Efficient Transitive Closure Algorithms

Yannis E. Ioannidis †
Raghu Ramakrishnan

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

We have developed some efficient algorithms for computing the transitive closure of a directed graph. This paper presents the algorithms for the problem of reachability. The algorithms, however, can be adapted to deal with path computations and a significantly broader class of queries based on one-sided recursions. We analyze these algorithms and compare them to algorithms in the literature. The results indicate that these algorithms, in addition to their ability to deal with queries that are generalizations of transitive closure, also perform very efficiently, in particular, in the context of a disk-based database environment.

1. Introduction

Several transitive closure algorithms have been presented in the literature. These include the Warshall and Warren algorithms, which use a bit-matrix representation of the graph, the Schmitz algorithm, which uses Tarjan's algorithm to identify strongly connected components in reverse topological order, and the Seminaive and Smart/Logarithmic algorithms, which view the graph as a binary relation and compute the transitive closure by a series of relational joins.

While all of the above algorithms can compute transitive closure, not all can be used to solve some related problems. Schmitz's algorithm cannot be used to answer queries about the set of paths in the transitive closure (e.g. to find the shortest paths between pairs of nodes) since it loses path information by merging all nodes in a strongly connected component. Only the Seminaive algorithm computes selection queries efficiently. Thus, if we wish to find all nodes reachable from a given node, or to find the longest path from a given node, with the exception of the Seminaive algorithm, we must essentially compute the entire transitive closure (or find the longest path from every node in the graph) first and then perform a selection.

We present new algorithms based on depth-first search and a scheme of marking nodes (to record earlier computation implicitly) that computes transitive closure efficiently. They can also be adapted to deal with selection queries and path computations efficiently. In particular, in the context of databases an important consideration is I/O cost, since it is expected that relations will not fit in main memory. A recent study [Agrawal and Jagadish 87] has emphasized the significant cost of I/O for duplicate elimination. The algorithms presented here will incur no I/O costs for duplicate elimination, and we therefore expect that they will be particularly suited to database applications. (We present an analysis of the algorithms that reinforces this point.)

The paper is organized as follows. We introduce some notation in Section 2. Section 3 presents the algorithms, starting with some simple versions and subsequently refining them. We present an analysis of these algorithms in Section 4 and discuss selection queries in Section 5. Section 6 contains a comparison of the algorithms to related work. We briefly discuss these algorithms in the context of path computations, one-sided recursion, and parallel execution in Section 7. Finally, our conclusions are presented in Section 8.

2. Notation and Basic Definitions

We assume that the graph G is specified as follows: For each node i in the graph, there is a set of successors $E_i = \{ j \mid (i, j) \text{ is an arc of } G \}$.

† Partially supported by the National Science Foundation under Grant IRI-8703592.

We denote the transitive closure of a graph G by G^*. The *strongly connected component* of node i is defined as $V_i = \{ i \} \cup \{ j \mid (i, j) \in G^* \text{ and } (j, i) \in G^* \}$. The component V_i is *nontrivial* if $V_i \neq \{ i \}$. The *condensation graph* of G has the strongly connected components of G as its nodes. There is an arc from V_i to V_j in the condensation graph if and only if there is a path from i to j in G.

The algorithms we present construct a set of successors in the transitive closure for each node in G. The set of successors in the transitive closure for a node i is $S_i = \{ j \mid (i, j)$ is an arc of $G^* \}$. A successor set S_i is partitioned into two sets M_i and T_i, and these may be thought of as the "marked" and "tagged" subsets of S_i. Initially $M_i = T_i = \phi$ for all i. †

3. The Transitive Closure Algorithms

3.1. A Marking Algorithm

In this section, we present a simple version of the algorithm. We do not suggest using this algorithm in general; we present better algorithms, which are derived by refining this algorithm. (In this algorithm alone, S_i is partitioned into two sets M_i and U_i - not T_i - that can be thought of as "marked" and "unmarked".)

```
proc Basic_TC ( G )
Input: A digraph G with successor sets E_i, i = 1 to n.
Output: S_i = U_i ∪ M_i, i = 1 to n, denoting G*.
{ U_i := E_i; M_i := φ;
 for i = 1 to n do
     while there is a node j ∈ U_i
       do M_i := M_i ∪ M_j ∪ { j }; U_i := U_i ∪ U_j - M_i od
 od
}
```

Proposition 3.1: Algorithm *Basic_TC* correctly computes the transitive closure of a directed graph G.

3.2. A Depth-First Transitive Closure Algorithm

Suppose that the graph G is acyclic. Let us number the nodes using a depth-first search such that all descendants of a node numbered n have a lower number than n. If we now run algorithm *Basic_TC* using this ordering, every time we add a successor set S_j to a set S_i, $S_j = M_j$, and $U_j = \phi$. We refer to such additions as *closed additions*. (The successor set S_j is closed, that is, it contains all successors of j in the transitive closure.)

To deal correctly with cycles, we must make some modifications. The idea is to ignore back arcs (arcs into nodes previously visited by the depth-first search procedure) during the numbering phase. The algorithm *Basic_TC* is run using this numbering. In the presence of cycles, not all additions are closed. (For example, the addition of S_j to S_i when the arc (i, j) is a back arc is not a closed addition.)

While numbering the nodes in a preprocessing phase for algorithm *Basic_TC* exposes the underlying ideas clearly, we might improve performance by doing the transitive closure work as we proceed in the numbering algorithm. The following simple algorithm illustrates the idea, although it only works for dags.

```
proc Dag_DFTC ( G )
Input: A graph G represented by successor sets E_i.
Output: S_i, i = 1 to n, denoting G*.

{ for i = 1 to n do visited[i] := 0; S_i := φ od
 while there is some node i s.t. visited[i] = 0 do visit(i) od
}

proc visit ( i )
{ visited[i] := 1;
 while there is some j ∈ E_i - S_i do
     if visited[i] = 0 then visit(j);
     S_i := S_i ∪ S_j ∪ { j }
 od
}
```

The above algorithm can be modified to deal with cyclic graphs as follows. We need to distinguish nodes that are reached via back arcs, and we now partition S_i into two subsets M_i, and T_i. T_i denotes nodes reached via back arcs.

```
proc DFTC ( G )
Input: A graph G represented by successor sets E_i.
Output: S_i = M_i ∪ T_i, i = 1 to n, denoting G*.
{ /* visited[i]    1 if visit 1(i) has been called.          */
  /* visited2[i]   1 if visit 2(i) has been called.          */
  /* popped[i]     1 if call to visit 1(i) has returned.     */
  /* Global        The succ. set of the root of a str. comp. */
  /*               Initialized before calling visit 2 for root. */

 for i = 1 to n do
     visited[i] := visited2[i] := popped[i] := 0;
     M_i := T_i := Global := φ
```

