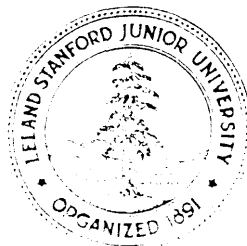ON COMPUTING THE TRANSITIVE CLOSURE OF A RELATION

by

James Eve

STAN-CS-75-508
SEPTEMBER   1975

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

On Computing the Transitive Closure of a Relation

James Eve

Abstract

An algorithm is presented for computing the transitive closure of
an arbitrary relation which is based upon a variant of Tarjan's algorithm
[4] for finding the strongly connected components of a directed graph.
This variant leads to a more compact statement of Tarjan's algorithm.

If V is the number of vertices in the directed graph representing
the relation than the worst case behavior of the proposed algorithm
involves $O(V^3)$ operations.   In this respect it is inferior to existing
algorithms [1,2] which require $O(V^3 / \log V)$ and $O(V^{\log_2 7} \log V)$
operations respectively.   The best case behavior involves only $O(V^2)$
operations.

1. Introduction.

The origin of this transitive closure algorithm is in a paper by Knuth [3] in which he defined certain sets, of interest in parsing context free languages, in terms of the transitive closure of relations. One class of sets and the method suggested by Knuth for computing it are germane and will be reviewed briefly.

Let

$$X \rightarrow X_1 X_2 \ldots X_2 Y\alpha$$

denote a production of some context free grammar in which the nonterminals, X_i , $1 \leq i \leq n$, derive the null string. Then X is said to <u>left depend</u> upon Y , denoted $X \, \ell \, Y$. The class of sets to be computed is

$$\text{first}(A) = \{a \in V_T \mid A \, \ell^+ \, a\} \quad \text{for all } A \in V_N$$

where V_N and V_T as usual denote the non-terminal and terminal symbols of the grammar, + and * appended to a relation denote respectively the transitive and reflexive transitive closures of the relation.

A necessary condition for top down deterministic parsing is that left recursion may not occur so that $X \, \ell^+ \, X$ is precluded for all X in V_N . This implies that the directed graph representing the left dependency relation has no closed paths and, as a consequence, a convenient partial ordering of the vertices of the graph exists which can be exploited in computing the required sets.

If A left depends upon $A_1, \ldots, A_m$ then the latter are immediate descendents of A in the left dependency graph and from the definition in terms of left dependency

$$\text{first}(A) = \bigcup_{i=1}^{m} \text{first}(A_i) \quad .$$

Given a partial ordering of the vertices in which descendents $A_1, \ldots, A_m$ always precede the ancestor, this identity may be used to compute the class of sets during a single left to right scan through the sequence of vertices in the partial ordering. A recursive procedure

2

FIRST(V) can be constructed which traverses in postorder a spanning tree, rooted at vertex V , of that subgraph of the directed graph accessible from V . Such a procedure would visit vertices in the subgraph in an appropriate sequence and as each vertex corresponding to a nonterminal symbol is visited the corresponding set can be computed. It merely remains to ensure that each subgraph is treated. The process is efficient in the sense that each vertex of the graph is visited once only and each arc is inspected once only.

If left recursion was not forbidden then $A \ell^+ B$ and $B \ell^+ A$ could occur implying $A \ell^+ A$ and, thereby, closed paths in the left dependency graph; however $A \ell^+ B$ and $B \ell^+ A$ imply that first(A) = first(B) which reflects the equivalence relation induced on the vertices of the graph by strong connectivity. If all vertices in such an equivalence class are mapped-onto a single representative vertex then the resulting directed graph is free of closed paths and can again be explored by Knuth's efficient "topological sorting" algorithm.

