A TRANSITIVE CLOSURE ALGORITHM

by

Paul W. Purdom

Computer Sciences Technical Report #33

July 1968

A TRANSITIVE CLOSURE ALGORITHM

Abstract

An algorithm is given for computing the transitive closure of a directed graph in a time no greater than $a_1 N_1 n + a_2 n^2$ 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 $\min\{a_1 pn^3 + a_2 n^2, \frac{1}{2} a_1 n^2 p^{-2} + a_2 n^2\}$ for large n. The algorithm will compute the transitive closure of an undirected graph in a time no greater than $a_2 n^2$ 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 transative 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, ..., n by an n x n incidence matrix M with elements m_{ij} defined by

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

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

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

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 convert the incidence matrix M of a gr ph into the incidence matrix of the transitive closure of G is equivalent to the following:

W1. For $\underline{1} \le \underline{k} \le \underline{n}$ do the remaining steps.

W2. For each i such that $1 \le \underline{i} \le \underline{n}$ and $\underline{M}[\underline{i}, \underline{k}]$ is true do step W3.

W3. For $1 \le \underline{j} \le \underline{n}$ set $\underline{M}[\underline{i},\underline{j}] \leftarrow \underline{M}[\underline{i},\underline{j}]$ $\underline{\text{OR}}$ $\underline{M}[\underline{k},\underline{j}]$.

A method for computing transitive closure using lists is given by Lars-Erik Thorelli[2]. His method, however, will in many cases take about n^4 operations if the transitive closure has about n^2 arcs.

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 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 means 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 node 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 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 insures that the transitive clsoure 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 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 .

2. THE ALGORITHM

This algorithm takes an n x n incidence matrix M for a directed graph and converts it into the incidence matrix for the transitive closure of the graph. An ALGOL procedure for the method is given in the appendix.

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 remain in the graph as nodes in 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.

Step 1. (Initialize). For $1 \le i \le n$ set Equivalent [i] ← i , Next [i - 1] ← i , Previous [i + 1] ← i , Onstack [i] ← 0 , and New [i] ← true. Set Next [n] ← 0, Previous [1] ← 0, Previous [0] ← n, top ← 0 and k ← 0 .

Step 2. (Start tree). 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. Set k ← Next [k]. If $k = 0$ then go to Step 10 (the start of Order Nodes). If New [k] is false then repeat this step. Set i ← k .

Step 3. (Stack). Paths leading from node i will now be investigated to find cycles. Node i will now be investigated to find cycles. Node i is put on the stack. Increase top by 1 . Set Stack [top] ← i , Onstack [i] ← top and j ← 0 .

Step 4. (Next arc). Each arc leading from node $\underline{i}$ will now be investigated unless it leads to part of the graph where all paths have already been investigated. Set $\underline{j} \leftarrow$ Next [$\underline{j}$]. If $\underline{j} = 0$ then go to step 9 (Unstack). If $\underline{i} = \underline{j}$, $\underline{M}$[$\underline{i}$, $\underline{j}$] is false or New [j] is false then repeat this step.

Step 5. (Check for cycles). If node $\underline{j}$ is already on the stack a cycle has been found. Otherwise paths from node $\underline{j}$ must be investigated. If Onstack [$\underline{j}$] $\neq 0$ then go to step 6 (Remove cycle). Otherwise set $\underline{i} \leftarrow \underline{j}$ and go to step 3 (Stack).

Step 6. (Remove cycles). Node $\underline{j}$ and all nodes above it on the stack form a cycle. All nodes except $\underline{j}$ are removed from the list of nodes and set equivalent to $\underline{j}$ (along with any nodes equivalent to them). The nodes removed from the list of nodes here are not used in the rest of the algorithm except in step 7 and in the steps of part 4. For Onstack [$\underline{j}$] $<$ $\underline{c} \leq$ top set $\underline{b} \leftarrow$ Stack [$\underline{c}$], Next [Previous [$\underline{b}$]] $\leftarrow$ Next [b], Previous [Next [$\underline{b}$]] $\leftarrow$ Previous [$\underline{b}$], and exchange Equivalent [$\underline{j}$] $\longleftrightarrow$ Equivalent [$\underline{b}$].

Step 7. (Combine). Arcs are now added to node $\underline{j}$ so that it will have the same connections to the rest of the graph that the nodes in the loop had. Node $\underline{j}$ will then be used to represent the entire equivalence class. Set $\underline{a} \leftarrow 0$, $\underline{M}$[$\underline{j}$, $\underline{j}$] $\leftarrow$ true. Repeat the rest of this step until $\underline{a}$ becomes zero again (test for zero each time a is changed). Set $\underline{a} \leftarrow$ Next [$\underline{a}$]. For all $\underline{c}$ where Onstack [$\underline{j}$]

< c ≤ top if M[a, Stack(c)] is true, then set M[a, j] ← true. Also for all c where Onstack[j] < c ≤ top if M[Stack[c], a] is true, then set M[j, a] ← true.

Step 8. (Return). Now all nodes above j have been removed from the stack. The investigation of paths from j is continued by taking care not to skip any paths added in step 7. Set top ← Onstack[j], i ← j, and j ← 0. Go to step 4 (next arc).

Step 9. (Unstack). All new paths from node i have now been investigated 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. Decrease top by 1. Set New[i] ← false. If top = 0 then go to step 2 (Start tree). Otherwise set j ← i, i ← Stack [top], and go to step 4 (Next arc.)

