shown by the numbers in each node.

class by a node. Once each class is replaced by a single node the resulting graph is a partial ordering (if the cycles of length one are ignored). The second part of the algorithm finds a linear ordering of the nodes consistent with the partial ordering. Figure 2 shows the results of the ordering. The third part computes the transitive closure of the graph of equivalence classes. It computes the transitive closure for one node at a time starting with the last node in the ordering and working back to the first. To form the transitive closure of a node, x, it takes each node with an arc from x and each node in the transitive closure of the nodes with an arc from x. It is possible to compute the transitive closure this way because the ordering of the nodes ensures that the transitive closure of a node is computed before it is needed to compute the transitive closure for another node. Figure 3 shows the graph after the algorithm has computed

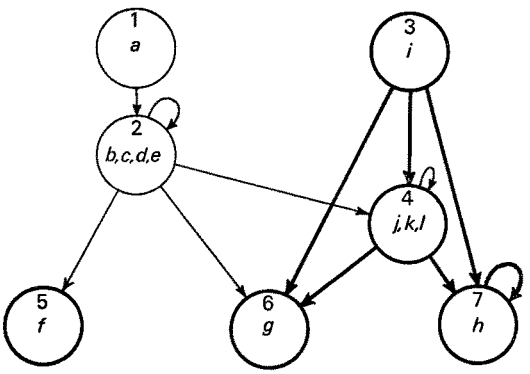

Figure 3. The graph being processed by part 3 of the algorithm. The dark nodes have been processed. The dark arcs form the transitive closure of the processed nodes. The algorithm is ready to compute the transitive closure for node 2 now that it has computed the transitive closure for all nodes after 2 in the linear ordering. The transitive closure for node 2 consists of all nodes to which there is an arc from node 2 (4, 5 and 6), and all nodes in their transitive closure (6 and 7).

the transitive closure for nodes 7, 6, 5, 4, and 3. Figure 4 shows that graph after the entire transitive closure has been computed for the path equivalence classes. The fourth part of the algorithm is quite simple. For a pair of nodes x and y an arc is added from x to y if and only if x is in a class which has an arc (in the transitive closure graph for the equivalence class) to the class which contains y.

The details in the algorithm are given in an Algol procedure in the appendix.

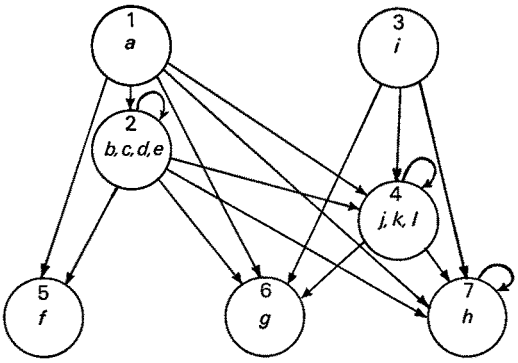

Figure 4. The transitive closure of the graph of path equivalence classes. Part 3 of the algorithm produces the transitive closure of the graph in which each equivalence class is represented by a single node. This transitive closure is used by part 4 of the algorithm to generate the transitive closure of the original graph by connecting the nodes in each equivalence class to each node in those equivalence classes to which their equivalence class is connected. The transitive closure of the original graph is not shown because of the large number of arcs in the transitive closure graph (71 arcs).

2. Analysis of performance.

A summary of the results of analyzing the time and space required to run the algorithm will be given. It is assumed that the algorithm is run on a computer with a random access storage large enough to hold the algorithm and its data. The analysis is in terms of n (the number of nodes), N (the number of arcs), and m (the number of equivalence classes). The values of N and m are limited by $0 \leqq N \leqq n^2$ and $1 \leqq m \leqq n$. For a graph selected at random from the 2^{n^2} possible graphs the expected value of N is $n^2/2$. The author does not know the expected value of m (See however Palasti [8]). Since calculating how often each step is done is straightforward but tedious for most steps in the algorithm, the results of this analysis is summarized in table 1 and the step numbers are given in the appendix. Additional details are available elsewhere [7].