The discussion above need not be restricted to the left dependency relation, similar comments may be made with respect to any relation. Algorithms for computing the transitive closure of relations based upon these observations have been used by the author for several years; in many cases the advantages accruing from inspecting each arc and vertex once while computing the transitive closure offset inefficiencies in the somewhat crude methods used to locate multiple vertex strongly connected components. Tarjan's elegant algorithm for finding the strongly connected components of a graph makes this approach a great deal more attractive. For graphs with close to V or V^2 arcs the computation involves $O(V^2)$ operations. Worst case behavior involving $O(V^3)$ operations would arise for the graph with $\frac{1}{2} V(V-1)$ arcs but with no closed paths.

The following section describes a variant of Tarjan's algorithm and outlines a tedious proof of its correctness. Section 3 deals with its use in computing the transitive closure of a relation.

2.   An Alternative Formulation of Tarjan's Algorithm.

     Tarjan's algorithm involves traversing a spanning tree (or forest)
of a directed graph, accumulating the vertices visited on a stack, and
periodically it emits sets of vertices corresponding to strongly
connected components from the stack. The traversal is postorder resulting
in the strongly connected components being emitted in the desired
sequence, i.e., if strongly connected components A and B are connected
by one or more arcs from A to B then B will be emitted before A .
Thus Tarjan's algorithm includes the required topological sort. It does
not however maintain a list of the strongly connected components
defining the ordered sequence.

     Assuming that such a list is required for a graph of n vertices
then, since a vertex is never simultaneously on the stack and in the
list, both the stack and the list can be represented in an n element
array. This conceptually undesirable mixing of data structures permits
an extremely convenient encoding of necessary status information. It
will suffice in the subsequent application if each strongly connected
component is represented in the list of strongly connected components
by a single vertex; however, for each vertex of the directed graph it
must remain possible to locate its representative in the list of strongly
connected components.

     During execution of the algorithm each vertex passes from one to
the next of three states.

State 1.   The vertex has not yet been placed on the stack.

State 2.   The vertex is present on the stack as part of some as yet
            incompletely determined strongly connected component.

State 3.   The vertex has been removed from the stack and is represented
            by a vertex in the list of strongly connected components.

     As part of the process of finding vertices belonging to the same
strongly connected component, that vertex of the component which arrived
on the stack first is eventually identified and is used to represent all
vertices of the component in the list. This identification is achieved
by maintaining an index for each-vertex in state 2;   INDEX(v) will

will designate some vertex on the stack which belongs to the same strongly connected component as v and which arrived on the stack no later than v . By the time that all vertices of a strongly connected component have been visited only that vertex of the component which arrived on the stack first will have an index value which designates itself on the stack.

Clearly all vertices in State 2 will have index values designating vertices in the stack; when vertices are removed from the stack the corresponding index values can be changed to indicate the position of their representative in the list. By adopting the convention that INDEX(v) = 0 if v is a State 1 vertex, the array INDEX encodes all state information and eventually defines the mapping of vertices onto representatives in the list of strongly connected components.

For each vertex v of the directed graph a set, SONS(v) , is assumed to exist such that $w \in$ SONS(v) precisely when the directed graph contains an arc from v to w . The topological sorting of the strongly connected components of directed graphs specified in this way is achieved by the procedure TOPOLOGICALSORT in Figure 1. Elements n,n-1,... of the array VERTICES represent the stack; elements 1,2,... are used to build the list of strongly connected components.

To facilitate proof that strongly connected components are correctly identified by this procedure, the following notational conveniences are adopted.

-1.   x → y denotes the existence of an arc from vertex x to vertex y .

2.   x < y denotes that vertex x <u>immediately</u> precedes y on the stack.

3. INDEX(y) used as an operand with < is to be interpreted as a
    reference to the vertex on the stack designated by INDEX(y) ,
       i.e., to VERTICES(INDEX(y)).Likewise INDEX(y) ≠ y is understood
       to mean that INDEX(y) does not designate itself on the stack.