† The set S_i may be thought of as containing elements that are either marked or tagged. Agrawal and Jagadish [Agrawal and Jagadish 88] pointed out that this would lead to $O(n^2)$ storage overhead for the marks and tags. They observed that implementing this by partitioning S_i into separate sets incurs almost no additional overhead. (The space for storing the graph is $O(n^2)$ in any case.)

```
 od
 while there is some node i s.t. visited[i] = 0 do visit1(i) od
}

proc visit1 ( i )
{ visited[i] := 1;
 while there is j ∈ E_i - M_i - T_i do
   if visited[j] = 0 then visit1(j);
   if popped[j] > 0  /* (i, j) is not a back arc. */
      then { M_i := M_i ∪ M_j ∪ { j }; T_i := (T_i ∪ T_j) - M_i }
      else T_i := T_i ∪ { j }
 od
 if i ∈ T_i then /* i belongs to a strong component. */
   if T_i = { i } /* i is the root. */
      then {M_i := M_i ∪ { i }; T_i = ϕ;
                Global := M_i; visit2(i) }
      else { T_i := T_i - { i }; M_i := M_i ∪ { i } }
 popped[i] := 1
}

/* Assigns Global to succ. sets of all nodes in str. comp. */

proc visit2( i )
{ visited2[i] := 1;
 while there is j ∈ E_i s.t. visited2[j]=0 and T_j ≠ ϕ      do
visit2(j) od
 M_i := Global; T_i := ϕ
}
```

Theorem 3.2: Algorithm *DFTC* correctly computes the transitive closure of G.

Notice that *visit2* is called immediately after a strong connected component is identified and fully updates the successor lists of all nodes in the component. An alternative would be to make the calls to *visit2* after *visit1* is called for all the nodes in the graph. This second alternative has strictly inferior performance to *DFTC*, because nodes in a strong connected component might be visited from nodes outside the component while still having their successor lists incomplete. A variant of this alternative was suggested by Agrawal and Jagadish [Agrawal and Jagadish 88].

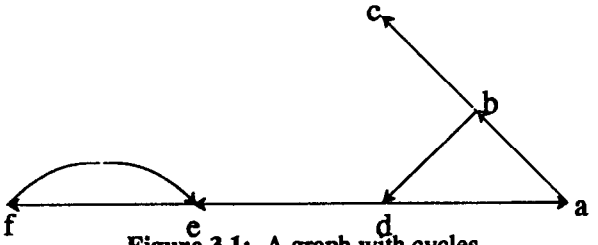

Figure 3.1: A graph with cycles.

The example of Figure 3.1 illustrates the basic difference between the *DFTC* algorithm and *Dag_DFTC*, which is the need for a second phase to take care of nontrivial strongly connected components. The graph of Figure 3.1 contains two such components, namely $\{a,b,d\}$ and $\{e,f\}$. Concentrating on the former, when a is visited via the arc (d,a), the successor list of a has not been computed yet, and so the successor set of d cannot be updated properly. *Basic_TC* solves the problem by continuing to visit the successors of a again, but this may lead to serious inefficiencies. *DFTC* solves the problem in the second pass, where a is identified as the root of the component and its successor list is distributed to all the nodes in the component. This is done by the calls to *visit2*. Similar comments hold for the other component also.

3.3. Optimized Processing of Nontrivial Strong Components

There is a major potential inefficiency in *DFTC* in that the second pass over a strong component reinfers several arcs of the transitive closure that have been inferred during the first pass also. This can be seen in the component $\{a,b,d\}$ of Figure 3.1. Assume that from d we first visit e and f and then visit a. As soon as (d,a) is discovered as a back arc, *DFTC* puts a in the tagged set of d, and it then pops back to b adding into its successor list d,e,f, and a, with a being tagged. Finally, the successor list of a is updated to contain all the nodes in the graph without any tags. During the second phase of going over the component $\{a,b,d\}$, nodes d,e,f, and a (as well as c) will be reinferred as successors of b. In this section we develop an algorithm that avoids this duplication of effort by essentially generating the successors of only one of the nodes in a nontrivial strong component during the first pass. In the second pass, the lists of all the other nodes are updated, thus avoiding any unnecessary duplication of work.

In this version of the algorithm, we do not need to distinguish tagged elements by partitioning successor lists, since a stack mechanism that is used to construct the successor set for (the root of) a strongly connected component allows us to make this distinction. During the process of the algorithm, the elements of the stack are lists of successors of nodes in some nontrivial strongly connected component. If we discover that some of these (potentially distinct) "components" are in fact part of the same component, then elements of the stack are merged to reflect this. † The array *visited* contains integer elements in this algorithm. The nota-

† There are other transitive closure algorithms that use stacks (e.g., [Schmitz 83, Agrawal and Jagadish 88]). Our use of the stack, however, is unique in that it is a stack of successor lists of nodes in nontrivial strong components, as opposed to a stack of nodes.

tion $L_1 := L_1 \bullet L_2$ is used to indicate that list L_2 is concatenated to list L_1 by switching a pointer, at O(1) cost. For the special case when L_1 is ϕ (that is, when list L_2 is to be assigned to the empty list L_1) we use the notation $L_1 := \bullet L_2$. In contrast, the notation $L_1 := L_1 \cup L_2$ is used to denote that a copy of L_2 is inserted into L_1.

```
proc Global_DFTC ( G )
Input: A graph G represented by successor sets E_i.
Output: S_i, i = 1 to n, denoting G*.
{
 /* root[f]      root of the str. comp. of stack frame f.   */
 /* list[f]      successors of nodes in the str. comp. of   */
 /*              stack frame f.                             */
 /* nodes[f]     nodes in the str. comp. of stack frame f.  */
 /* ptr[i]       pointer to the stack frame of the strong   */
 /*              component where i belongs.                 */
 /* top          pointer to the top of the stack.           */
 /* visited[i]   order in which visit(i) is called.         */
 /* bot          temporary variable used when collapsing    */
 /*              multiple potential str. comp. into one.    */

 vis := 1; top := 0;
 for i := 1 to n do
     visited[i] := popped[i] := root[i] := 0;
     ptr[i] := n+1; list[i] := nodes[i] := S_i := nil
 od
 while there is some i s.t. visited[i]=0 do visit(i) od
}
proc visit ( i )
{ visited[i] := vis; vis := vis + 1;
 while there is j ∈ E_i-S_i-{ i } do
     if visited[j] = 0 then visit(j);
     if popped[j] > 0 and ptr[j] = n+1
                 /* i,j in different strong component. */
       then S_i := S_i ∪ S_j ∪ { j };
     if popped[j] > 0 and ptr[j] ≠ n+1
        /* i,j in same str. comp. but (i,j) not a back arc. */
       then {
          bot := min (top,ptr[i],ptr[j]);
          /* merge multiple potential str. comp. into one. */
          while top > bot do
            list[top-1] := list[top-1] • list[top];
            nodes[top-1] := nodes[top-1] • nodes[top];
            if visited[root[top]] < visited[root[top-1]]
               then root[top-1] := root[top];
            top := top - 1;
          od
          if ptr[i] = n+1 /* (i,j) is a back arc.          */
                          /* New stack frame is created.  */
               then list[top] := list[top] • S_i;
          ptr[i] := top; S_i := • list[top]
       };
     if popped[j] = 0
       then { top := top + 1; root[top] := j;
              list[top] := • S_i;
              nodes[top] := nil; ptr[i] := top }
 od
 if i = root[top] /* Propagate successors of root to the rest */
                  /* of the nodes in a strong component.     */
     then { for each j ∈ nodes[top] ∪ { i }
              do S_j := list[top] ∪ { i }; ptr[j] := n+1 od;
            top := top - 1 }
 elseif ptr[i] ≠ n+1 /* Insert i into the strong component */
                     /* where it belongs.                  */
     then { list[ptr[i]] := list[ptr[i]] ∪ { i };
            nodes[ptr[i]] := nodes[ptr[i]] ∪ { i } };
 popped[i] := 1
}
```