The analysis for the execution of step 20 will be given in detail since it often dominates the running time for the entire algorithm. Step 20 is entered from step 19 no more times than step 19 is done. Also if N_1 is the number of arcs in the original graph not counting the arcs in cycles and not counting the arcs which can be removed without changing the transitive closure, then step 20 is entered from step 19 no more than N_1 times. Step 20 is done at most $m+1$ times each time it is entered from

step 19. Also it is done at most once for each value of j, i, and k such that $0 \leqq j < i < k \leqq m$. Thus it is done at most

$$\min \left\{ N_1(m+1), \sum_{0<k\leqq m} \sum_{0\leqq i\leqq k-1} i = \tfrac{1}{6}[m^3 - m] \right\} \text{ times} .$$

For the graph with nodes $1, 2, \ldots, n$ where each node from 1 to $n/3$ has an arc to each node from $n/3+1$ to n (and where there are no other arcs) step 20 will be done $\frac{2}{27}n^2(n+1)$ times. The author does not know if there are graphs which take more time. If the graph is selected by taking each of the possible n^2 arcs in the graph independently with probability p, then step 20 is done no more than $\min\{pn^3, \frac{1}{2}[n^2p^{-2} - np^{-2}]\}$ times on the average. It can be done an average of no more than pn^3 times because the graph has an average of pn^2 arcs and it is done no more than n times each time it is entered. To see the second part of the limit notice that to find whether there is a path from the kth node in the linear ordering to the jth node (where $k<j$) the algorithm at step 20 tests each node i where $k<i<j$ and where there is an arc from k to i. It starts with the smallest and continues until it finds a path from i to j or until all such i have been tested. The probability that b values of i (where $b \leqq j-k-2$) are investigated is

$$\begin{aligned} P(a=b) &= P \text{ (there is no path from } i_a \text{ to } j \text{ for } 1 \leqq a < b \\ &\qquad \text{and there is a path from } i_b \text{ to } j) \\ &\leqq P \text{ (there is no arc from } i_a \text{ to } j \text{ for } 1 \leqq a < b) . \end{aligned}$$

The probability of no arc from i_{a1} to j is independent of the probability of no arc from i_{a2} to j if $i_{a1} \neq i_{a2}$. The probability of no arc from i_a to j is no more than $1-p$ if $i_a > j$ even though the original nodes have been combined into equivalence classes and reordered. Therefore $P(a=b) \leqq (1-p)^{b-1}$. The expected number of searches to try to connect k to j is limited by

$$E(s) \leqq \sum_{1\leqq i\leqq\infty} (1-p)^{i-1} i = p^{-2} .$$

The range of the sum was permitted to go to infinity because all the terms in the sum are positive. Step 20 is done for at most $\frac{1}{2}n(n-1)$ pairs of k and j. Thus the loop is done at most $\frac{1}{2}[n^2p^{-2} - np^{-2}]$ times.

If the graph is undirected so that if there is an arc from i to j then there is also an arc from j to i so the graph of equivalence classes has no arcs and step 20 is not done at all. Of course, if one wishes an algorithm

for finding just the connectivity of undirected graphs, then one can use parts one and four of this algorithm by themselves.

The maximum time for the entire algorithm can be expressed as

$$a \min\{(m+1)N_1, \tfrac{1}{6}n^3\} + \sum_{0 \leqq i,j \leqq 2} a_{ij} n^i m^j$$

where a is the time required to do the loop in step 20 and a_{ij} is the sum of the times required to do all the steps with the factor $n^i m^j$ in the formula for the limit of the number of times the step is done (weighting each time in the sum by the coefficient of $n^i m^j$ in the formula). The limit for the average time is the same with $\min\{(m+1)N_1, \frac{1}{6}n^3\}$ replaced by $\min\{pn^3, \frac{1}{2}n^2p^{-2}\}$.

The algorithm requires n^2 bits for storing the M array. Storing linear arrays requires $5n$ words (for words which can hold numbers from 0 to $n+1$) assuming the *Next*1 and *Count* arrays shares space with *Stack* or *Onstack*) and n bits. The rest of the program requires a constant amount of storage.