4. $y \in$ scc(x) denotes that y belongs to the same strongly connected
       component as x .

```
procedure TOPOLOGICALSORT;
    begin integer stackindex, listindex, i;
    integer array VERTICES, INDEX(1::n);
    procedure ORDERVERTICES(integer value x);
        begin integer w;
        INDEX(x) ← stackindex ← stackindex -1; VERTICES(stackindex) ← x;
        for w ∈ SONS(x) do
            begin if INDEX(w) = 0 then ORDERVERTICES(w);
            if INDEX(w) > INDEX(x) then INDEX(x) ← INDEX(w);
            comment  if the preceding test is satisfied then w is on the
            stack and INDEX(w) designates a vertex which arrived on the
            stack earlier than that indicated by INDEX(x). As this earlier
            vertex belongs to the same strongly connected component as x,
            INDEX(x) is updated to designate this earlier vertex;
            end;     .
        comment the implicit assignment TREATED(x)←true occurs here;
        if x = VERTICES(INDEX(x)) then
            begin x is the vertex of a strongly connected component which
            arrived on the stack first. Pop the vertices of the component
            from the stack updating the index of each to indicate the next
            list position until x itself is popped then insert x in the next
            list position;
            listindex ← listindex + 1;
            repeat w ← VERTICES(stackindex); INDEX(w) ← listindex;
                stackindex ← stackindex + 1;
                until w = x;
            VERTICES(listindex) ← x
            end;
    . end;

listindex ← 0; stackindex ← n+1; for i ← 1 until n do INDEX(i) ← 0;
for i ← 1 until n do if INDEX(i) = 0 then ORDERVERTICES ( i) ;
end:
```

Figure 1

A major obstacle to understanding this algorithm' and indeed Tarjan's original version appears to be that certain variables recording the current status of the computation are never referenced and so do not appear explicitly in encodings of the algorithm. For purposes of proof it is convenient to insert them. Conceptually there exists within the procedure TOPOLOGICALSORT a Boolean array TREATED; TREATED(x) is assumed to be initialized to false for each vertex x . An implicit assignment

TREATED(x) ← true

immediately follows the for statement of the procedure ORDERVERTICES in Figure 1.

The proof that strongly connected components are correctly identified depends upon the following properties of the procedure ORDERVERTICES in Figure 1.

1. For any vertex x , immediately after TREATED(x) is assigned the value true , if INDEX(x) designates x then x will be removed from the stack at that time.

2. As a consequence of the postorder traversal any true descendent y (in the spanning tree) of a vertex x on the stack will arrive on the stack after x ; when TREATED(x) is assigned the value true then TREATED(y) = true . A true ancestor y (in the spanning tree) of a vertex x on the stack will precede x on the stack; when TREATED(x) is assigned the value true , TREATED(y) will be false .

3. For each vertex x , when x is placed on the stack, INDEX(x) is initialized to designate x itself; subsequent assignments to 'INDEX(x) preserve INDEX(x) $\overset{+}{<}$ x .

Lemma 1. When TREATED(x) is assigned the value true , if INDEX(x) designates vertex z then for all y for which $z \overset{*}{<} y$,

(i) $y \in scc(x)$,

(ii) if TREATED(y) = true then there exists no arc $y \to r$ in the directed graph such that $r \overset{+}{<} z$ and $x \overset{*}{<} y$.

<u>Proof.</u>   Assertion (i) is proved by induction on the hypothesis that it
holds for all vertices v for which TREATED(v) = t̰r̰ṵḛ .

    Consider the processing of vertex x . The conditional call on
ORDERVERTICES ensured that any son of" x is either on the stack or
represented in the list of strongly connected components prior to its
inspection with a view to updating INDEX(x) .   Since TREATED(x) is
about to be assigned the **value** true , it must be shown that INDEX(x)
is left with a value consistent with the assertions.

<u>Case 1</u>.   No assignment changing the initial value of INDEX(x) is made.
This case arises if x has no sons or only sons which are either
represented in the list of strongly connected components or sons such
as w , which are on the stack, but for which $INDEX(x) \overset{*}{<} INDEX(w)$.

