EFFICIENT TRANSITIVE CLOSURE ALGORITHMS

by

Yannis E. Ioannidis
and
Raghu Ramakrishnan

Computer Sciences Technical Report #765

April 1988

EFFICIENT TRANSITIVE CLOSURE ALGORITHMS

Yannis E. Ioannidis
Raghu Ramakrishnan

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

(Submitted to VLDB '88)

Efficient Transitive Closure Algorithms

Yannis Ioannidis †
Raghu Ramakrishnan

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

We present some efficient algorithms for computing the transitive closure of a directed graph. The algorithms are adapted to compute a number of related queries such as the set of nodes reachable from a given node, or queries posed over the set of paths in the transitive closure such as the shortest path between each pair of nodes, or the longest path from a given node. We indicate how these algorithms could be adapted to a significantly broader class of queries based on one-sided recursions. We also 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 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, and 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 algo-

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

rithms, starting with some simple versions and subsequently refining them. We present an analysis of these algorithms in Section 4. We consider selection queries in Section 5, and path computations in Section 6. In Section 7, we consider how some of these algorithms can be adapted to deal with one-sided recursions. We discuss related work in Section 8, and present our conclusions in Section 9.

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 $E_i = \{ j \mid (i, j)$ is an arc of G $\}$.

We denote the transitive closure of a graph G by G^*. The *strongly connected component* of node i is defined as $V_i = \{ i \} \cup \{ j \mid (i, j) \in G^*$ 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 emphasize that we do not suggest using this algorithm in general; we present better algorithms, which are derived by refining this algorithm.

```
proc Basic_TC( $G$ )
Input: A digraph $G$ specified using successor sets $E_i$, i = 1 to n.
Output: $S_i = U_i \cup M_i$, i = 1 to n, denoting $G^*$.
{ $U_i := E_i$; $M_i := \Phi$
 for $i$ = 1 to n do
      while there is a node $j \in U_i$ do $M_i := M_i \cup M_j \cup \{ j \}$;  $U_i := U_i \cup U_j - M_i$ od
 od
}
```

Proposition 3.1: $j \in M_i \Rightarrow E_j \subseteq M_i \cup U_i$.

Proof: Whenever a node j is added to M_i, $M_j \cup U_j$ is also added to $M_i \cup U_i$. The claim follows from the observation that initially $M_i = \Phi$ and $U_i = E_i$, and $M_i \cup U_i$ is monotonically increasing, for all i. []

Lemma 3.2: Algorithm *Basic_TC* correctly computes the transitive closure of a directed graph G.

Proof: $j \in M_i \cup U_i$ implies that $j \in E_i$ or that there is some node k such that $k \in E_i$ and $j \in M_i \cup U_i$. It follows that only nodes that are reachable from i are in $M_i \cup U_i$. To see that all such nodes are in $M_i \cup U_i$, we note that

† 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.

when the algorithm terminates $U_i = \phi$ for all i. The proof is completed by the fact that initially $U_i = E_i$ and $M_i \cup U_i$ is monotonically increasing for all i, and Proposition 3.1. []

3.2. Depth-First Search to Number Nodes

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. This is reflected by the numbering - *popped*[j] is > *popped*[i].) We now present the depth-first numbering algorithm.

```
proc number ( $G$ )
Input: A graph $G$ represented by successor sets $E_i$.
Output: Graph $G$ with nodes numbered.
{ $vis$ = 1;
 for $i$ = 1 to n do $visited[i] := 0$; $popped[i] := 0$ od
 while there is some $i$ s.t. $visited[i] = 0$ do visit($i$) od
}

proc visit(i)
{ $visited[i] := 1$;
 while there is $j \in E_i$ s.t. $visited[j] = 0$ do visit($j$) od
 $popped[i] := vis$; $vis := vis + 1$;
}
```

The above algorithm implicitly establishes a spanning forest over the graph G. There is an arc (i, j) in the spanning forest if there is a call to *visit*(j) during the execution of the call *visit*(i). It is easy to establish that this is a spanning forest since there is exactly one call *visit*(i) for each node i. An arc (i, j) in the graph G is called a *forward arc* if j is a descendant of i, a *back arc* if j is an ancestor of i, and a *cross arc* if there is no ancestor-descendant relationship between i and j in the spanning forest. The following lemma identifies an important property of the spanning forest induced by algorithm *number*.