Notice that at steps 2 and 4 it is only necessary to investigate paths involving two 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 removed from the stack.

Part 2. Order Nodes.

Part 2 takes the graph of equivalences 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].

Step 10. (Count Successors). The number of arcs leaving each node is counted and stored in the array Count. Set $j \leftarrow 0$. Repeat the rest of this step until $j = 0$ again (test for zero each time j is changed). Set $j \leftarrow$ Next[j]. Set Count[j] $\leftarrow 0$ and $i \leftarrow 0$. Repeat the rest of the step until $i = 0$ again. Set $i \leftarrow$ Next [i]. If M[j, i] is true and $i \neq j$ then increase Count[j] by 1.

Step 11. (Start). Now i is set to the end of the list of nodes and k is set to the end of the new list of nodes (the new list will be ordered). Let $i \leftarrow 0$ and $k \leftarrow n + 1$.

Step 12. (Advance j). Let $j \leftarrow i$.

Step 13. (Check for successors). Set $i \leftarrow$ Previous [i]. If $i = 0$ then go to step 15 (Start processing queue). If Count [i] $\neq 0$ then go to step 12 (Advance j).

Step 14. (Add to queue). Each node with no successors is added to the new list and removed from the old. Set Previous [k] $\leftarrow i$, Previous [j] $\leftarrow$ Previous [i], Next[Previous [i]] $\leftarrow j$, Next[i] $\leftarrow k$, $k \leftarrow i$, and $i \leftarrow j$. Go to step 13 (Check for successors).

Step 15. (Start processing queue). 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. Set $i \leftarrow n+1$ and Previous [k] ← 0. Repeat this step until $i = 0$ again. Set $i \leftarrow$ Previous [i] and $a \leftarrow 0$. Repeat the rest of this step until $a = 0$. Set $b \leftarrow a$ and $a \leftarrow$ Previous [a]. If M[a, i] is true then reduce Count [a] by 1 and if Count [a] = 0 then set Previous [k] ← a, Next [a] ← k, Previous [b] ← Previous [a], Next [Previous [a]] ← b, Previous [a] ← 0, k ← a, and a ← b.

Step 16. (Move list head). Set Next [0] ← k, Previous [0] ← Previous [n + 1], and Next [Previous [n + 1] ← 0.

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 connecting it only to nodes which occur after it on the list. Therefore the transitive closure of a node _k_ can be computed by taking the union of _k_ and the transitive closures of all nodes _i_ for which there is an arc from _k_ to _i_.

Step 17. (Initialize). Set _k_ to the end of the list of nodes. Set $k \leftarrow 0$.

Step 18. (Next node). Move _k_ one place closer to the front of the list of nodes and make a copy of the list of nodes after _k_. Set k ← Previous [k] If $k = 0$ then go to step 22 (the start of Output). Set $i \leftarrow k$ and $j \leftarrow k$. Repeatedly set Next1 [j] ← Next [j] and j ← Next [j] until $j = 0$.

Step 19. (Test for arc). Find the next node i on the list such that there is an arc from node k to node i. Set i ← Next1 [i]. If i = 0 then go to step 18 (Next node). If M [k, i] is false then repeat this step. Set j ← i.

Step 20. (Test for in node closure). Find the next node j on the list such that j is in the transitive closure of i. Set oldj ← j and j ← Next1 [j]. If j = 0 then go to step 19 (Test for arc). If M [i, j] is false then repeat this step.

Step 21. (Add to closure). Add an arc from k to j to the transitive closure matrix and remove j from the list Next1 so that we don't make additional tests for k connected to j in the transitive closure. Set M [k, j] ← true, Next1 [oldj] ← Next1 [j], and j ← oldj. Go to step 20 (Test for node in closure).

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.

Step 22. (Begin). Set i ← 0.

Step 23. (New i). Set i ← Next [i]. If i = 0 then the algorithm is finished and M contains the incidence matrix for the transitive closure of the original graph. Set j ← 0.

Step 24. (New j). Set $\underline{j} \leftarrow \underline{\text{Next}}[\underline{j}]$. If $\underline{j} = 0$ then go to step 23 (New i). If $\underline{M}[\underline{i}, \underline{j}]$ is false repeat this step. Set $\underline{a} \leftarrow \underline{i}$.

Step 25. (More a). Set $\underline{a} \leftarrow$ Equivalent $[\underline{a}]$ and $\underline{b} \leftarrow \underline{i}$.

Step 26. (New b). Set $\underline{b} \leftarrow$ Equivalent $[\underline{b}]$. Set $M[a, b] \leftarrow$ true. If $\underline{b} = \underline{j}$ then go to step 27 (New a). Repeat this step.

Step 27. (New a). If $\underline{a} = \underline{i}$ then go to step 24 (New j). Go to step 25 (More a).

3. Performance of the Algorithm

Now the time and space required to run the algorithm will be analyzed. 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. It is also assumed that each of the basic operations found in the steps of the algorithm can be done in a constant amount of time or (where upper limits are being discussed) that there is a constant upper limit for the time required for each basic operation.