    If x is the last vertex on the stack then the assertions are
vacuous.

    If x is not the last vertex on the stack then there exists
. $w_1 \in SONS(x)$ for which

(a) $x < w_1$,

(b) $INDEX(w_1) \overset{+}{<} w_1$, since w1 has not been removed from the stack,

(c) $TREATED(w_1) = t̰r̰ṵḛ$ so that the inductive hypothesis applies to w_1 .

    (a) and (b) imply that $INDEX(w_1) \overset{*}{<} x$, but since no assignment to
INDEX(x) occurs, $INDEX(w_1)$ must designate x . Consequently the
inductive hypothesis applied to wl establishes that, for all y
satisfying $x \overset{+}{<} y$, $y \in scc(x)$; assertion (i) then is true in this
case.

    The following cases cover the three distinct situations in which a
new value,   INDEX(w) , is assigned to INDEX(x) after its initialization;
such assignments are conditional upon $INDEX(w) \overset{+}{<} INDEX(x)$ where
$w \in SONS(x)$.   In that this new value is potentially the value z in the
assertions of the lemma, it will be shown that in each case

(1) $x \in scc(t)$ where t is the vertex designated by INDEX(w) ,

(2)   any vertex y on the stack which follows t and precedes x
      which is not covered by the inductive hypothesis belongs to the
      same strongly connected component as t .   TREATED(y) is true
      for $x < y$ so the inductive hypothesis applies to such y .

      (1) and (2) suffice to complete the proof of assertion (i) of the
lemma.

<u>Case 2.</u> $w \in SONS(x)$ and $x \overset{+}{<} w$.

      Since only descendents of x in the spanning tree follow x on
the stack,   TREATED(w) is <u>true</u> and the inductive hypothesis applied
to w together with INDEX(w) $\overset{+}{<}$ INDEX(x) $\overset{*}{<}$ $x \overset{+}{<} w$ establish that
$x \in scc(t)$. The inductive hypothesis for w suffices to demonstrate
that any y satisfying $t < y$ and $y < x$ also satisfies $y \in scc(t)$.

<u>Case 3.</u> $w \in SONS(x)$ where $w \overset{+}{<} x$ and INDEX(w) $\neq w$.

      INDEX(w) $\neq w$ implies that TREATED(w) = true . For x to follow w
.in the stack when TREATED(w) is true there must exist in the spanning
tree some vertex v with distinct sons u_1 and u_2 , and paths such that

(a) $v \rightarrow u_1 \overset{*}{\rightarrow} w$, where TREATED(s)= true for any vertex s in the
     path $u_1 \overset{*}{\rightarrow} w$,

(b) $v \rightarrow u_2 \overset{*}{\rightarrow} x$, where TREATED(s) = false for any vertex s in the
     path $u_2 \overset{*}{\rightarrow} x$.

      The inductive hypothesis applies to u_1 ; INDEX(u_1) $\overset{*}{<}$ v since u_1
remains on the stack and so $w \in scc(v)$ implying that $w \overset{*}{\rightarrow} v$ is a path
in the directed graph. Combining this with (b) above and the fact that
$w \& Ok(x)$, we have

(c) $w \overset{*}{\rightarrow} v \rightarrow u_2 \overset{*}{\rightarrow} x \rightarrow w$

so that $x \in scc(w)$. But the inductive hypothesis applies to w so
$w \in scc(t)$. Hence $x \in scc(t)$.

      Only vertices in the path $u_2 \rightarrow x$ can satisfy both of the conditions,
that they precede x on the stack and were not present on the stack when
TREATED(w) was assigned the value true . These vertices are therefore

9

not known to be members of the same strongly connected component as t by virtue of the inductive hypothesis applied to w ; they clearly do belong to $scc(t)$ by virtue of (c) and $w \in scc(t)$.