Theorem 3.3: Algorithm *Global_DFTC* correctly computes the transitive closure of G.

The key point in the algorithm is the following invariant: If $ptr[i] = n$, and $m = \min(n, top)$, then every node in the set $list[m] \cup nodes[m]$ is reachable from i, and node i is reachable from every node in the set $nodes[m]$. The node $root[m]$ is the earliest visited node which can be reached from some node in the set $nodes[top]$. This underlies the collapsing of multiple potential strong components into one.

Duplication of effort is avoided by distributing the work associated with a nontrivial strong component between the first and the second pass. In component $\{a,b,d\}$ of Figure 3.1, as successors are generated, they are put into the appropriate list of the global stack. When the root a has been processed, that list contains the successors of a, which have been generated once for every independent path of some node in the component. Nodes e and f may have been generated as successors of d originally, but when the algorithm recognizes that d belongs to a nontrivial strong component, these successors are moved to the appropriate list of the global stack in O(1) time (by list concatenation). Hence, all of these inferences can be attributed to a, so that when in the second phase we make the list of a list of b and d also, this effort has not been accounted before.

We want to illustrate two points about the operation of the global stack of lists. The first is concerned with separate strong components. In Figure 3.1, assume that (d,a) is traversed before (d,e). When (d,a) is traversed, an empty list is pushed on the stack. Later, when (f,e) is traversed and the second component is discovered, another empty list is pushed on the stack. When we pop up to e again, the list of the top of the stack contains e and f, the fact that the visit to the top strong component is completed is recognized,

and after the second pass, the top of the stack is removed. Thus, when we continue popping up from d, the lower strong component does not appear as such in the stack, and so no undesirable interference occurs.

The second point we want to illustrate is concerned with a single strong component which is discovered in a piecemeal fashion. Figure 3.2 will serve as the working example.

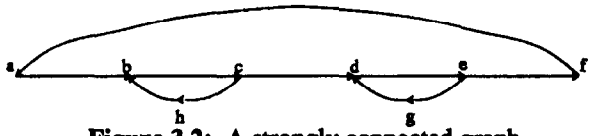

Figure 3.2: A strongly connected graph.

The whole graph is one strong component. Assume that the nodes are visited in the order a, b, c, h, d, e, g, and f. Thus the back arcs (h,b) and (g,d) are discovered before (f,a) is. This results in two potentially independent components to be pushed on the stack, namely, $\{b,c,h\}$ and $\{d,e,g\}$ After (f,a) is discovered, a third level is added to the stack, because there is no way of knowing that all of the nodes belong to the same component. This is discovered when we pop up back to e again, the second if-statement in the algorithm case is triggered, and the two lists at the top (corresponding to a and d respectively) are merged into one in O(1) time by simply changing some pointers. When c is reached, similar actions are taken, so that when a, the root, is reached, all its successors are correctly found in the top list.

4. Analysis of the Algorithms

We now present an analysis of the complexity of all the above algorithms. For each algorithm, we first analyze its time complexity assuming that everything fits in main memory. We then analyze its I/O complexity assuming that data has to be moved back and forth between main memory and disk. For the second case, the first analysis represents the expected CPU time. In addition, in Section 6, we will present an analysis of the Seminaive algorithm [Bancilhon 85], Warren's algorithm [Warren 75], and an algorithm by Schmitz [Schmitz 83], and we will compare their performance with that of our algorithms.

The forthcoming analysis assumes that all algorithms use the appropriate structures (combination of list representation and bit representation of a graph) so that duplicate elimination can be done in constant time. This can be achieved as follows: Whenever an arc (i,j) is to be added to a list we check the ij bit of the adjacency matrix. If it is 1, we don't do anything. If it is 0, we make it 1 and add the arc in the successor list. All this is of cost O(1). We could have duplicate elimination done in O(1) time even if we used the adjacency matrix representation alone, but then we would not be able to search only existing arcs; we would have to scan the 0's of the matrix as well. This would increase the time complexities of all the algorithms.

Analyzing the I/O performance of the algorithms is very hard when taking into account the effect of buffering. For several of the algorithms concerned, the appropriate buffering strategy is not obvious. We felt that unless the algorithms are implemented and tested, the comparison may be unfair if we uniformly assume the same buffering strategy. Hence, in the forthcoming analysis we assumed minimal amount of buffering, i.e., we assume that the size of main memory is $O(n)$. Also, to simplify the analysis, we used a successor set as the unit of transfer between main memory and disk. Although successor sets may be very different in size and data is read from and written back to disk one page at a time, we believe that the number of successor set reads and writes gives an excellent indication of the actual I/O cost. For our analysis we will use the following parameters. (In the sequel, "strong component" refers to a nontrivial one.)

n	number of nodes in the graph
e	number of arcs in the graph
e_{con}	number of arcs in the condensation graph of a given graph
n_c	number of nodes in a strong component c
e_c	number of arcs in a strong component c
e_c^{out}	number of arcs emanating from nodes in a strong component c $(= \sum_{v \in c} d_v)$
t	number of arcs in the transitive closure
t_v	number of nodes reachable from node v
t_c	number of nodes reachable from (any node of) a strong component c
d_v	*out*-degree of v

We will also use the following notation for various necessary sets.