The analysis will be 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 $0 \le N \le n^2$ and $0 \le m \le n$. 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 . With $N_c = \frac{n}{2} \log n + cn$ for some constant c , the probability that $m(n, N_c) \neq 1$ has the limit $\lim_{n \to \infty} P(m(n, N_c) \neq 1) = e^{-e^{-2c}}$ for a graph selected at random from the $\binom{n^2}{N_c}$ possible graphs with n nodes and N_c edges.[7]

The number of times each step and each part of a step is done will now be considered. The loop in step 20 will be considered in more detail than the other parts of steps since for some graphs it is done much more often (by a factor of n) than the other steps. For many steps only an upper limit on the number of times the step is done will be given. In several cases, smaller upper limits exist than the ones given here. The results of the next four paragraphs are summarized in table 1.

Step 1 is done once. The operations in its loop are done n times. Step 2 is done $m + 1$ times. It is done once for each node (including the fake node zero which is used to mark the end of the list) that is not removed from the list of nodes. It is not done for any node removed from the list of nodes because the bottom node on the stack can not be removed from the list of nodes (see step 6) and a node can be stacked only once (see steps 2, 4, and 5). Step 3 is done n times since each node is stacked one time (see steps 2, 4, and 5). If the value of $\underline{j}$ was not reset at step 8, then step 4 would be done between $n(m + 1)$ and $n(n + 1)$ times. Each time $\underline{j}$ is reset, step 4 may be done an addition i times where i is the number of nodes on the list of nodes at the time. Therefore, the maximum of times step 4 can be done is

$$n(n + 1) + \sum_{m \le i \le n-1} i = \frac{1}{2}[n(3n + 1) - m(m - 1)] .$$

Step 5 is done at least $2m$ times less than step 4 since it is not done for $\underline{j} = \underline{i}$ or for $\underline{j} = 0$ and each of these conditions happens for at least m values of $\underline{i}$. Thus step 5 is done no more than $\frac{1}{2}[3n(n - 1) - m(m - 1)]$ times. Each time step 8 or 9 time is done, at least one node is removed from the stack. Therefore, together they are done at most n times (step 8 and 9 are the only places where a node may be removed from the stack). Step 6 and 7 are done the same number of times as step 8. Each time the operations in the loop of step 6 are done, a node is removed from the stack (at step 8). Thus the number of times the algorithm does either the steps in the loop at step 6 or step 9 is n. Each time step 7 is done, its outer loop will be done once for each node on the list of nodes. Each time the outer loop is done, the inner loop is done the same number of times as the loop in step 6 was done when step 6 was last done. Both the inner and outer loops will be done a maximum number of times if a single node is removed each time step 8 is done. In this case the loops are done $\sum_{m < i \leq n} i = \frac{1}{2}[n(n+1) - m(m+1)]$ times.

Step 10 is done once. If the outer loop is done $m + 1$ times (once for each node (including zero) on the list of nodes). The inner loop is done $m + 1$ times (once for each node including zero) for each time the outer loop is done except for last time. Thus the inner loop is done $m(m+1)$ times. Step 11 is done once. For each of the $m+1$ nodes on the list (including node zero) either step 12 or step 14 is done. Also step 13 is done the same number of times. Step 15 is done once. The outer loop is done once for each of the $m + 1$ nodes (including node zero). The inner loop is entered m times and each time it is done at most $i + 1$ times where i is the number of nodes (including zero) on the old list of nodes). Thus the inner loop is done at most $\sum_{1 \leq i \leq m} i + 1$

$= \frac{1}{2}[m^2 + 3m]$ times. Each time the second part of step 15 is done a node is transferred from the old list to the new list. At least one node is transferred in step 14. Thus the second if-part of step 15 plus step 14 are done at most m times. Step 16 is done once.

Step 17 is done once. Step 18 is done $m+1$ times and the operations in its loop are done $\sum_{1 \le i < m} i = \frac{1}{2}[m(m+1)]$ times. Step 19 is done $\sum_{1 \le i \le m} i = \frac{1}{2}[m(m+1)]$ times. Step 20 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 $0 \le j < i < k \le m$. Thus it is done at most $\min \left\{ N_1(m+1), \sum_{0 < k \le m} \sum_{0 \le i \le k-1} i = \frac{1}{6}[m^3 - m^2] \right\}$ times. For the graph with nodes 1,2, . . .,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 $\min \{pn^3, \frac{1}{2}[n^2 p^{-2} - np^{-2}]\}$ 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 k-th node in the linear ordering to the j-th 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 c. 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 $b \leq j - k - 2$) are investigated as

$$P(a = b) = P\text{ (there is no path from } i_a \text{ to } j \text{ for } 1 \leq a < b$$
$$\text{and there is a path from } i_b \text{ to } j)$$
$$\leq P\text{ (there is no arc from } i_a \text{ to } j \text{ for } 1 \leq a < b).$$

The probability of no arc from i_{a_1} to j is independent of the probability of no arc from i_{a_2} to j if $i_{a_1} \neq i_{a_2}$. 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 $P(a = b) \leq (1-p)^{b-1}$. The expected number of searches to try to connect k to j is limited by

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

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 $\frac{1}{2}[n^2 p^{-2} - np^{-2}]$ 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 then the graph of equivalence classes has no arcs and step 20 is not done at all. Of course, if one wishes an algorithm to just find the connectivity of undirected graphs, then they can use parts one and four of this algorithm by themselves. Step 21 is done only when an arc is added to the transitive closure. Thus it is done at most $\frac{1}{2} m(m-1)$ times.

Step 22 is done once. Step 23 is done $m + 1$ time (once for each equivalence class (including zero)). Step 24 is done $m + 1$ time (once for each equivalence class (including zero)) each of the m times it is entered from step 23. Step 25 is done once for each combination of the $n + 1$ nodes (including zero) and the m equivalence classes. Step 26 is done once for each combination of the $n + 1$ nodes (including zero) and the n nodes. Step 27 is done the same number of times as step 25.