Lemma 3.3: ([Aho et al. 74]) Let $G1$ be a strongly connected component of a directed graph G. Then, the vertices of $G1$ together with those of its arcs that are common to the spanning forest form a tree. []

The node in the connected component which is the root of this tree is called the *root of the strongly connected component*.

The following lemma identifies an important property of the numbers assigned by the above algorithm.

Lemma 3.3: ([Hecht 77]) Arc (i, j) is a back arc in the spanning forest if and only if *popped*[i] < *popped*[j]. []

3.3. A Depth-First Transitive Closure Algorithm

We have presented the numbering algorithm as a preprocessing phase for algorithm *Basic_TC*. While this 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; $S_i$ := $\phi$ 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 $j \in E_i - S_i$ do
      do if visited[$i$] = 0 then { visit(j) }; $S_i := S_i \cup S_j \cup \{ j \}$ 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: $S_i = M_i \cup T_i$, i = 1 to n, denoting $G^*$.

{ vis := 1;
  for $i$ = 1 to n do visited[$i$] := visited2[$i$] := popped[$i$] := 0; $M_i$ := $T_i$ := Global := $\phi$ 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 $j \in E_i - M_i - T_i$ do
      if visited[$j$] = 0 then visit(j);
      if popped[$j$] > 0 then { $M_i := M_i \cup M_j \cup \{ j \}$; $T_i := (T_i \cup T_j) - M_i$ }
                           else $T_i := T_i \cup \{ j \}$
  od
  if $i \in T_i$ then { if $T_i = \{ i \}$ then { $M_i := M_i \cup \{ i \}$; $T_i = \phi$; Global := $M_i$; visit2($i$) } }
                   else { $T_i := T_i - \{ i \}$; $M_i := M_i \cup \{ i \}$ }
  popped[$i$] := vis; vis := vis + 1
}

proc visit2( $i$ )
{ visited2[$i$] := 1;
  while there is $j \in E_i$ s.t. visited2[$j$] = 0 and $T_j \neq \phi$ do visit2(j) od
  $M_i$ := Global; $T_i$ := $\phi$
```

}

Theorem 3.4: Algorithm *DFTC* correctly computes the transitive closure of G.

Proof: We note that the nodes are assigned the same numbers by algorithm *DFTC* as by algorithm *number*, and so a spanning forest is induced whose back arcs (i, j) are characterized by $popped[i] < popped[j]$.

Claim 1: After the execution of the while-loop in *visit* $1(i)$, $M_i = \{ j \mid j$ is reachable from i through a path that does not contain a back arc $\}$, and $T_i = \{ k \mid$ there is some $j \in M_i$ s.t. (j, k) is a back arc, and not $k \in M_i \}$.

This claim can be established by a simple induction on the height of the subtree rooted at i.

Claim 2: After the execution of the while-loop in *visit* $1(i)$, $T_i = \{ i \}$ if and only if i is the root of a connected component.

This claim follows immediately from the previous claim.

We observe that for every node j in a strongly connected component rooted at a node i, the call to *visit*2 when the root i is identified sets $M_j = M_i$, and $T_i = \emptyset$. Thus, after the call *visit* $1(i)$, where i is the root of a strongly connected component, M_j contains all successors of j for every node in this component, and $T_j = \emptyset$. It is now easy to show that $T_i = \emptyset$ after the call *visit* (i) for all nodes i which do not belong to a connected component. The proof of Theorem 3.4 is completed by the observation that when algorithm *DFTC* terminates, all connected components have been identified, and so M_i includes all successors of i for all nodes i. (The termination of the algorithm is straightforward, and can be shown through induction on the height of the tallest tree in the spanning forest.) []

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 *vistil* 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].

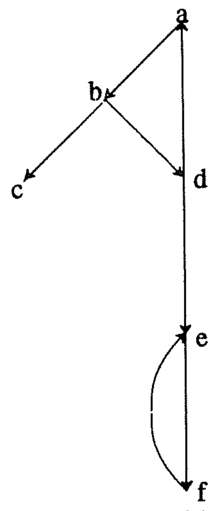

Figure 3.1: A graph with cycles.

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.

3.4. Optimized Processing of Nontrivial Strong Components

There is a major potential inefficiency in *DFTC* in that the second pass over a strong component re-infers 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.

To deal with this problem, we now present a further refinement of algorithm *DFTC*. In this version of the algorithm, we do not need to distinguish tagged elements by partitioning successor lists, since a stack mechanism hat 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 notation $L_1 := L_1 \bullet L_2$ 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 $\emptyset$ (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 $G^*$.
{ 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
```