V	set of nodes in the graph
E	set of arcs in the graph
E_{con}	set of arcs in the condensation graph of a given graph
E_c	set of arcs in a strong component c
T	set of arcs in the transitive closure
SCC	set of strong components in the graph

Notice that $E = E_{con} \cup \sum_{c \in SCC} E_c$ and that $e = e_{con} + \sum_{c \in SCC} e_c$. Finally, we will use the O(.) notation for both cpu and I/O cost. We will retain, however, several of the constants of the various terms in the cost so the comparison between the various algorithms

can be more accurate. Also, the cost will always be broken into two parts, the *search part* and the *inference part*. In our notation, the inference part will be put within square brackets [...]. For example, a cost of $O(x+[y])$ indicates $O(x)$ search time and $O(y)$ inference time.

4.1. Basic_TC

The outer for-loop of *Basic_TC* is executed n times. For every node v, the while-loop may be executed t_v times in the worst case (i.e., when all nodes are reachable from v and they are all unmarked as they are discovered). The list manipulation inside the loop represents the number of arcs inserted in T (these may include duplicates). Put differently, it represents the number of inferences performed by the algorithm. Inserting the successors of w to the successors of v involves d_w additions. In addition, the initialization of S_v costs d_v additions. We conclude that the cpu cost of the algorithm is

$$cpu(Basic\_TC) = O(n + t + [e + \sum_{(v,w) \in T} d_w]). \quad (1)$$

One can verify that in the worst case this is an $O(n^3)$ algorithm.

We now turn to analyzing the I/O cost of *Basic_TC*. A node's original successor set is brought once into memory and from that point on stays there until it is processed completely. So, the outer loop represents n reads. The initialization step and the list manipulation steps require one read for each arc in T. So the total I/O cost of the algorithm is

$$i\_o(Basic\_TC) = O(n + [t]). \quad (2)$$

4.2. *Dag_DFTC*

Dag_DFTC is a straightforward adaptation of the depth-first algorithm, with an additional list manipulation every time we pop up from a node. The search part of the algorithm costs $O(n+e)$ time [Aho et al. 74]. This includes the calls to *visit* and the execution of the for-loop inside *visit*. In the inference part of the algorithm, every arc (v,w) in $T - E$ is inferred once for every successor of v that can reach w. Equivalently, this can be seen from the fact that every time we pop up from an arc (v,w) in E, w and its successors are added to the successors of v. Hence, the total complexity of *Dag_DFTC* becomes

$$cpu(Dag\_DFTC) = O(n + e + [e + \sum_{(v,w) \in E} t_w]). \quad (3)$$

In the worst case this can again be an $O(n^3)$ algorithm. Notice, however, the improvement over *Basic_TC*. On the search part, *Basic_TC* searches t arcs as opposed to e arcs. On the inference part, the two terms cannot be directly compared, but we can show that their average over all graphs is the same.

For the I/O cost, recall that we assume only minimal buffering (at least two successor sets, though). In the worst case, the successor set of a node is brought in from disk once for every call to the node and once for every pop-up to the node from one of its successors. The former corresponds to the search part and can involve up to $n+e$ calls (one for each incoming arc and one for a possible visit to the node from the outer level of the algorithm). The latter corresponds to the inference part and can involve up to e pop-ups. The worst case assumes that visits to a node from its predecessors and pop-ups to the node from its successors are far enough in time that the successor set of the node has been paged out. Hence, the I/O cost of *Dag_DFTC* is

$$i\_o(Dag\_DFTC) = O(n + e + [e]). \quad (4)$$

Notice again the improvement over *Basic_TC*.

4.3. *DFTC*

The general *DFTC* algorithm, which can handle cyclic graphs as well, is much more complex to analyze in comparison to the special algorithm for dags. This is due to the partitioning of the nodes reachable from another node into *tagged* and *marked* so that cycles can be identified, and due to the overhead of a second visit to the nodes in all nontrivial strongly connected components to adjust their sets of reachable nodes. For nodes that do not belong to a nontrivial strongly connected component, the algorithm performs exactly as *Dag_DFTC*. For nodes in nontrivial strongly connected components the following differences can be identified between the two algorithms with respect to their cost:

(a) Each strongly connected component is traversed in depth-first order a second time by calls to *visit2*. For a strongly connected component c, the cost of that is e_c^{out}. (There is no n_c factor here, because we always start from the root of the c and all the interesting nodes are known to be reachable from the root.)

(b) In the first pass, some of the transitive arcs from nodes in a strongly connected component are not inferred. Nevertheless, in the worst case, all those arcs will be inferred in the first pass too, and the inference cost of the first pass would be like the one for the acyclic graphs.

(c) The nodes reachable from nodes in a strongly connected component (except the root) are inferred once in the second pass. Some of them have already been inferred in the first phase, so

this may represent unnecessary work.

Incorporating all the above observations we may conclude that the cpu cost of *DFTC* is

$$cpu(DFTC) = O(n + e + \sum_{c \in SCC} e_c^{out} + [e + \sum_{(v,w) \in E} t_w + \sum_{c \in SCC} (n_c - 1)t_c]). \quad (5)$$

Notice that if *SCC* is empty, the formula reduces to (3). Also notice that most of the time the inferences in the first pass will be fewer than what is implied by the first summation in the inference part of the cost.

Comments similar to (a), (b), and (c) hold for the disk-based version of the algorithm. Assuming no buffering again, the cost of the first pass is exactly the same as it was before (in terms of successor set retrieval). In the second pass over a strongly connected component the successor sets of all the nodes in it are brought from disk once to be updated. For this we assume that the tagged successors of a node can be brought in separately (so that when a node has an empty tagged successor list nothing is brought in memory). They may need to be brought as many times as their out-degree, however, when *visit2* pops-up to the node. So, the extra I/O involved with the second visit of strongly connected components is n_c+e_c for each component c. The successor set of each node (except the root) is then updated (actually, assigned a value) once as well. This can be done, however, after we pop up to the node from its last child and we are ready to pop up to the parent of the node. Hence this cost has been already accounted as part of the search cost of the second pass. For uniformity, however, we will remove it from there and account it as inference cost. Given the above, the total number of extra I/O needed for that is n_c-1 for each component c. This brings the total I/O up to

$$i\_o(DFTC) = O(n + e + \sum_{c \in SCC} e_c + [e + \sum_{c \in SCC} (n_c - 1)]). \quad (6)$$

Again, if *SCC* is empty, (6) reduces to (4).