The maximum time for the entire algorithm can be expressed as

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

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 limit on 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 $\min \{(m + 1)N, \frac{1}{6} n^3\}$ replaced by $\min \{pn^3, \frac{1}{2} n^2 p^{-2}\}$.

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 Next1 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 Warhsall's algorithm can be expressed as $a_0 + a_1 n + a_2 n^2 + a_3 n N_i$ with $N \le N_i \le N_t$ 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.

The maximum time required by the algorithm in this paper to compute the transitive closure of a graph with n nodes is less on most computers than the maximum time for Warshall's algorithm. The inner loop of Warshall's algorithm can be done n^3 times whereas the innerloop of this algorithm can be done no more than $1/6\, n^3$ times. The inner loop of Warshall's algorithm consists of incrementing an index, testing the index against a limit, and storing in an array element the result of the OR of two array elements (the algorithm can be modified by replacing the OR and store with a test and store). Step 20 of the algorithm in this paper consists of taking the next element from a list, checking for the end of the lists, and testing an array element. The operations in each loop will take about the same amount of time on most computers.

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 $i \leq j$ 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.

Consider now the time Warshall's algorithm takes to compute the transitive closure for a graph whose nodes have been numbered at random. Also suppose the number of ordered pairs of nodes connected by a path of length k or less, N_k, is greater than $\alpha k \sqrt{N_t}$ for $k \leq \sqrt{2N_t}$ where α is constant for the set of graphs being considered and N_t is the number of arcs in the transitive closure of the graph being considered. Let $M_k[i,j]$ be the i,j element of the M matrix during the k-th execution of the outer loop in Warshall's algorithm. For the graphs being considered

$$P(M_k[i,k]) \geq P(i \text{ connected to } k \text{ by a route through nodes less than } k)$$

$$\geq \sum_{0 \leq \ell < \lfloor \sqrt{2N_t} \rfloor} P(\ell + 1 = \text{length of shortest path from } i \text{ to } k) \binom{n}{k} \Big/ \binom{n}{\ell}$$

$$\geq \sum_{0 \leq \ell < \lfloor \sqrt{2N_t} \rfloor} \frac{\alpha \sqrt{N_t} \binom{k}{\ell}}{n^2 \binom{n}{\ell}}$$

the sum can be simplified as follows:

$$\sum_{0 \le \ell < \lfloor \sqrt{2N_t} \rfloor} \binom{k}{\ell} \Big/ \binom{n}{\ell} = \sum_{0 \le \ell < \lfloor \sqrt{2N_t} \rfloor} \binom{n-\ell}{k-\ell} \Big/ \binom{n}{k}$$

$$= \sum_{0 \le \ell < \lfloor \sqrt{2N_t} \rfloor} \binom{n-\ell}{n-k} \Big/ \binom{n}{k}$$

$$= \sum_{n - \lfloor \sqrt{2N_t} \rfloor < i \le n} \binom{i}{n-k} \Big/ \binom{n}{k}$$

$$= \left[\binom{n+1}{n-k+1} - \binom{n - \lfloor \sqrt{2N_t} \rfloor + 1}{n-k+1} \right] \Big/ \binom{n}{k}$$

$$= \left[\binom{n+1}{k} - \binom{n - \lfloor \sqrt{2N_t} \rfloor - 1}{k - \lfloor \sqrt{2N_t} \rfloor} \right] \Big/ \binom{n}{k}$$

The expected number of times the inner loop of Warshall's algorithm will be done is

$$\sum_{1 \le k \le n} P(M_k[i,k])\, n^2 \ge \sum_{1 \le k \le n} \alpha \sqrt{N_t} \left[\binom{n+1}{k} - \binom{n - \lfloor \sqrt{2N_t} \rfloor - 1}{k - \lfloor \sqrt{2N_t} \rfloor} \right] \Big/ \binom{n}{k}$$

$$\ge \alpha \sqrt{N_t} \sum_{1 \le k \le n} \frac{n+1}{n+1-k} \left(1 - \frac{\binom{n+1 - \lfloor \sqrt{2N_t} \rfloor}{k - \lfloor \sqrt{2N_t} \rfloor}}{\binom{n}{k}} \right)$$

$$\ge \alpha \sqrt{N_t} \left(\sum_{1 \le k \le n} \frac{n+1}{k} - \frac{n+1}{\binom{n+1}{\lfloor \sqrt{2N_t} \rfloor}} \sum_{1 \le k \le n} \binom{k}{\lfloor \sqrt{2N_t} \rfloor} \right)$$

$$\left(\text{since } \frac{n+1}{n+1-k} \le n+1 \text{ for } 1 \le k \le n\right)$$

$$\ge (n+1) \ln n - (n+1) \frac{n+1 - \lfloor \sqrt{2N_t} \rfloor}{\lfloor \sqrt{2N_t} \rfloor + 1}$$

If for a set of graphs $N_t(n) \geq n^2 (\ln n)^{-2+\epsilon}$ for $\epsilon > 0$ then

$$\lim_{n \to \infty} \frac{n^2}{T_w(n)} = 0 \quad \text{where} \quad T_w(n)$$

is the expected time for Warshall's algorithm to compute the transitive closure of a graph from the set with n nodes where the nodes have been numbered at random.