[†] 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. (The space for storing the graph is $O(n^2)$ in any case.)

```
 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 $j \in E_i - \{ i \}$ s.t. $visited[j] = 0$ do
      visit(j);
      if $popped[j] > 0$ and $ptr[j] = n+1$ then $S_i := S_i \cup S_j \cup \{ j \}$;
      if $popped[j] > 0$ and $ptr[j] \neq n+1$
           then { $bot := \min(top, ptr[i], ptr[j])$;
                     while $top > bot$ do
                          $list[top-1] := list[top-1] \bullet list[top]$; $nodes[top-1] := nodes[top-1] \bullet nodes[top]$;
                          if $visited[root[top]] < visited[root[top-1]]$ then $root[top-1] := root[top]$;
                          $top := top - 1$;
                     od
                     if $ptr[i] = n+1$ then $list[top] := list[top] \bullet S_i$;
                     $ptr[i] := top$; $S_i := \bullet\, list[top]$
                     };
      if $popped[j] = 0$
           then { $top := top + 1$; $root[top] := j$; $list[top] := \bullet S_i$; $nodes[top] := nil$; $ptr[i] := top$ }
 od
 if $i = root[top]$
      then { for each $j \in nodes[top] \cup \{ i \}$ do $S_j := list[top] \cup \{ i \}$; $ptr[j] := n+1$ od; $top := top - 1$ }
 elseif $ptr[i] \neq n+1$ then { $list[ptr[i]] := list[ptr[i]] \cup \{ i \}$; $nodes[ptr[i]] := nodes[ptr[i]] \cup \{ i \}$ };
 $popped[i] := 1$
}
```

Theorem 3.5: Algorithm *Global_DFTC* correctly computes the transitive closure of G.

Proof (Sketch): We develop our proof by establishing several claims about the algorithm. We first prove a simple claim about subgraphs of G which are dags.

Claim 1: Let the subgraph rooted at i in G be a dag. Then, in the calls $visit(j)$, for all nodes j is in this subgraph, $popped[k] > 0$ and $ptr[k] = n+1$ for all children k of j after $visit(k)$.

The claim is easily proved by induction on the height of the dag. Thus, only the first if-statement is considered in each of these calls. From the statement, it follows that for all nodes j in this subgraph, S_j contains all successors of j when $visit(j)$ terminates.

Claim 2: In the call $visit(i)$, a new level is added to the top of the stack if and only if (i,j) is a back arc.

This follows from the fact that $popped[j] = 0$ (when testing the conditions of the if-statements in procedure *visit*) if and only if (i,j) is a back arc, and from the **if** statement which is executed when $popped[j] = 0$.

Claim 3: 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 is the important invariant property that underlies the algorithm. We present an informal justification of this claim. The formalization of this proof relies upon an induction over the condensation graph for G. New levels are added to the stack only when back arcs are found. When a new level is created in the call $visit(i)$, the *root* field records the node reached via the back arc, and $ptr[i]$ is assigned the current level (top) of the stack. Further, the successor set S_i is concatenated to the set *list*[*top*], and S_i is then identified with (i.e., via the pointer denoting S_i) to *list*[*top*]. This indicates that i is part of a nontrivial connected component, and as the last step in *visit*(i), i is added to *nodes*[*top*] and *list*[*top*], unless i = *root*[*top*]. (We show later that i is the root of the component if i = *root*[*top*].) Further, if j is a child of i, and j does not belong to this connected component, the successor list of j is complete when the call *visit*(j) terminates. (This is a property of the depth-first nature of the algorithm, and we need an induction on the condensation graph of G to establish this formally.) This successor list is added to the successor list of i (through the execution of the first *if* statement), and thus, to the set of nodes in *list*[*top*] if i is part of a connected component. Finally, if j is a child of i, and j belongs to the same component as i (either *popped*[j] > 0 and *ptr*[j] $\neq n+1$, or *popped*[j] = 0) we update *root*[*top*] to the earliest visited node that is reachable through either j or through a previously discovered member of the connected component (that contains i and j).