4.4. *Global_DFTC*

The last algorithm that was presented for reachability (Section 3.3) was *Global_DFTC*, which instead of popping up the list of nodes reachable from a strongly connected component to its root, it makes use of a global "stack" of successors. Thus, the number of inferences in the first pass over a component is minimized. Specifically, we observe the following:

(a) Search time for the first pass is $O(n+e)$. The total cost of manipulating the stack while the algorithm operates in a strongly connected component c is no more than $O(n_c)$. This is because, in the worst case, a new level is introduced to the "stack" for every back arc in the graph, there can be at most n_c back arcs in a strongly connected component, and because merging of two consecutive levels is of cost $O(1)$.

(b) Search time for the second pass over a strongly connected component c is $O(n_c)$. This is because, all the nodes of c have been collected in a separate list.

(c) In comparison to *DFTC*, the second pass costs the same in terms of inferences. There is a big win, however, over the first pass. Each node reachable from a strongly connected component is generated only once, unless it is outside the component and it is reachable from nodes in the component by two completely independent paths. This means that the set of arcs of the condensation graph E_{con} will be used as the basis of the inference, instead of the complete set of the arcs. That is, the number of inferences in the first pass will be $O(e_{con} + \sum_{(v,w) \in E_{con}} t_w)$. In addition, each node of a strongly connected component c is inferred once during the first pass over the component.

Adding up all the costs involved we conclude that the cpu cost of the algorithm is

$$cpu(Global\_DFTC) = O(n + e + \sum_{c \in SCC} n_c + \sum_{c \in SCC} n_c + [e_{con} + \sum_{(v,w) \in E_{con}} t_w + \sum_{c \in SCC} n_c + \sum_{c \in SCC} (n_c - 1)t_c]). \quad (7)$$

For the sake of marginally additional search time, the inference time of *Global_DFTC* is significantly smaller than that of *DFTC*.

Since the *stack* is assumed to be in main memory, the search part of the second pass over the strongly connected components costs no I/O. >From the first pass over the whole graph we have $O(n+e)$. In analogy to the cpu time, e_{con} successor sets are inferred during the first pass and n_c-1 during the second pass for every strong component c. Hence, the total I/O cost becomes

$$i\_o(Global\_DFTC) = O(n + e + [e_{con} + \sum_{c \in SCC} (n_c - 1)]). \quad (8)$$

The improvement over *DFTC* is again noticeable.

5. Selections

When a selection of the form "column1 = c" is specified, the algorithm deals with it effectively. (That is, we want to compute all tuples of the form (c,?) in the transitive closure.) In fact, the algorithm becomes much simpler. We need not do any numbering of nodes, and so we can directly run algorithm *Basic_TC*. Further, the first loop is no longer necessary. We can simply consider the selected node c and execute the inner loop.

On the other hand, a selection of the form "column2 = c" (i.e. compute all tuples of the form (?,c)) requires us to first generate a new representation for the relation p, which is the set of *predecessor sets*. The algorithm can then be used exactly as for the other selection.

Finally, consider a selection of the form "column1 = $c1$ and column2 = $c2$". That is, we simply wish to see if ($c1$,$c2$) is in the transitive closure. To do this, we proceed as in the case of selection "column1 = $c1$", with the difference that we can stop if $c2$ is added to SL_{c1}.

6. Related Work

A large body of literature exists for main-memory based algorithms for transitive closure. Recently, with the realization of the importance of recursion in new database applications, transitive closure has been revisited and re-examined in a data intensive environment. In this section, we will review a significant subset of the existing algorithms comparing them with ours. In particular, we compare *Global_DFTC* with the traditional Warshall and Warren algorithms [Warshall 62], [Warren 75], [Agrawal and Jagadish 87], an algorithm by Schmitz [Schmitz 83], and the Seminaive algorithm [Bancilhon 85]. We also discuss some other related work on transitive closure.

6.1. Schmitz

In all the relevant literature, the algorithm by Schmitz [Schmitz 83] is the one closest to our best algorithm for reachability, i.e., *Global_DFTC*. It is based on Tarjan's algorithm for identifying the strong connected components of a graph [Tarjan 72]. Schmitz showed that his algorithm had better performance than an algorithm by Eve and Kurki-Suonio [Eve and Kurki-Suonio 77], which we will not discuss further, as well as Warshall's algorithm [Warshall 62]. The common characteristics of Schmitz's algorithm and *Global_DFTC* are that (a) they are based on a depth-first traversal of the graph, (b) they identify the strong connected components of the graph, and (c) they take advantage of the fact that nodes in the same component have exactly the same descendants and that they are descendants of each other. On the other hand, the two algorithms differ in that (a) Schmitz is using a stack of nodes in the graph, whereas we use a "stack" of successor lists and (b) Schmitz is waiting for a whole strong connected component to be identified before it starts forming the descendant list of the nodes in the component, whereas we do that dynamically by associating partial descendant lists with the elements of the stack. Due to space limitations we do not present Schmitz's algorithm here. We will only give the formulas for its cost and compare them with the corresponding formulas of *Global_DFTC*. The basic idea of the algorithm is that when Tarjan's algorithm identifies a strong component, its nodes are at the top of the stack. Thus, Schmitz's algorithm scans the successor sets of all the elements of the component in the stack, and adds their descendants to the descendant list of the component.

Schmitz's algorithm (in its original form) finds the transitive closure of the condensation graph only. That is, it generates only one descendant list per strong component. To compare it with *Global_DFTC* uniformly, we assume that after the descendant list of the representative node of the component is found, it is copied to all other members of the component as well. With this modification the cost of Schmitz's algorithm is

$$cpu(Schmitz) = O(2n + 2e + n + [e_{con} + \sum_{(v,w)\in E_{con}} t_w + n_c + \sum_{c \in SCC} (n_c - 1)t_c]). \quad (11)$$

Comparing (11) to (7) we notice that the inference time is exactly the same: the two algorithms are identical. The search time, however, is different. In particular,

- Schmitz's algorithm always manipulates the stack, paying a cost of $O(n)$, whereas *Global_DFTC* manipulates the stack only when it operates in a nontrivial strong connected component, paying a cost of $O(\sum_{c \in SCC} n_c)$. Assuming that each operation on the stack costs roughly the same in the two algorithms, *Global_DFTC* wins. Also,
- Schmitz's algorithm delays the generation of the descendant list of any node until a complete strong connected component is found. Therefore, in its second pass it scans all the nodes and all their successors again, paying an additional cost of $O(n+e)$, whereas *Global_DFTC* simply scans the nodes in the nontrivial components, paying a cost of $O(\sum_{c \in SCC} n_c)$. *Global_DFTC* outperforms Schmitz's algorithm again.

A final note on the cpu performance of the two algorithms is that on acyclic graphs, the performance of *Global_DFTC* is the same as that of *DFTC*; no overhead is paid. In contrast, Schmitz's algorithm pays the extra overhead of a second pass and of manipulating the stack.