It would be nice if one could now show that there is an α such that $N_k \geq \alpha k \sqrt{N_t}$ for $k \leq \sqrt{2N_t}$ for all graphs. The author believes that this is true for $\alpha = \frac{1}{2}\sqrt{2}$, but he has not been able to find a proof. It is obvious that if the graph consists of several unconnected parts and if $N_k^{(i)} \geq \alpha \sqrt{N_t^{(i)}}$ for each part i then $N_k \geq \alpha \sqrt{N_t}$ for the graph since $\sum_i \sqrt{N_t^{(i)}} \geq \sqrt{\sum_i N_t^{(i)}}$. If the graph consists of one path equivalence class then each node is connected to at least one node by a path of length one, to two nodes by a path of length one or two, and so up to a path whose length is equal to the number of points. Thus for the equivalence class $N_k \geq k\sqrt{N_t/2}$ for $k \leq \sqrt{2N_t}$. Also if the graph consists of a single chain where each node has one successor and one predecessor except the first node has only a successor and the last has only predecessor then $N_k \geq k\sqrt{N_t/2}$ for $k \leq \sqrt{2N_t}$. Furthermore if one has a rooted tree with all the arcs pointing either toward or away from the root such that $N_k \geq k\sqrt{N_t/2}$ for $k \leq \sqrt{2N_t}$, and one removes an arc to a terminal node from a node of distance b from the root and replaces it with an arc to the terminal node from a node of distance c from the root where $c \leq b$ then the new tree has

$$N_k' = \begin{cases} N_k & \text{for } k \le c+1 \\ N_k + c + 1 - k & \text{for } c+1 \le k \le b+1 \\ N_k - a + b & \text{for } a + 1 - k \end{cases}$$

One can easily show then that $N_k' \ge k\sqrt{N_t/2}$. This is all of the cases for which the author has been able to prove the conjecture for $\alpha = \frac{1}{2}\sqrt{2}$. There are several more unimportant cases where the conjecture can be proved for smaller values of α.

To summarize, the following results were found for graphs which have a large enough number of nodes n (see table 2 to see how large n should be in some cases). (1) For most computers the maximum time for Warshall's algorithm will be about 6 times the maximum time of the algorithm in this paper if the algorithms are used to compute the transitive closure of each graph with n nodes. (2) The algorithm in this paper will be faster than Warshall's algorithm for a graph selected at random where each arc has been selected with probability $p(n)$ if $\lim_{n\to\infty} np(n) = \infty$. In particular, if $p(n)$ is a constant, the time for Warshall's algorithm will increase as n^3 whereas the time for the algorithm in this paper will increase as n^2. (3) Warshall's algorithm takes an amount of time that increases faster than n^2 to compute the transitive closure of graphs from a set with the nodes numbered at random for which the number of arcs in the transitive closure N_t increases faster than $(n/\ln n)^2$ provided that the number of pairs of nodes connected by a path of length k or less is at least $\alpha k\sqrt{N_t}$ for fixed α and for $k \le \sqrt{2N_t}$. Such graphs include rooted threes and undirected graphs. There are many such graphs, for example rooted trees and undirected graphs, where the algorithm in the paper can compute the transitive closure in time n^2.

5. Variations on the Algorithm

Many details of the algorithm have been selected to make it easier to understand. Others have been selected arbitrarily. Now 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 Next1 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 $\underline{M}$ 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 computing time for this work.

Table 1

Step (substep)	Number of times (max)
1	1
1 (loop)	n
2	m + 1
3	n
4	$\frac{1}{2}[3n^2+n-m^2+m]$
5	$\frac{1}{2}[3n^2+n-m^2-3m]$
6 + 9	n
6 (loop) + 9	n
7 + 9	n
7 (outer loop)	$\frac{1}{2}[n^2+n-m^2-m]$
7 (inter loop)	$\frac{1}{2}[n^2+n-m^2-m]$
8 + 9	n
10	1
10 (outer loop)	m + 1
10 (inter loop)	m^2+m
11	1
12 + 14	m + 1
13	m + 1
15	1
15 (outer loop)	m + 1
15 (inter loop)	$\frac{1}{2}[m^2+3m]$
15 (2^{nd} if-part)+14	m
16	1
17	1
18	m + 1
18 (loop)	$\frac{1}{2}[m^2+m]$
19	$\frac{1}{2}[m^2+m]$
20	$\min\{\frac{1}{6}[m^3-m^2], N_1 m + N_1\}$
20 [average]	$\min\{pn^3, \frac{1}{2}[n^2p^2-np^{-2}]\}$
20 [undirected]	0
21	$\frac{1}{2}[m^2-m]$
22	1
23	m + 1
24	m^2+m
25	n m + m
26	n^2
27	n m + 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, which are not part of cycles and which can not be removed without changing the transitive closure, N_1 - the number of arcs, 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 ≤ 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 ≤ 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		
Warshall	.2 if i ≤ 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 ≤ 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		

Table 2 (continued)

Method	Probability	Number of Nodes 10	20	30	40	50	60
Warshall	.1 if $i \le j$	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 $i \le j$	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 $i \le j$	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 $i \le j$	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 1/60ths of a second) required by each method to 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 $\sqrt{\frac{n}{n-1}}$ times the observed standard deviation where n is the number of graphs which were tested. Since the clock measured time in 1/60ths 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 $[\frac{n}{47}]^{-1} n$, where $[x]$ is the smallest integer greater than or equal to x.