*Claim*4: If *root*[*top*] = i after the execution of the **while** statement in the call *visit*(i), then i is the root of a connected component, the set *nodes*[*top*] contains all other members in the component, and the set *list*[*top*] contains all nodes which are reachable from the component.

Since *root*[*top*] = i, there must be some descendant j of i from which there is a back arc to i. Thus, i belongs to a nontrivial strongly connected component. Further, i is the earliest visited node that is reachable from this component (by Claim 3). Thus, i is the root of this component. The call *visit*(j) has already terminated for every node in this component (since i is the root of the spanning forest induced by the sequence of calls), and so every node in the component, other than i, has been added to *nodes*[*top*]. Similarly, all their successor lists have been added to *list*[*top*] (and these lists include the complete successor lists of all children of these nodes which do not belong to the connected component rooted at i). This concludes the proof of Claim 4.

The theorem is established by a simple induction on the condensation graph, using Claims 1 and 4. []

From the proof of the algorithm it should be clear how 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.

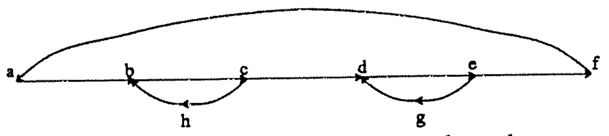

Figure 3.2: A strongly connected graph.

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 8, 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.

Regarding the I/O performance of the algorithms, it is very hard to analyze while 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 use 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.

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 (all the 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

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

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, the successors of w and w 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

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

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 for each component c. This brings the total I/O up to

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

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

4.4. *Global_DFTC*

The last algorithm that was presented for reachability (Section 3.4) 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

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

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 ones have to be done during the second pass for every strong component c. Hence, the total I/O cost becomes

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

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. Path Computations

6.1. Algorithm

We have considered how to compute the transitive closure of a graph. We have also considered how to compute this closure given a selection of the form "column = c". The latter problem is in fact that of finding all points in the graph which are reachable from c. In this section, we consider how to compute several properties that are specified over the set of paths in the graph. We call such properties *aggregate* properties, and we refer to the computation of such a property as a *path computation.*

We use the path algebra formalism developed in [Carre 79], and the terminology in [Rosenthal et al. 86, Agrawal et al. 87], which we present below. There is an *arc* $E(i,j)$ from node i to node j, if and only if $j \in E_i$. There is a *label* L_{ij} associated with each arc $E(i,j)$. Node i is called the *source* and node j is called the *destination* of this arc. A *path* from node i to node j, $P(i,j)$, is an ordered set of arcs $\{E_k(source, destination)\}$, $k = 1, \ldots, n$, such that $i = E_1.source$, $E_1.destination = E_2.source$, ... , $E_n.destination = j$. A label may be associated with the path $P(i,j)$. This *path label* is computed as a function, called CON (for concatenate), of the labels of the arcs in $P(i,j)$. We sometimes specify the path by the sequence of nodes on it. A *path set* is any set of paths, and we can associate a label with a path set S. This *path set label* is computed as a function, called AGG (for *aggregate*), of the path labels of the paths in S. CON must be associative. Let $E(i,j)$, $E(j,k)$ and $E(k,l)$ be three arcs, with associated labels L_{ij}, L_{jk}, L_{kl}:

$$\text{CON}(L_{ij}, \text{CON}(L_{jk}, L_{kl})) = \text{CON}(\text{CON}(L_{ij}, L_{jk}), L_{kl})$$

AGG must be associative, commutative and idempotent. Let P_1, P_2 and P_3 be any three paths, with associated labels L_1, L_2 and L_3:

$$\text{AGG}(L_1, \text{AGG}(L_2, L_3)) = \text{AGG}(\text{AGG}(L_1, L_2), L_3)$$
$$\text{AGG}(L_1, L_2) = \text{AGG}(L_2, L_1)$$
$$\text{AGG}(L, L) = L$$

Finally, there is a *zero label* ϕ and a *unit label* θ such that for all labels L,

$$\text{AGG}(\phi, L) = L, \text{CON}(\theta, L) = \text{CON}(L, \theta) = L$$

The following table, from [Carre 79, Rosenthal et al. 86], shows how several familiar problems can be described in this terminology.