Analogous comments are appropriate for the I/O cost of the two algorithms. Assuming minimal buffering, the two major overheads for Schmitz's algorithm are the following:

- Since additions are delayed until a component is found, every time the algorithm pops up to a node v from a node w, v's successor list will be brought back without taking advantage of the fact that w's list is in memory. This accounts to an additional $O(e)$ in successor list reads during search time for Schmitz's algorithm.
- In the second pass over a strong connected component, we assume that all but one of its nodes have their successor lists on disk. Hence, $O(\sum_{c \in SCC} n_c)$ more lists have to be brought in during this phase.

According to the above, the I/O cost of Schmitz's algorithm becomes

$$i\_o(Schmitz) = O(n + 2e + \sum_{c \in SCC} n_c + [e_{con} + \sum_{c \in SCC}(n_c - 1)]). \quad (12)$$

Comparing (12) with (8) we see that the total overhead paid by Schmitz is $O(e + \sum_{c \in SCC} n_c)$ and is paid at search time. Regarding the inference part, the two algorithms are again identical. In the best case (which happens to be when the graph is one strong component), *Global_DFTC* wins by almost a factor of 2 in successor list I/O over Schmitz's algorithm. In the worst case (which happens when the graph is acyclic), and assuming that $e \geq n$, *Global_DFTC* outperforms Schmitz's algorithm by at least 1/3.

6.2. Seminaive

The Seminaive algorithm was developed as an algorithm to answer queries on general recursively defined relations [Bancilhon 85]. We present the algorithm in a way that resembles the algorithms we have developed in order to compare its time complexity with theirs. In particular, the descendants of every node are found first, before finding the descendants of any other node. In contrast, Seminaive works in stages, and at each stage k finds the descendants of all the nodes that are k arcs away from the node. This does not affect the cpu cost of the algorithm, whereas it should improve its I/O cost, since the descendant list of each node is not moved back and forth between main-memory and disk. Considering the main memory version of Seminaive, one realizes that it is equivalent to *Basic_TC* without taking marking into account. The algorithm is shown below.

```
proc Seminaive ( G ) {

Input: A Graph G specified using successor sets E_i, i =1 to n.
Output: S_i, i = 1 to n, denoting G*.

U_i := E_i; M_i := φ
for i := 1 to n do
        while there is j ∈ U_i-{i}
                do M_i := M_i ∪ {j}; U_i := U_i ∪ E_i - M_i od
od
```

Seminaive will always perform like *Basic_TC* if the latter is provided with the worst of ordering of nodes (so that no advantage can be taken from marking). Hence, its performance is given by the same formulas like *Basic_TC*, since they represent worst-case behavior. We would like to emphasize, however, that on the average, even *Basic_TC* will do much better than Seminaive, due to the effect of marking. 1†

Seminaive imposes an order on how U_i is processed. In particular, nodes are processed on a first-come-first-served basis, which corresponds to a breadth-first traversal of the nodes in the graph rooted in i. Since no marking is in effect, however, the order of processing does not affect the cpu time analysis in any way. The formula for the cpu cost is repeated below for ease of reference:

$$cpu(Seminaive) = O(n + t + [e + \sum_{(v,w) \in T} d_w]). \quad (13)$$

Comparing with *Global_DFTC*, we see that the inference parts are not directly comparable. We can show, however, that on cyclic subgraphs, *Global_DFTC* always wins, whereas on the acyclic part (the condensation graph) the two formulas have the same average over all graphs, but one can be better than the other on any specific graph. With respect to the cost of searching, the presence of t in Seminaive's cost formula, as opposed to e in *Global_DFTC*'s cost formula, makes *Global_DFTC* superior.

† In fact, this is how the algorithms were originally conceived. Marking provides a way of exploiting search order, and depth-first search provides a way of finding a good order. Further, focusing on one node at a time enables us to do duplicate elimination with no additional I/O since the required successor sets are always in memory, under the assumption that at least two sets fit into memory.

In terms of I/O, traditional implementations of Seminaive work by performing a sequence of joins of relations (i.e., successor list blocks). Blocking, however, can be applied to all the algorithms we have described so far. For example, instead of getting one node's successor set, one can bring a block's worth of successor sets and proceed appropriately. We believe that blocking affects all algorithms in this paper in the same manner. Hence, for the sake of comparison, we will adopt the *Basic_TC* I/O cost formula for Seminaive as well. It is given below:

$$i\_o(Seminaive) = O(n + [t]). \quad (14)$$

Comparing (14) with (8) we see that there are some cases where Seminaive will do better. A specific example is a graph that is fully connected, i.e., has n^2 nodes. In that case (14) gives $O(n+n^2)$ whereas (8) gives $O(2n+n^2)$. For most graphs, however, *Global_DFTC* is far superior to Seminaive.

6.3. Warshall and Warren

The traditional transitive closure algorithms are the one proposed by Warshall [Warshall 62] and its modification proposed by Warren [Warren 75]. They are both based on an adjacency matrix representation of the graph, and their main difference is the order in which they access the elements of the matrix. Both algorithms have $O(n^3)$ complexity, where the primitive operations are bit *or*'s and *and*'s. On the average, however, the Warren algorithm performs better than Warshall's. Moreover, this is true, for the most part, in disk-based implementations of the algorithms also [Agrawal and Jagadish 87]. Thus, we decided to discuss only the Warren algorithm. The Warren algorithm can be written in the notation we have developed as follows.

Input: A Graph G specified using successor sets E_i, i =1 to n.
Output: S_i, i = 1 to n, denoting G^*.

```
proc Seminaive ( G ) {

S := E;
for i := 1 to n do
      for j := 1 to i−1 do if j ∈ S_i then S_i := S_i ∪ S_j; od
od
for i := 1 to n do
      for j := i+1 to n do if j ∈ S_i then S_i := S_i ∪ S_j; od
od
```

This is the "straightforward implementation" [Agrawal and Jagadish 87] of the Warren algorithm written in terms of successor lists. We assume that the if-statement is checked while scanning over the range of j (i.e., the successor list of i is sorted). Since the way the algorithm will run depends on the names of (numbers assigned to) the nodes, it is relatively difficult to come up with a precise measure of the complexity of the algorithm. In the worst case, the two for-loops over j will be executed once for every descendant of i, except itself, (i.e., all descendants are inserted in front of j). In both loops, complete descendant lists might be added. With this pessimistic assumption, the worst case cpu cost of the algorithm is given by the formula

$$cpu(Warren) = O(n + t + [e + \sum_{(v,w) \in T} t_w]). \quad (15)$$

Comparing even against (13), (15) makes the Warren algorithm look even worse than Seminaive, let alone Global_DFTC. We believe, however, that on the average it will perform better than Seminaive. To get a better feeling for the Warren algorithm let us consider the best case. In that case, nothing happens in the second pass, and the first pass scans only original arcs (i.e., all descendants are inserted behind j). In that case the best case cpu cost of the algorithm is given by

$$cpu(Warren) = \Omega(n + e + [e + \sum_{(v,w) \in E} t_w]). \quad (16)$$

This can only happen if the graph is acyclic (this is just a necessary condition, not a sufficient one). Notice that (16) is equal to (3), which is the running time of Global_DFTC for the acyclic case. Although this is simply an indication and not a proof, it seems that Global_DFTC will never perform worse than the Warren algorithm, and in most cases it will perform much better.