Case 4. $w \in SONS(x)$ where $w \overset{+}{<} x$ and $INDEX(w) = w$. w must be an ancestor of x in the spanning tree.   ($w \overset{+}{<} x$ precludes it being a descendent and as seen in Case 2, $INDEX(w) = w$ rules out the only other possibility.)

Thus $w \overset{+}{\to} x$ and $w \in SONS(x)$ implies $x \in scc(w)$.   In this case $w = t$ so $x \in scc(t)$.   As a consequence of the postorder traversal any vertices on the stack between t and x lie on the path $t \overset{+}{\to} x$ and so also belong to $scc(t)$.

Assertion (ii) follows from observing that if any vertex y has several sons w_i ; $1 \leq i \leq m$, when $TREATED(y)$ becomes true , $INDEX(y) \overset{*}{<} INDEX(w_i)$ for $1 < i < m$.

Assume that an arc $y \to r$ exists such that $r \overset{+}{<} z$ and $x \overset{*}{<} y$. When $TREATED(y)$ becomes true since $r \in SONS(y)$ then $INDEX(y) \overset{*}{<} r$.

Now $x \overset{*}{<} y$ implies

either

> (1) that $x = y$ in which case $INDEX(x) \overset{*}{<} r$, but then when
> $TREATED(x)$ is assigned true since $INDEX(x)$ designates z
> we have $z \overset{*}{<} r$ -- a contradiction,

or

> (2) x is an ancestor of y in the spanning tree, i.e.,
> $x = u_1 \to u_2 \to \cdots \to u_n = y$ is a path in the spanning tree.
> Since by the same observation, for $1 \leq j \leq n-1$,
> $INDEX(u_j) \overset{*}{<} INDEX(u_{j+1})$ when $TREATED(u_j)$ is assigned true ;
> it is clear that the same contradiction will arise.   □

Theorem 1.   The procedure ORDERVERTICES correctly identifies strongly connected components of an arbitrary directed graph.

Proof.   If $INDEX(x)$ designates x when $TREATED(x)$ is assigned true then the theorem follows from Lemma 1 since $TREATED(y) = $ true for all y satisfying $x \overset{*}{<} y$. □

10

3. Computing the Transitive Closure of a Relation.

Let R be a given relation and RPLUS be a two dimensional array
initialized so that all elements have the value false ; RPLUS(i,j) is to
be assigned the value true if and only if $i R^+ j$. It is assumed
that R is specified by sets

SONS(i) = (j | i R j} .

If ORDERVERTICES in Figure 1 is replaced by the procedure CLOSURE
Figure 2 then invoking TOPOLOGICALSORT will achieve this objective.
CLOSURE is merely an elaboration of ORDERVERTICES. While processing
$w \in$ SONS(x) , RPLUS(x,w) is assigned true . Subsequently two
possibilities arise,

either

(1) w is on the stack and so WE scc(x) in which case no
assignment is made to RPLUS at this time

or

(2) w is represented in the list of strongly connected components
and the assignment
    RPLUS(x,*) ← RPLUS(x,*) or RPLUS(w,*)
is necessary. RPLUS(z,*) denotes row z of RPLUS; for
convenience availability of bitwise Boolean operations on
Boolean vectors is assumed.

Finally, whenever a vertex w , in the same strongly connected component
as the representative vertex x , is removed from the stack the assignment

RPLUS(x,*) ← RPLUS(x,*) or RPLUS(w,*)

is executed.

11

```
procedure CLOSURE(integer value x);
   begin integer w;
   INDEX(x) ← stackindex ← stackindex -1; VERTICES(stackindex) ← x;
   for w ∈ SONS(x) do-
       =  RPLUS(x,w)  +  true; if INDEX(w) = 0 then CLOSURE(w);
       if INDEX(w) < stackindex then RPLUS(x,*) ← RPLUS(x,*) or RPLUS(w,*)
           else if INDEX(w) > INDEX(x) then INDEX(x) ← INDEX(w);
       end;
   if x = VERTICES(INDEX(x)) then
       begin listindex ← listindex + 1;
       repeat w + VERTICES(stackindex); INDEX(w) ← listindex;
           RPLUS(x,*) ← RPLUS(x,*) or RPLUS(w,*);
           stackindex ← stackindex + 1;
           until w = x;
       VERTICES(listindex) ← x;
       end;
   end;
```

Figure 2

Theorem 2.   If $x \overset{+}{\to} y$ is a path in the directed graph representing
relation R then the procedure CLOSURE leaves RPLUS(x,y) = true for
any x in the list of strongly connected components.