4. Comparison with Warshall's Algorithm.

Warshall's algorithm is much simpler. It always takes less space although this is usually not important for graphs with a large number of nodes since the ratio of the space requirements for the two methods approaches one as the number of nodes increases. The time required for Warshall's algorithm can be expressed as $a_0 + a_1 n + a_2 n^2 + a_3 n N_i$ with $N \leqq N_i \leqq N_t$ where the a's are constants, n is the number of nodes, N is the number of arcs in the original graph, and N_t is the number of arcs in the transitive closure. Since the steps in Warshall's algorithm which are done n^2 times or less are much simpler than those steps for the algorithm in this paper, Warshall's algorithm should always be faster for graphs with a small number of nodes and for graphs where N_i is not larger than n by a large factor.

For any set of graphs where the number of arcs increases faster than the number of nodes the time for Warshall's algorithm will increase faster than n^2. Thus for graphs where each possible edge is selected randomly with fixed probability p (or with a probability $p(n)$ which depends on the number of nodes where $\lim_{n\to\infty} np(n) = \infty$) the algorithm in this paper will be faster than Warshall's algorithm if the graph has enough nodes. Table 2 presents the time required for running each algorithm

on the Burroughs 5500 computer with randomly selected graphs (these tests were done with no other programs running on the computer so that the time the multiprogramming system spends switching between jobs would not have a significant effect on the running times). Tests were done both with graphs where each arc was selected at random with fixed probability and with graphs where there were no arc from node i to node j if $i > j$ but for $i \leqq j$ each possible arc was selected at random with fixed probability. This second set of graphs were included to show how the algorithm in this paper performs on graphs which do not have a large number of path equivalence classes. Runs were made with various probabilities and numbers of nodes to show cases where each algorithm was faster. The reader is warned that the time either algorithm takes for a random graph may not be related to the time the algorithm will take for the problems he is interested in.

The author suspects that the algorithm in the paper will be faster than Warshall's algorithm for nearly all sets of graphs where the transitive closure has close to n^2 arcs providing the graph is one which the algorithm in the paper can do in a time proportional to n^2 (and providing n is large enough). The algorithm in this paper, however, definitely is not faster in all such cases since Warshall's algorithm, for example, will be faster for a graph which has arcs from each node to the last node and arcs from the last node to each node and no other arcs. Additional details are available elsewhere [7].

4. Variations on the Algorithm.

Many details of the algorithm have been selected to make it easier to understand while others have been selected arbitrarily. Possible variations of the algorithm which some readers may wish to consider will be suggested.

It is possible to combine together the first three parts. Combining together parts 2 and 3 is quite easy. Knuth [5] gives the basic idea needed for combining parts 1 and 2 together.

If the elements in the matrix M are rearranged in part 2 (and restored after part 3), then indexing can be used in place of the *Next* and *Previous* lists. The *Next*1 list can also be eliminated. One then will have an algorithm where the loop in step 20 will be quite similar to the inner loop in Warshall's algorithm but will be done only $\frac{1}{6}$ as often. The resulting algorithm will not be as fast for randomly selected graphs.

On many computers it is possible to *OR* together two computer words

at once. For such computers step 20 can be modified to treat many values of i at once. (This is often done in programs for Warshall's algorithm). To do this one would wish to reorder the matrix M in part 2.

There are often many linear orderings consistent with a partial ordering. Perhaps there is an ordering algorithm more suitable than the one used for part 2 of this algorithm.

If a copy of the Count array produced at step 10 is saved, then near the bottom of step 19 one could add the substep, if $Count[i]=0$, then repeat this step.

If one uses n^2 words for storing the matrix M then it is possible to make further use of list processing techniques. See Knuth [5] for an example of how this can be used in part 2 of the algorithm.

Acknowledgments.

The author wishes to thank Stephen Stigler for his helpful discussion of some of the statistical results. He wishes to thank the University of Wisconsin Research Committee and the University of Wisconsin Computing Center for providing time for this work.