Problem	CON	AGG
Reachability	AND	OR
Shortest Path	Add	Min
Critical Path	Add	Max
Maximum Capacity Path	Min	Max
Most Reliable Path	Multiply	Max
Bill of Materials	Multiply	Add

The path set in each of the above definitions is the set of all paths in the graph.

A path problem can also be formulated using matrices. A graph can be defined using an adjacency matrix A

where $A_{ij} = L_{ij}$, for each arc $E(i,j)$. Let A^k denote the kth power of A. Then, L_{ij}^k denotes the label for the set of paths from i to j of length at most k. For a path problem to be well-defined, A must be *stable*, that is, for some k, $A^k = A^{k+1}$. A graph G with operations AGG and CON defined over its labels is said to be *absorptive* if for every cycle γ in G, AGG(CON(γ), θ) = θ. If <G, AGG, CON> is absorptive, then (the adjacency matrix associated with) it is stable [Carre 79]. All the examples in the table presented earlier are absorptive. In this paper, we assume that <G, AGG, CON> is absorptive for the given path problem.

The path label for a path can always be computed as we construct the path. Consider paths $P1(i,j)$ and $P2(j,k)$ with associated labels L_{ij} and L_{jk}. We can construct a new path $P3(i,k)$ with label L_{ik} by concatenating these paths, and the label for the new path can be computed by applying CON to their labels. Using a triple to denote a path and its label:

$$P1(i,j,L_{ij}) \,.\, P2(j,k,L_{jk}) = P3(i,k,\mathrm{CON}(L_{ij},L_{jk}))$$

This follows from the associativity of CON.

Similarly, the straightforward way to compute a path set label is to compute each path in the set, along with its path label, and to apply AGG to the set of path labels as they are generated. This is clearly very expensive (exponential in the size of p) since we must explicitly enumerate all paths in the set.

An important property of path set labels is identified in [Agrawal et al. 87, Carre 79]. † Efficient path computations are possible if AGG and CON obey the distributive law:

$$\mathrm{AGG}\,(\,\mathrm{CON}(a,b), \mathrm{CON}(a,c)) = \mathrm{CON}\,(\,a, \mathrm{AGG}(b,c))$$

(An example of a path set label which does not satisfy this property is the *paint problem*, discussed in [Agrawal et al. 87]. Let each path from a node i to a node j be painted with a different color. The problem is to compute the amount of paint needed. For this problem, both AGG and CON are the addition function, and addition does not distribute with respect to itself.)

We now consider the intuition behind why this property permits efficient path computations. Consider the case when there are two paths from node j to node k, say $P1(j,k)$ and $P2(j,k)$, with associated labels L_1 and L_2. Let there also be a path $P(i,j)$ from node i to node j, with label L. Let S be the set of paths from i to k. The problem is to compute AGG for this set. We would like to do this without explicitly enumerating both the paths $P(i,j)$. $P1(j,k)$ and $P(i,j)$. $P2(j,k)$. In particular, suppose we apply AGG to $P1$ and $P2$, and simply record label $L' = \mathrm{AGG}(L_1, L_2)$, instead of L_1 and L_2, with the ordered pair (j,k). Now, instead of generating the two paths from i to k, we only generate a single path from i to k, with the label CON(L,L'). If AGG and CON are distributive, then this is indeed the correct path set label for the set of paths from i to k.

This property is easily exploited to generalize the transitive closure algorithms we presented earlier to also do path computations. First, we must augment the successor set representation to also record labels. Consider successor set S_i, and let j be a node in it. There is a label L_{ij} associated with this node. Initially, when the set of successor

† Carre included this property in his definition of a path algebra. Agrawal et al. do not require this in their definition, but in all their algorithms, they assume that it holds. (They also assume that <G, AGG, CON> is absorptive.)