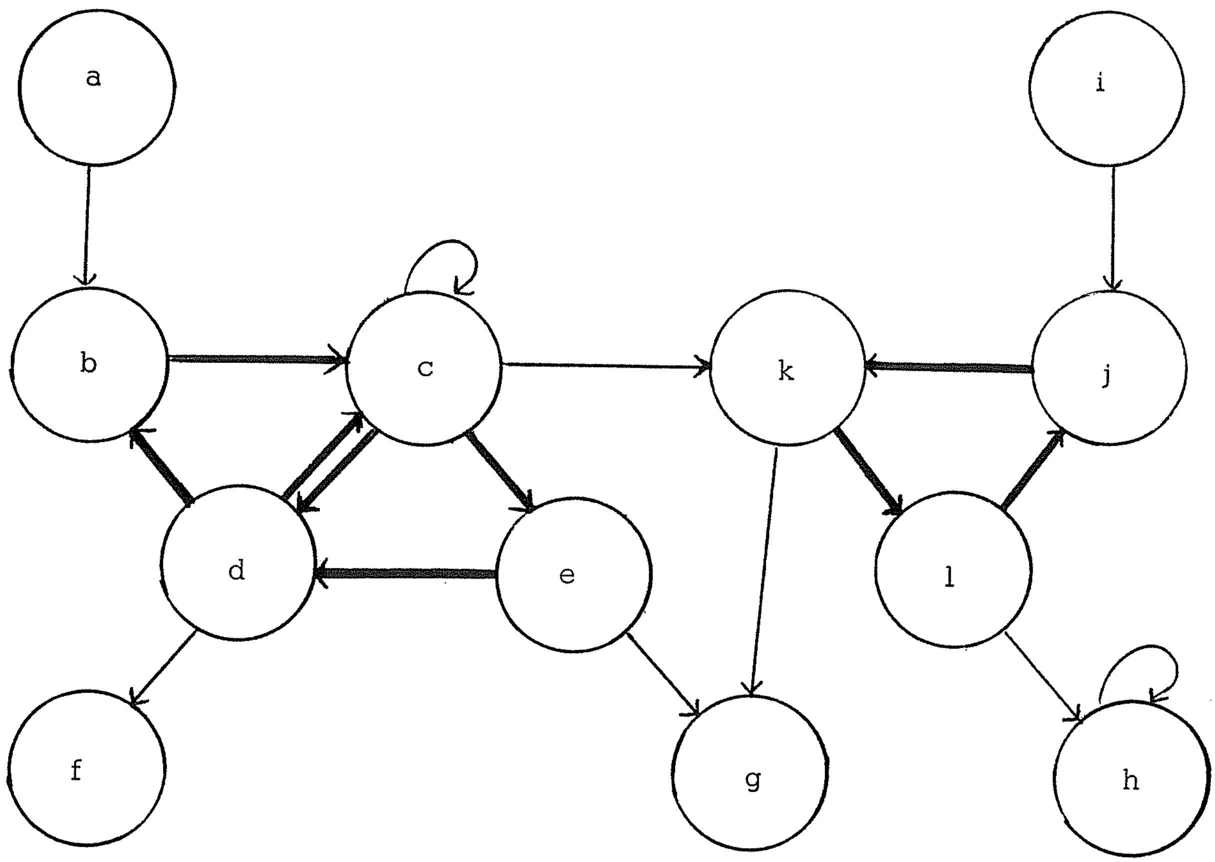

Figure 1. A directed graph with 12 nodes and 18 arcs. Arcs which connect pairs of nodes in the same path equivalence class are shown as dark arrows. Arcs which connect pairs of nodes in different equivalence classes and arcs which connect nodes to themselves are shown as light arrows.