Table 1

Step (substep)	*Number of times (max)*	*Step (substep)*	*Number of times (max)*
1	1	15	1
1 (loop)	n	15 (outer loop)	$m+1$
2	$m+1$	15 (inner loop)	$\frac{1}{2}[m^2+3m]$
3	n	15 (2^{nd} if-part) + 14	m
4	$\frac{1}{2}[3n^2+n-m^2+m]$	16	1
5	$\frac{1}{2}[3n^2+n-m^2-3m]$	17	1
6 + 9	n	18	$m+1$
6 (loop) + 9	n	18 (loop)	$\frac{1}{2}[m^2+m]$
7 + 9	n	19	$\frac{1}{2}[m^2+m]$
7 (outer loop)	$\frac{1}{2}[n^2+n-m^2-m]$	20	$\min\{\frac{1}{6}[m^3-m], N_1 m+N_1\}$
7 (inner loop)	$\frac{1}{2}[n^2+n-m^2-m]$	20 [average]	$\min\{pn^3, \frac{1}{2}[n^2p^{-2}-np^{-2}]\}$
8 + 9	n	20 [undirected]	0
10	1	21	$\frac{1}{2}[m^2-m]$
10 (outer loop)	$m+1$	22	1
10 (inner loop)	m^2+m	23	$m+1$
11	1	24	m^2+m
12 + 14	$m+1$	25	$nm+m$
13	$m+1$	26	n^2
		27	$nm+m$

An upper limit (not always sharp) is given for the number of times the algorithm does each step, each group of steps (indicated by step numbers connected by a plus sign), or part of a step (indicated by giving the part in parenthesis). For the loop in step 20 an upper limit for the number of times, an upper limit for the average number of times, and the number of times for an undirected graph, is given. The formulas are in terms of

n—the number of nodes, m—the number of equivalence classes, N_1—the number of arcs, which are not part of cycles and which can not be removed without changing the transitive closure, and p—the probability of an arc.

Table 2

Method	Probability	Number of Nodes					
		10	20	30	40	50	60
Warshall	.5	11 9.2(0.9) 8	76 73.0(2.6) 68	257 251.2(3.6) 246	607 606 a 605		
Paper	.5	7 5.8(0.6) 5	12 11.9(0.3) 11	23 21.9(0.7) 21	37 35.5(0.9) 34		
Warshall	.2	8 4.4(1.6) 3	62 53.2(4.9) 46	233 204.3(17.7) 179	516 495.5 b 480		
Paper	.2	9 6.4(1.1) 5	12 11.5(0.5) 11	22 21.5(0.7) 20	36 34.4(1.4) 32		
Warshall	.1	3 1.9(0.6) 1	40 25.4(9.2) 12	147 114.5(21.6) 80	426 347.2(55.2) 250		
Paper	.1	9 7.3(0.8) 6	16 13.0(1.8) 11	23 21.7(0.8) 21	37 33.5(2.1) 30		
Warshall	.05	3 1.5(0.9) 0	15 8.5(2.7) 6	61 32.6(11.5) 22	165 113.6(36.8) 67		
Paper	.05	10 7.1(1.1) 6	18 17.2(0.8) 16	36 31.9(4.7) 23	46 40.3(4.2) 35	64 52.6(4.7) 47	75 73 b 71
Warshall	.5 if $i \leqq j$	6 4.5(0.7) 4	37 34.8(2.0) 32	125 119.3(3.4) 113	291 279.5(6.5) 270		
Paper	.5 if $i \leqq j$	10 8.1(0.7) 7	21 20.5(0.5) 20	42 41.1(0.6) 40	70 69.6(0.5) 69		

Table 2 (continued)

Method	Probability	Number of Nodes					
		10	20	30	40	50	60
Warshall	.2 if $i \leqq j$	3 2.1(0.6) 1	23 18.0(3.7) 12	84 77.2(7.4) 64	224 183.8(24.0) 148		
Paper	.2 if $i \leqq j$	8 7.5(0.5) 7	19 18.6(0.5) 18	39 37.8(0.6) 37	65 64.1(1.0) 62		
Warshall	.1 if $i \leqq j$	2 1.2(0.4) 1	12 8.7(2.4) 5	50 34.8(9.3) 21	129 82.6(21.2) 59		
Paper	.1 if $i \leqq j$	8 7.2(0.4) 7	18 17.3(0.7) 16	36 34.5(1.1) 33	61 58.7(1.5) 56		
Warshall	.05 if $i \leqq j$	2 1.0(0.5) 0	6 4.5(1.0) 3	19 15.2(3.1) 9	45 33.0(7.6) 24		
Paper	.05 if $i \leqq j$	9 7.0(0.8) 6	17 16.2(0.6) 15	33 32.0(1.1) 30	55 53.7(1.1) 52		

a) Results for 3 graphs.
b) Results for 4 graphs.