lists denotes the graph G, this is the label associated with the arc (i,j) in p. After the path computation is completed, this is the path set label associated with the set of all paths from node i to node j. Next, we must modify the algorithm itself. Algorithm Global_DFTC cannot be used, however, since it loses path information, and so we use algorithm *DFTC*. To simplify the presentation, we make the following assumption: If S_i does not contain node k, the label L_{ik} is taken to be the zero label ϕ.

```
proc Path_DFTC ( $G$ )
Input: A directed graph $G$ represented by successor sets $E_i$ in which node $j$ has label $L_{ij}$.
Output: $S_i = M_i \cup T_i$, i = 1 to n, denoting $G^*$.
{ $vis := 1; pop := 1$
 for $i$ = 1 to n do $visited[i] := visited2[i] := 0; popped[i] := 0; M_i := T_i := \phi$ od
 while there is some node $i$ s.t. $visited[i] = 0$ do visit1(i) od
}

proc visit1 ( $i$ )
        { $visited[i] := vis, vis := vis + 1$;
 while $(E_i - M_i - T_i) \neq \phi$ do
      choose $j \in (E_i - M_i - T_i)$;       /* s.t. $L_{ij} \leq L_{ik}$ for all $k \in T_i$ */
      if $visited[j] = 0$ then visit1(j);
      if $popped[j] > 0$
            then { for all $k \in S_j$ do
                        $L_{ik}^{old} := L_{ik}$; $L_{ik}$ := AGG ($L_{ik}$, CON ($L_{ij}, L_{jk}$));
                        if $L_{ik}^{old} \neq L_{ik}$ and $k \in M_j$ then $M_i := M_i \cup \{ k \}$
                   od
                   $M_i := M_i \cup \{ j \}$
                   }
            else $T_i := T_i \cup \{ j \}$;
      $T_i := T_i \cup T_j$
 od
 if $i \in T_i$
      then { if $T_i = \{ i \}$ then { $M_i := M_i \cup \{ i \}; T_i = \phi$, $visit2$ } }
      else { $T_i := T_i - \{ i \}; M_i := M_i \cup \{ i \}$ }
 $popped[i] := pop; pop := pop + 1$
}

proc visit2
{ $visited2[n] := 1$;
 while there is $i \in E_n$ s.t. $visited2[i] = 0$ and $T_i \neq \phi$ do
      choose such a node $i$ with the smallest $visited[i]$;
      while $T_i \neq \phi$ do
            choose $j \in T_i$;  /* s.t. $popped[j] \geq popped[k]$ for all $k \in T_i$ */
            for all $k \in S_j$ do
                  $L_{ik}^{old} := L_{ik}$; $L_{ik}$ := AGG ($L_{ik}$, CON ($L_{ij}, L_{jk}$));
                  if $L_{ik}^{old} \neq L_{ik}$ and $k \in M_j$ then $M_i := M_i \cup \{ k \}$
            od
            $M_i := M_i \cup \{ j \}; T_i := T_i - M_i$
      od
      $visit2(i)$
 od
}
```

We state the following simple lemma without proof.

Lemma 5.1: If $j \in T_i$, then *visited*$1[j]$ < *visited*$1[i]$, for all nodes m, n. Further, for every node i in a connected component, if $j \in T_i$, then j is also in this component. []

We now prove the correctness of algorithm *Path_DFTC*.

Theorem 5.2: Algorithm *Path_DFTC* terminates with M_i equal to the set of all successors of i, for all i, and L_{ij} equal to the correct label for the set of paths from i to j for all nodes $j \in M_i$, if $<G$, AGG, CON> is absorptive, and CON is distributive over AGG.

Proof: The proof that the algorithm terminates and computes M_i correctly is identical to the proof of Theorem 3.4. It remains to be shown that the labels are correctly computed. Since the graph is absorptive, we need only show that AGG is applied exactly once to the label of every acyclic path from i to j in computing L_{ij}, for all ordered pairs (i, j). Whenever a new path from i to j is generated, AGG is applied to the label of this path and the previous label L_{ij}. Thus, since CON is distributive over AGG, we only need to show that all acyclic paths are generated, and generated exactly once.

The algorithm generates a path $P = (i, k, \ldots, j)$ by first generating the subpath from k to j and then concatenating this path to the arc $E(i,k)$. This is done during the call *visit*$1(i)$. The numbering *popped*$[i]$ specifies the order in which calls to *visit*1 terminate, that is, *popped*$[i]$ < *popped*$[j]$ if and only if *visit*$1(i)$ terminates before *visit*$2(j)$.