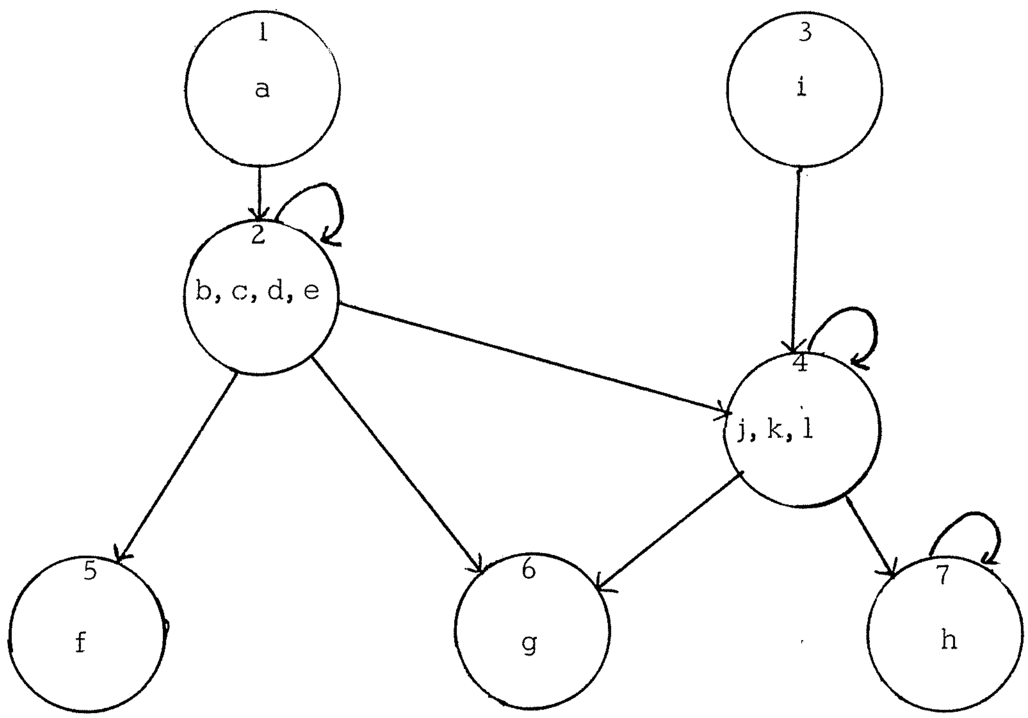

Figure 2. The graph from Figure 1 after the path equivalence classes have been replaced by single nodes. Part 1 of the algorithm combines those nodes which are members of the same path equivalence class into a single node. Thus nodes b, c, d, and e are now represented as a single node as are nodes j, 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.

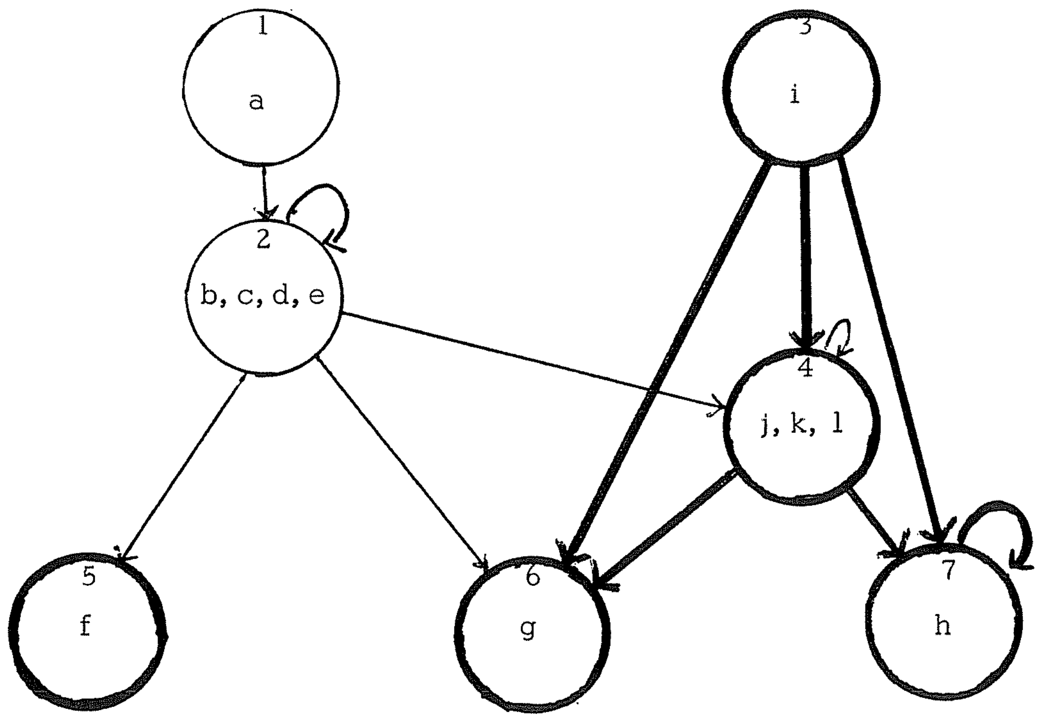

Figure 3. The graph being processed by part 3 of the algorithm. The dark nodes have been processed. The dark arcs form the transitive closure of the processed nodes. The algorithm is ready to compute the transitive closure for node 2 now that it has computed the transitive closure for all nodes after 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).