The table shows the amount of processor time (in $\frac{1}{60}$ths of a second) required by each method ro run on the Burroughs 5500 computer. The graphs were selected at random with each arc (from node i to node j) subject to the probability and condition (if any) shown in the probability column. The two methods, however, did their calculations on the same graphs. In each case ten graphs were tried unless a footnote indicates otherwise. The top and bottom lines for each method give the maximum and minimum times the method took for the cases (usually ten cases). The middle line gives the average time and the $\sqrt{n/(n-1)}$ times the observed standard deviation where n is the number of graphs which were tested. Since the clock measured time in $\frac{1}{60}$ths of a second, the timing process contributed at least 0.4 to the standard deviation. The version of Warshall's algorithm used stored one matrix element per word. If the matrix had been packed Warshall's algorithm would have been faster by a factor of $[n/47]^{-1}n$, where $[x]$ is the smallest integer greater than or equal to x.

Appendix.

```
procedure TRANSCLOSURE (m, n); value n; integer n; boolean
  array m;
comment This algorithm takes an n × n incidence matrix M for a directed
  graph and converts it into the incidence matrix for the transitive closure
  of the graph;
begin
  integer array next[0:n+1]; previous[0:n+1]; equivalent[1:n];
  comment Part 1. Eliminate Cycles
    This part of the algorithm finds cycles in the graph, and each time it
    finds a cycle, it replaces the cycle by a single node. The elimination of
    cycles continues until the graph has no cycles. When this part is
    finished each equivalence class has been replaced by a single node. The
    arrays Next and Previous form a doubly linked list of nodes that re-
    main in the graph as nodes if the equivalence classes are removed. The
    array Equivalent has a circular list for each node remaining in the
    graph. Each circular list has the original nodes from one equivalence
    class. The array Stack contains a list of the nodes on the path being
    investigated for cycles. The array Onstack has the position in the stack
    for each node in the stack and zero for each other node. The array New
    is used to indicate which nodes have not yet been removed from the stack;
begin
  integer array stack[1:n], onstack[1:n];
  boolean array new[1:n];
  integer i, k, top, j, b, c, temp, a;
  comment Step 1. Initialize;
  for i := 1 step 1 until n do
  begin
  equivalent[i] := next[i-1] := previous[i+1] := 1;
    onstack[i] := 0; new[i] := true
  end;
  next[n] := previous[1] := 0;
  previous[0] := n; top := k := 0;
  comment Step 2. The paths leading from each node in the graph will
    now be investigated for cycles, except that nodes which have already
    been investigated will be skipped;
  starttree:
    k := next[k];
    if k = 0 then go to order;
    if ¬new[k] then go to starttree;
    i := k;
```

```
    comment Step 3. Paths leading from node i will now be investigated
      to find cycles. Node i is put on the stack;
  stack i:
    top := top+1;
    stack[top] := i;
    onstack[i] := top;
    j := 0;
    comment Step 4. Each arc leading from node i will now be investigated
      unless it leads to part of the graph where all paths have already been
      investigated;
  nextarc:
    j := next[j];
    if j = 0 then go to unstack;
    if i = j ∨ ¬m[i, j] ∨ ¬new[j] then go to nextarc;
    comment Step 5. If node j is already on the stack a cycle has been
      found. Otherwise paths from node j must be investigated;
    if onstack[j] ≠ 0 then go to removecycle;
    i := j;
    go to stack i;
    comment Step 6. Node j and all nodes above it on the stack form a
      cycle. All nodes except j are removed from the list of nodes and set
      equivalent to j (along with any nodes equivalent to them). The nodes
      removed from the list of nodes here are not used in the rest of the algo-
      rithm except in step 7 and in the steps of part 4;
  removecycle: for c := onstack[j]+1 step 1 until top do
    begin
      b := stack[c];
      next[previous[b]] := next[b];
      previous[next[b]] := previous[b];
      temp := equivalent[j];
      equivalent[j] := equivalent[b];
      equivalent[b] := temp
    end;
    comment Step 7. Arcs are now added to node j so that it will have
      the same connections to the rest of the graph that the nodes in the
      loop had. Node j will then be used to represent the entire equivalence
      class;
    a := 0;
    m[j, j] := true;
  combine 1:
    a := next[a]; if a = j then go to combine 1;
```

```
    if a = 0 then go to return;
    for c := onstack[j]+1 step 1 until top do
    begin
      b := stack[c];
      if m[a, b] then
      begin
        m[a, j] := true;
        go to combine 2
      end
    end;
combine 2:
    for c := onstack[j]+1 step 1 until top do
    begin
      b := stack[c];
      if m[b, a] then
      begin
        m[j, a] := true;
        go to combine 1
      end
    end;
    go to combine 1;
    comment Step 8. Now all nodes above j have been removed from the
      stack. The investigation of paths from j is continued, taking care
      not to skip any paths added in step 7;
return:
    top := onstack[j];
    i := j;
    j := 0;
    go to nextarc;
    comment Step 9. All new paths from node i have now been investi-
      gated and all nodes equivalent to i have been found. Node i is now
      removed from the stack and the investigation of paths from nodes
      below i on the stack is continued;
unstack:
    top := top-1;
    new[i] := false;
    if top = 0 then go to starttree;
    j := i;
    i := stack[top];
    go to nextarc;
end of Part 1. Notice that at steps 2 and 4 it was necessary to investigate
```

```