Similar conclusions can be drawn in terms of the I/O performance of the Warren algorithm. Assuming no blocking, the worst and best case performance are given by the following formulas:

$$i\_o(Warren) = O(2n + [t]). \quad (17)$$

$$i\_o(Warren) = \Omega(n + [e]). \quad (18)$$

In the worst case, the Warren algorithm has worse I/O behavior than Seminaive, whereas in the best case it may outperform *Global_DFTC* by less than a factor of 2 ($n+e$ vs. $n+2e$). We believe that on the average *Global_DFTC* will perform much better than the Warren algorithm, but an average-case analysis and/or implementation is needed to establish this. There is, however, some empirical evidence in support of this conjecture. Agrawal and Jagadish have results that show that the I/O costs for Seminaive are 100 to 700 times more than the I/O costs for a careful implementation of Warren. This factor comes down to about 4

when the implementation of Seminaive is refined to reduce the cost of duplicate elimination [Agrawal and Jagadish 87]. We remarked earlier that the behavior of *Basic_TC* is similar to the performance of Seminaive (assuming no costs for duplicate elimination) when the ordering of nodes is such that the marking optimization never applies. We therefore expect that *Basic_TC*, and even more so *Global_DFTC*, will perform better than Seminaive by a significant factor on the average. Since the average case behavior of Seminaive is seen to be close to that of a careful implementation of Warren, this indicates that our algorithms will outperform Warren on the average.

We would like to emphasize here that the above analysis is done under the assumption of minimal buffering and *no blocking* of successor sets on disk. Agrawal and Jagadish's implementation of the Warren algorithm uses blocking extensively. Since the Warren algorithm is quite different in nature from the algorithms presented in this paper, it is hard to say whether blocking will affect the Warren algorithm and *Global_DFTC* in the same way. (Of course, the appropriate blocking and paging strategies will also differ significantly.) Further investigation is needed in this direction in order to compare the two algorithms with blocking.

6.4. Other Work

Besides Seminaive, another popular algorithm that has been proposed for general recursion is the *Smart* or *Logarithmic* algorithm [Valduriez and Boral 86, Ioannidis 86]. The idea behind the algorithm is to first compute all the pairs of nodes that are a number of arcs apart that is a power of 2, and then compute the remaining arcs performing much fewer operations than would otherwise be needed (i.e., if Seminaive was used). Regarding the transitive closure of a graph, it has been shown that Smart outperforms Seminaive for a large class of graphs and under varying assumptions about storage structures and join algorithms. The power of the algorithm relies heavily on computing sets of arcs, so it is hard to formulate it in a way that can be directly compared with the algorithms presented in this paper. It has been shown, however, that the straightforward implementation of the Warren algorithm sometimes performs better than Smart and sometimes worse, whereas the blocked implementation uniformly outperforms Smart. We speculate that since our analysis showed that *Global_DFTC* outperforms the Warren algorithm, it will outperform Smart as well.

A straightforward disk-based implementation of Warren's algorithm was proposed and tested against Smart/Logarithmic [Lu, Mikkilineni, and Richardson 87]. It used hashing as a basic storage structure and employed hash-based join techniques. The cost of the algorithm was analyzed and compared to the cost of two versions of Smart/Logarithmic. The analysis was much more detailed than the one presented in this paper for the Warren algorithm, since the cost of buffering and hashing had to be taken into account. The main results of the analysis were that the Warren algorithm works better than Logarithmic when there is ample main memory available and when there is a great variation in the lengths of the various paths in the graph. As we mentioned above, another implementation of the Warren algorithm, much better suited to disk-based data, was developed by Agrawal and Jagadish [Agrawal and Jagadish 87]. They used blocking to improve the performance and provided empirical evidence that the algorithm outperforms both Seminaive and Smart/Logarithmic almost uniformly.

Lu proposed another algorithm for reachability that uses hash-based join techniques to compute the transitive closure of a relation [Lu 87]. Its basic structure is that of Seminaive, but it employees two interesting tricks that speed up computation: (a) the original relation is dynamically reduced by eliminating tuples that are known to be useless in the further production of the transitive closure, and (b) as soon as a tuple is produced, if it is inserted in the same hash bucket that is being processed, the tuple is processed also. Lu showed that for a restricted class of graphs his algorithm performs better than both Seminaive and Smart/Logarithmic.

In the context of the Probe DBMS prototype, transitive closure was identified as an important class of recursion and was generally termed *traversal recursion* [Rosenthal et al. 86]. Traversal recursion was formally specified using path algebras [Carre 79], and it focused primarily on path computation problems. The algorithms proposed for traversal recursion were Seminaive and *one-pass traversals*, i.e., algorithms that need to traverse a graph only once. It was argued that one-pass traversals are better than Seminaive, but no formal argument or empirical results were provided. Under the assumptions made in this paper, our results confirm the above claim (at least for reachability).

7. Path Computations, One-sided Recursion, Parallelism

In this paper, we have focussed on the reachability problem, presenting a number of increasingly sophisticated algorithms and analyzing their performance. While this analysis shows that these algorithms perform efficiently, they do not bring out what we consider to be one of their most important assets, which is their broad applicability and versatility. They are easily adapted to deal with path computations, in which we

ask for aggregate properties such as the shortest path between two points, and one-sided recursions, which is a class of recursive programs that generalizes transitive closure [Naughton 87]. Some of the algorithms can be adapted for parallel evaluation, and to take advantage of infrequent updates. We discuss these issues briefly in this section.

An important generalization of reachability is the problem of path computations. Examples include finding the shortest path between two points, bill-of-materials, and other problems of practical significance. A number of transitive closure algorithms cannot deal with path computations [Schnorr78, Schmitz83]. Of the algorithms presented in this paper, only *Global_DFTC* cannot be adapted to deal with path problems, since it loses path information in processing strongly connected components. We have adapted *DFTC* to perform path computations, proved it correct, and analyzed its performance [Ioannidis and Ramakrishnan 88]. The adaptation is straightforward. As with the reachability problem, selections can be dealt with efficiently. Thus, we can effectively find the shortest path from a given node to every other node in the graph. (In this special case, it coincides with Dijkstra's algorithm for shortest paths.)