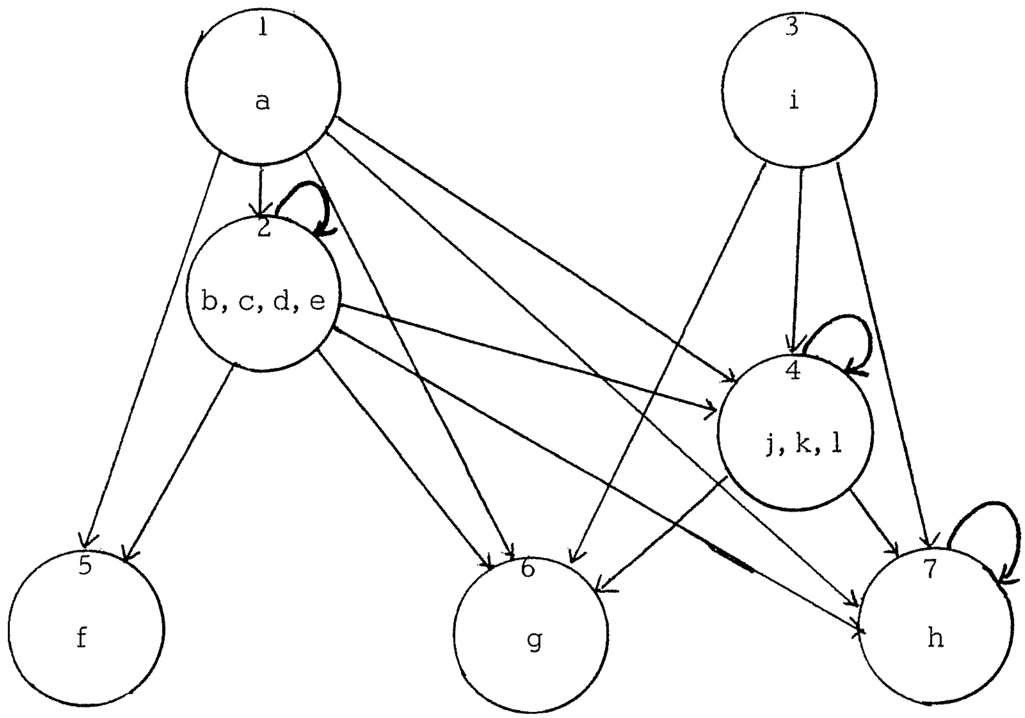

Figure 4. The transitive closure of the graph of path equivalence classes. Part 3 of the algorithm produces the transitive closure of the graph in which each equivalence class is represented by a single node. This transitive closure is used by part 4 of the algorithm to generate the transitive closure 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).

Appendix

```
procedure   TRANSCLOSURE (m, n); value n; integer n; boolean array m;
    comment Since the paper has extensive comments, the only comments
        given here are those to show the start of each step and part of the
        algorithm;
    begin
        integer array next [0 : n + 1]; previous [0 : n + 1], equivalent [1 : n];
        comment Part 1;
        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;
            for i := 1 step 1 until n do
            begin
                equivalent [i] := i ;
                next [i - 1] := i ;
                previous [i + 1] := i ;
                onstack [i] := 0 ;
                new [i] := true ;
            end ;
            next [ n] := 0;
            previous [1] := 0;
            previous [0] := n;
            top := 0;
            k := 0;
            comment Step 2;
```

```
starttree:
     k := next [k] ;
     if  k = 0  then go to order ;
     if ¬ new [k] then go to starttree ;
     i := k ;
     comment  Step 3 ;
stack i :
     top := top + 1 ;
     stack [top] := i ;
     onstack [i] := top ;
     j := 0 ;
     comment  Step 4 ;
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 onstack [j] ≠ 0  then go to removecycle;
     i := j ;
     go to stack i ;
     comment  Step 6 ;
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 ;
    a := 0 ;
    m [j, j] := true ;
combine1:
    a := next [a] ;
    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 ;
combine2:
    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 combine1;
    comment Step 8 ;
return:
    top := onstack [j] ;
    i := j ;
    j := 0 ;
    go to nextarc ;
```

```
        comment Step 9;
    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;
    comment Part 2;
order:
    begin
        integer array count [1 : n];
        integer j, i, k, a, b;
        comment Step 10;
        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;
    start:
        i := 0;
        k := n + 1;
        comment Step 12;
    advancej :
        j := i;
        comment Step 13;
```

```
checksuccessors ;
    i := previous [i] ;
    if i = 0 then go to startqueue ;
    if count [i] ≠ 0 then go to advance j ;
    comment Step 14 ;
    previous [k] := i ;
    previous [j] := previous [i] ;
    next [previous [i]] := j ;
    next [i] := k ;
    k := i ;
    i := j ;
    go to checksuccessors ;
    comment Step 15 ;
startqueue :
    i := n + 1 ;
    previous [k] := 0 ;
process 1 :
    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 process 2 ;
            comment  Step 16
        outorder :
            next [0]  :=  k ;
            previous [0]  :=  previous [n + 1] ;
            next [previous [n + 1]]  :=  0 ;
        end of Part 2 ;
        comment  Part 3 ;
    transitiveclosure:
        begin
            integer  k, i, j, oldj ;
            integer array next 1 [0 : n] ;
            comment  Step 17;
            k  :=  0 ;
            comment  Step 18;
        nextnode:
            k  :=  previous [k] ;
            if  k = 0  then go to output ;
            i  :=  k ;
            j  :=  k ;
        nextnode 1 :
            if  j = 0  then go to testarc ;
            next 1 [j]  :=  next [j] ;
            j  :=  next [j] ;
```

```
        go to nextnode 1 ;
        comment  Step 19 ;
    testarc:
        i  :=  next 1 [i] ;
        if  i = 0  then go to nextnode ;
        if ¬ m[k, i]  then go to testarc ;
        j  :=  i ;
        comment  Step 20 ;
    testclosure:
        oldj  :=  j ;
        j  :=  next 1 [j] ;
        if  j = 0  then go to testarc ;
        if ¬ m[i, j]  then go to testclosure ;
        comment  Step 21 ;
        m[k, j]  :=  true ;
        next 1 [oldj]  :=  next 1 [j] ;
        j  :=  oldj ;
        go to testclosure ;
    end of Part 3 ;
    comment  Part 4 ;
output:
    begin
        integer  i, j, a, b;
        comment  Step 22 ;
        i  :=  0 ;
        comment  Step 23 ;
    newi:
        i  :=  next [i] ;
        if  i = 0  then go to endalgorithm;
        j  :=  0 ;
        comment  Step 24 ;
```

```
        newj:
            j := next [j] ;
            if j = 0  then go to newi ;
            if ¬ m[i, j]  then go to newj ;
            a := i ;
            comment  Step 25 ;
        morea:
            a := equivalent [a] ;
            b := j ;
            comment  Step 26 ;
        newb:
            b := equivalent [b] ;
            m[a, b]  := true ;
            if b = j  then go to newa ;
            go to newb ;
            comment  Step 27 ;
        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. Palasti, I., "On the Strong Connectedness of Directed Random Graphs", Studia Scientiarum Mathematicarum Hungarica 1 (1966) 205-214.