    only paths involving nodes which have not been on the stack and then
    removed. Whenever a node is removed from the stack all paths from that
    node have already been investigated for cycles. Therefore it is not
    necessary to investigate paths involving nodes which have been re-
    moved from the stack;
  comment Part 2. Order Nodes.
    Part 2 takes the graph of equivalence classes (produced by part 1),
    which is a partial ordering, and finds a consistent linear ordering.
    The lists Next and Previous are reordered so that if there is an arc
    from node i to node j then i occurs before j on the Next list (and after j
    on the Previous list). The method used is similar to the ones given by
    Kahn [6] and by Knuth [5];
  order:
    begin
      integer array count[1:n];
      integer j, i, k, a, b;
      comment Step 10. The number of arcs leaving each node is counted
        and stored in the array Count;
      j := 0;
    count1:
      j := next[j];
      if j = 0 then go to start;
      count[j] := 0;
      i := 0;
    count2:
      i := next[i];
      if i = 0 then go to count1;
      if m[j,i] ∧ i ≠ j then count[j] := count[j]+1;
      go to count2;
      comment Step 11. Now i is set to the head of the list of nodes
        and k is set to the end of the new list of nodes (the new list will be
        ordered);
    start:
      i := 0;
      k := n+1;
      comment Step 12. Advance j;
    advancej:
      j := i;
      comment Step 13. Check for successors;
    checksuccessors:
      i := previous[i];
```

```
    if i = 0 then go to startqueue;
    if count[i] ≠ 0 then go to advancej;
    comment Step 14. Each node with no sucessors is added to the new
      list and removed from the old;
    previous[k] := i;
    previous[j] := previous[i];
    next[previous[i]] := j;
    next[i] := k;
    k := i;
    i := j;
    go to checksuccessors;
    comment Step 15. The index i goes from front to back on the new
      list of nodes. Each node which has an arc to node i and has other
      arcs only to nodes which are after i on the new list of nodes is now
      added to the back of the queue;
startqueue:
    i := n+1;
    previous[k] := 0;
process1:
    i := previous[i];
    if i = 0 then go to outorder;
    a := 0;
process2:
    b := a;
    a := previous[a];
    if a = 0 then go to process1;
    if m[a, i] then
    begin
      count[a] := count[a]-1;
      if count[a] = 0 then
      begin
        previous[k] := a;
        next[a] := k;
        previous[b] := previous[a];
        next[previous[a]] := b;
        previous[a] := 0;
        k := a; a := b
      end
    end;
    go to process2;
    comment Step 16. Move list head;
```

```
    outorder:
      next[0] := k;
      previous[0] := previous[n+1];
      next[previous[n+1]] := 0;
    end of Part 2;
    comment Part 3. Transitive Closure
      Part 3 computes the transitive closure for the graph of equivalence
      classes starting with the last node on the new list of nodes (produced in
      Part 2). At all times the transitive closure will be available for the
      nodes after the one being worked on since each node has arcs connect-
      ing it only to nodes which occur after it on the list. Therefore the transi-
      tive closure of a node k can be computed by taking the union of k
      and the transitive closure of all nodes i for which there is an arc
      from k to i;
  transitiveclosure:
      begin
      integer k, i, j, oldj;
      integer array next1[0:n];
      comment Step 17. Initialize;
      k := 0;
      comment Step 18. Move k one place closer to the front of the list of
        nodes and make a copy of the list of nodes after k;
    nextnode:
      k := previous[k];
      if k = 0 then go to output;
      i := j := k;
    nextnode 1:
      if j = 0 then go to testarc;
      next1[j] := next[j];
      j := next[j];
      go to nextnode 1;
      comment Step 19. Find the next node i on the list such that there is
        an arc from node k to node i;
    testarc:
      i := next1[i];
      if i = 0 then go to nextnode;
      if ¬m[k, i] then go to testarc;
      j := i;
      comment Step 20. Find the next node j on the list such that j is in
        the transitive closure of i;
    testclosure:
```

```
      oldj := j;
      j := next1[j];
      if j = 0 then go to testarc;
      if ¬m[i, j] then go to testclosure;
      comment Step 21. Add an arc from k to j to the transitive closure
        matrix and remove j from the list Next1 so that we do not make
        additional tests for k connected to j in the transitive closure;
      m[k, j] := true;
      next1[oldj] := next1[j];
      j := oldj;
      go to testclosure
    end of Part 3;
    comment Part 4. Output
      The transitive closure of the graph of equivalence classes (computed
      in Part 3) is now expanded to give the transitive closure of the original
      graph. For each pair of equivalence classes i and j where there is an
      arc from i to j in the transitive closure an arc is added to the transitive
      closure for each pair of nodes a and b where a is equivalent to i and b
      is equivalent to j;
  output:
    begin
      integer i, j, a, b;
      comment Step 22. Begin;
      i := 0;
      comment Step 23. New i;
    newi:
      i := next[i];
      if i = 0 then go to endalgorithm;
      j := 0;
      comment Step 24. New j;
    newj:
      j := next[j];
      if j = 0 then go to newi;
      if ¬m[i, j] then go to newj;
      a := i;
      comment Step 25. More a;
    morea:
      a := equivalent[a];
      b := j;
      comment Step 26. New b;
```

```
    newb:
      b := equivalent[b];
      m[a, b] := true;
      if b = j then go to newa;
      go to newb;
      comment Step 27. New a;
    newa:
      if a = i then go to newj;
      go to morea
    end of Part 4;
endalgorithm:
  end of TRANSCLOSURE;
```