*Claim*1: For all nodes i in G, all acyclic paths from i are generated (1) when *visit*$1(i)$ terminates, if i does not belong to a nontrivial strongly connected component, or (2) when *visit*$2(r)$ terminates, where r is the root of the strongly connected component containing i, otherwise.

We prove this by induction on the condensation graph of G. A number *popped* is also associated with each node of the condensation graph. If node i represents a nontrivial strongly connected component of G with root r, we define *popped*$[i]$ = *popped*$[r]$. For all other nodes, which represent themselves, *popped* is as before. Our induction is on the ordering *popped*. Consider the basis case of node a such that *popped*$[a]$ is the least over all nodes in the condensation graph. (It may not be 1 - this node may represent a connected component in G which contains, or is reachable from a connected component which contains, the node in G with *popped* = 1.) If this node is a leaf in G, the claim holds trivially. Suppose this node represents a strongly connected component in G. Then we show that the claim holds for all nodes in this strongly connected component. Node a must be a leaf in the condensation graph, and so no nontrivial connected component in G is reachable from a. Let r be the root of this component in G.

*Claim*2: Consider a node n, $n \neq r$, in this connected component. When *visit*$1(n)$ terminates, every acyclic path from n which does not contain a back arc has been generated and considered in the label computation for labels of the form L_{nk}, for any node k that is reachable from n by such a path.

The proof of Claim 2 is by induction on the height of the spanning tree rooted at r which is induced by the calls to *visit*1. This proof is straightforward, and is omitted here.

Claim 3: When the call $visit(i)$ is generated for a node i in this component, if $j \in T_i$, then all acyclic paths from j have been generated.

We prove this by induction on the numbering $visited[i]$, for the nodes in this component. For the basis, $visited[r] < visited[j]$ for all nodes j in this component. Consider the call $visit2(r)$, which is invoked when we identify the root r (during the call $visit1(r)$). We note that $T_r = \phi$, and so Claim 3 holds trivially.

Let the claim hold for all nodes j in this component with $visited[j] <$ N, N $> visited[r]$. Consider a node n such that $visited[k] =$ N+1. When the call $visit2(n)$ is generated, if $m \in T_n$, then the call $visit2(n)$ has already been generated. This follows from the fact that calls to $visit2$ are generated in the same order as calls to $visit1$, and from Lemma 5.1. Thus, all acyclic paths from m have been generated, and their labels computed. Further, since $visit1(n)$ has terminated, all acyclic paths from n that do not contain a back arc have been generated, by Claim 2. The only other acyclic paths from n are those which are of the form $(n, m, \ldots)$, where $m \in T_n$. Since all acyclic paths from m have already been generated, $visit2$ generates all the remaining acyclic paths from n by concatenation of paths $(m, \ldots)$ to $E(n,m)$. This completes the proof of Claim 3 and also the proof of the basis case for Claim 1.

Let Claim 1 hold for all nodes i in the condensation graph with $popped[i] \leq$ N. Consider a node j with $popped[j] =$ N+1. If j does not belong to a nontrivial connected component, then, for every child k of j, all acyclic paths from k are generated when $visit1(k)$ terminates. Thus, $visit1(j)$ generates all acyclic paths from j. If j belongs to a connected component, the proof is similar to the proof of the basis case, and is omitted. This concludes the proof of Claim 1.

Claim 4: Each (acyclic) path is generated at most once.

Consider a path from node i to node j, $(i, k, \ldots, j)$. If the path does not involve back arcs, it is (only) generated by concatenating $(k, \ldots, j)$ to $E(i,k)$. Thus, it is generated only once. Let the path involve a back arc. Then we can denote the path as $P = (i, k, \ldots, n) \,.\, (n, m) \,.\, (m, \ldots, j)$, where the segment from i to n does not involve any back arcs, and $E(n,m)$ is a back arc. Each acyclic subpath from m is concatenated to the path from i to m exactly once, and further, this is the only way P is generated. This concludes the proof of Claim 4, and of Theorem 5.2. □

Consider the labeled graph of Figure 5.1. We illustrate the algorithm by using it to find the shortest path between all pairs of points. Note that AGG is *min* and CON is *add* for this problem.