Proof.   For any vertex v on the stack maintained by the procedure
CLOSURE, when all members of SONS(v) have been examined then by
construction RPLUS(v,t) = true   iff

    either   (1) t ∈ SONS(v)
    or   (2) w ∈ SONS(v) and RPLUS(w,t) = true and $w \notin scc(v)$.

Since a ∈ SONS(b) implies b R a , apart from the constraint that $w \notin scc(v)$
this is simply the usual recursive definition of R^{+} .   It follows from

12

this and the work of Knuth cited that if $v \to u_1 \to \ldots \to u_n$ is a path in the directed graph corresponding to R , in which, for $1 < j < n$, $u_j \notin \text{scc}(v)$ then $\text{RPLUS}(v,u_j)$ is true .

If $v_k \in \text{scc}(x)$ for $1 \leq k \leq m$ then by construction

$$\text{RPLUS}(x,*) = \bigvee_{k=1}^{m} \text{RPLUS}(v_k,*)$$

from which it follows that if $x \xrightarrow{+} y$ is an arbitrary path in the directed graph corresponding to R then $\text{RPLUS}(x,y) = \text{true}$. $\square$

In computing the transitive closure of R , each arc and each vertex of the directed graph corresponding to R are visited once; for each strongly connected component of k vertices, k-1 Boolean vector operations are performed each involving V elements. One Boolean vector operation is needed for each arc connecting distinct strongly connected components. One Boolean assignment is made for each arc.

If there are t strongly connected components each with more than one vertex and these account for V_s of the total of V vertices; if in addition E_s of the total of E arcs connect vertices in the set V_s then the number of operations needed is bounded by

$$k_1(E - E_s)V + k_2 E + k_3(V_s - t)V + k_4 V + k_5$$

for some constants k_i , $1 \leq i \leq 5$.

Worst case behavior occurs when the vertices can be placed in a -sequence in which each member is connected by an arc to its successors and itself, then $E_s = V_s = t = 0$ and $E = \frac{1}{2} V(V+1)$. Best case behavior occurs when E = V-1 and $E_s = V_s = t = 0$ or when $E = E_s = V^2$, $V_s = V$ and t = 1 . The number of operations needed therefore ranges between $O(V^2)$ and $O(V^3)$.

This algorithm has brevity as one recommendation (compared with the algorithms in [1,2] which improve on the worst case behavior). It offers occasional conveniences when compared to methods involving incidence matrix representations of R . For example, Knuth's sets first(A) are relations on $V_N \times V_T$ rather than $(V_N \cup V_T) \times (V_N \cup V_T)$. It is easy to

take advantage of this with the present algorithm. Many computers offer Boolean vector operations for moderately large n as a primitive operation, thus in many practical situations the effective behavior is more like $O(V)$ to $O(V^2)$ operations.

References

[1] V. L. Artazarov, E. A. Dinic, M. A. Kronrod and I. A. Faradzev, "On economical construction of the transitive closure of a directed graph," Soviet Math. Dokl. 11 (1970), 1209.

[2] M. E. Furman, "Application of a method of fast multiplication of matrices in-the problem of finding the transitive closure of a graph," Soviet Math. Dokl. 11 (1970), 1252.

[3] D. E. Knuth, "Top-Down Syntax Analysis," Acta Informatica 1 (1971), 79.

[4] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Corn-put. 1 (1972), 146.