REFERENCES

1. Warshall, Stephen, *A Theorem on Boolean Matrices*, JACM 9 (1962), 11–12.
2. Thorelli, Lars-Erik, *An Algorithm for Computing All Paths in a Graph*, BIT 6 (1966), 347–349.
3. Wirth, Niklaus and Weber, Helmut, *Euler: A Generalization of ALGOL, and its Formal Definition*, CACM 9, 13–25 and 89–99.
4. Lynch, W. C., *Ambiguities in BNF Languages* thesis, Univ. of Wisconsin, 1963 and *A High-Speed Parsing Algorithm for ICOR Grammars*, 1968, Report No. 1097, Computing Center, Case Western Reserve University.
5. Knuth, Donald, *The Art of Computer Programming*, Vol. 1, Addison–Wesley, Reading, Mass., 1968, pp. 258–268.
6. Kahn, A. B., *Topological Sorting of Large Networks*, CACM 5 (1962), 558–562.
7. Purdom, Paul W., *A Transitive Closure Algorithm*, July 1968, Computer Sciences Technical Report #33, University of Wisconsin.
8. Palasti, I., *On the strong Connectedness of Directed Random Graphs*, Studia Scientiarum Mathematicarum Hungarica 1 (1966), 205–214.

COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN
MADISON, WISCONSIN
U.S.A.