One-sided recursions form a class of recursive programs that generalize transitive closure. They are presented as a class of programs that permit efficient algorithms for selection queries [Naughton 87]. We have considered how the algorithms in this paper can be adapted to deal with one-sided recursions [Ioannidis and Ramakrishnan 88]. For selections, *Basic_TC*, suitably refined, coincides with the algorithm presented by Naughton [Naughton 87]. For computing queries that do not involve selections, the adapted algorithm may perform better than Seminaive (which is the algorithm that Naughton suggests in this case).

Finally, we remark that the simplest algorithm presented in this paper, *Basic_TC*, may often be the algorithm of choice. This is for two reasons. First, consider a situation in which the graph is acyclic (or close to acyclic) and updates are infrequent. We can store the relation according to a reverse topological ordering, and re-organize it periodically to restore this property (which may be affected by intervening updates). If *Basic_TC* is run on such a relation, it obtains much of the improvement in *DFTC*, since the depth-first order of processing (which is achieved in *DFTC* by the order of calls) is achieved through the order in which the nodes are stored (and selected for processing by *Basic_TC*). *DFTC* improves on *Basic_TC* in this case only when there are cycles. In fact, *Basic_TC* might well outperform *DFTC* since it does not have the overhead of setting up the calls to *visit* 1, which involves the I/O of fetching in successor lists. An adaptation of *Basic_TC* for path computations is of particular interest since many path computations are based on the acyclicity of the underlying graph.

The second reason for choosing *Basic_TC* has to do with its potential for parallel evaluation. The addition of successor set S_j to S_i in the loop can be parallelized. Further, the loop can simultaneously be executed for more than one node. (In doing this, we might lose some of the benefits of the depth-first ordering, but this is a trade-off that can be refined.)

Space limitations prevent us from developing the ideas in this section further. We refer the interested reader to [Ioannidis and Ramakrishnan 88].

8. Conclusions

We have presented several closely related algorithms for evaluating a broad range of queries related to transitive closure. With the exception of Seminaive, no other approach offers efficient performance over such a variety of queries, including selections, single-source and all-sources path problems, and even one-sided recursions. Our analysis indicates that this flexibility is not achieved at the cost of efficiency; indeed, in many cases, the algorithms are seen to reduce to well-known algorithms (e.g. Dijkstra's algorithm) or to do better than less flexible algorithms (e.g. Schmitz). The algorithms are similar to the Schmitz algorithm and some other algorithms that identify strongly connected components and compute the transitive closure over the condensation graph in that they exploit a topological ordering of nodes. They differ significantly in not separating the identification of the components from the transitive closure phase, and in not merging all nodes in strongly connected components *a-priori*. The first of these differences offers a computational advantage, whereas the latter allows the adaptation of these algorithms to path problems.

We view this work as a first step. Our analysis, while it indicates the promise of the algorithms presented here, still needs to be refined and supplemented by a comprehensive performance evaluation based on actual implementations of the algorithms. We also need to explore the effect of the various heuristics mentioned in the paper, and to study the relationship of the more sophisticated algorithms to one-sided recursions.

9. References

[Agrawal and Jagadish 87]
Agrawal, R., and H. V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database

Relations", *Proc. of the 13th International VLDB Conference*, Brighton, England, September 1987, pp. 255-266.

[Agrawal et al. 87]
Agrawal, R., S. Dar, and H. V. Jagadish, "Transitive Closure Algorithms Revisited: The Case for Path Computations", unpublished manuscript, December 1987.

[Agrawal and Jagadish 88]
Agrawal, R., and H. V. Jagadish, personal communication, January 1988.

[Aho et al. 74]
Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Bancilhon 85]
Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", Technical Report DB-004-85, MCC, Austin, TX, 1985.

[Carre 79]
Carre, B., *Graphs and Networks*, Clarendon Press, Oxford, England, 1979.

[Dijkstra 59]
Dijkstra, E. W., "A note on two problems in connection with Graphs", *Numerische Mathematik*, Vol. 1, pp. 269-271.

[Eve and Kurki-Suonio 77]
Eve, J. and R. Kurki-Suonio, "On Computing the Transitive Closure of a Relation", Acta Informatica, Vol. 8, 1977, pp. 303-314.

[Ioannidis 86]
Ioannidis, Y. E., "On the Computation of the Transitive Closure of Relational Operators", *Proc. of the 12th International VLDB Conference*, Kyoto, Japan, August 1986, pp. 403-411.

[Ioannidis and Ramakrishnan 88]
Ioannidis Y. E. and R. Ramakrishnan, "Efficient Transitive Closure Algorithms", Technical Report #765, Computer Sciences Dept., University of Wisconsin, Madison, April 1988.

[Lu 87]
Lu H., "New Strategies for Computing the Transitive Closure of a Database Relation", *Proc. of the 13th International VLDB Conference*, Brighton, England, September 1987, pp. 267-274..

[Lu, Mikkilineni, and Richardson 87]
Lu, H., K. Mikkilineni, and J. P. Richardson, "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation", *Proc. of the 3rd International Data Engineering Conference*, Los Angeles, CA, February 1987, pp. 112-119.

[Naughton 87]
Naughton, J. F., "One-Sided Recursions", *Proc. of the 6th ACM-PODS Conference*, San Diego, CA, March 1987, pp. 340-348.

[Rosenthal et al. 86]
Rosenthal, A., et al., "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", *Proc. of the 1986 ACM-SIGMOD Conference*, Washington, DC, May 1986, pp. 166-176.

[Schmitz 83]
Schmitz, L., "An Improved Transitive Closure Algorithm", *Computing*, Vol. 30, 1983, pp. 359-371.

[Schnorr 78]
Schnorr C. P., "An Algorithm for Transitive Closure with Linear Expected Time", SIAM J. Computing, Vol. 7, No. 2, May 1978, pp. 127-133.

[Tarjan 72]
Tarjan, R. E., "Depth First Search and Linear Graph Algorithms", *SIAM Jour. of Computing*, Vol. 1, No. 2, 1972, pp. 146-160.

[Valduriez and Boral 86]
Valduriez, P., and H. Boral, "Evaluation of Recursive Queries Using Join Indices", *Proc. of the 1st International Expert Database Systems Conference*, Charleston, SC, April 1986, pp. 197-208.

[Warren 75]
Warren, H. S., "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations", *CACM*, Vol. 18, No. 4, April 1975, pp. 218-220.

[Warshall 62]
Warshall, S., "A Theorem on Boolean Matrices", *JACM*, Vol. 9, No. 1, January 1962, pp. 11-12.