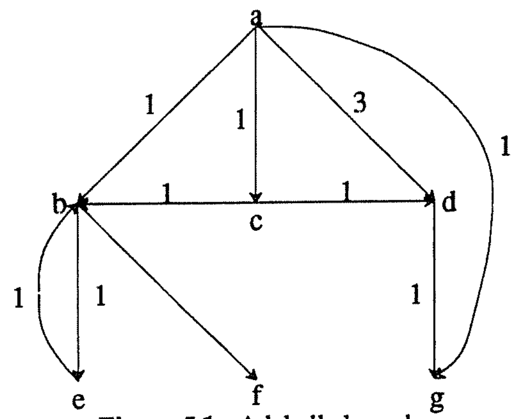

Figure 5.1: A labelled graph.

We assume that the visit order is (a, c, b, e, f, d, g). Thus, the call *visit* $1(a)$ recursively generates (in order) the calls *visit* $1(c)$, *visit* $1(b)$, *visit* $1(e)$, *visit* $1(f)$, *visit* $1(d)$, and *visit* $1(g)$. The induced spanning tree contains the arcs $E(a,c)$, $E(c,b)$, $E(b,e)$, $E(b,f)$, $E(c,d)$ and $E(d,g)$.

The following table indicates the marked and tagged sets, with the labels for the corresponding paths included, for node i after the call *visit* $1(i)$. The rows are in the order that the calls terminate:

Node i	M_i	T_i
e	Φ	{ <b,1> }
f	Φ	Φ
b	{ <e,1>, <b,2>, <f,1> }	Φ
g	Φ	Φ
d	{ <g,1> }	Φ
c	{ <b,1>, <e,2>, <f,2>, <d,1>, <g,2> }	Φ
a	{ <b,1>, <e,2>, <f,2>, <d,2>, <g,1> }	Φ

We identify b as the root of a connected component after the calls *visit* (e) and *visit* (f) terminate. We then invoke *visit* $2(b)$, which recursively invokes *visit* (e). After this call terminates, M_e = { <b,1>, <e,2>, <f,2> }, and $T_e = \Phi$. The call *visit* $2(b)$ then terminates, and thus, *visit* $1(b)$ terminates.

Let us consider the call *visit* $1(a)$ at the point when the call *visit* $1(c)$ has terminated. The successor sets for all the other nodes are as shown in the table, and the successor set of a is empty. E_a = { <b,1>, <c,1>, <d,3>, <g,1> }. We now generate all paths from a through c by concatenating the successor set for c to the arc $E(a,c)$. After this step, M_a = { <c,1>, <e,3>, <f,3>, <d,2> } and $T_a = \Phi$. Note that b and g are not in M_a at this point although they are in M_c, since their labels do not change. Next, we generate all paths from a through b by concatenating the successor set of b to the arc $E(a,b)$. After this step, M_a = { <b,1>, <c,1>, <e,2>, <f,2>, <d,2> } and $T_a = \Phi$. Notice that the labels of some of the nodes already in M_a changed in this step. Finally, we consider the arc $E(a,g)$, and since the successor set of g is empty, we terminate after moving g to the marked set, as shown in the last row of the table. []

Note that we could use the following heuristic in deciding which successor set to add next:

choose j s.t. $L_{ij} \leq L_{ik}$ for all $k \in U_i$

instead of the heuristic of choosing the node with the largest *popped* value. However, we could simply pick some arbitrary node j without affecting the correctness of the algorithm.

A few additional remarks may be in order. First, it is possible to ask for path set labels for an arbitrary set of paths. The above algorithm only computes labels for sets of all paths from some node i to some node j. However, by subsequently applying AGG to the set of all labels, it also possible to compute the path set label for the set of all paths in the transitive closure. Thus, we can find the longest path in a graph. (In this case, CON would be concatenation and AGG would be the function which picks the longer path.)

Second, it is possible to specify aggregate queries with selections (of the form "column = c" etc.). For example, we may wish to compute the longest path from node i. Thus, we are only interested in the set of paths from node i. This is computed using essentially the algorithm for the selection "column1 = i", with some label computation, followed by applying AGG to all labels in the successor set S_i. We present this algorithm below.

```
**proc** select_path_TC ( $i$ )
{ $U_i := E_i$; $M_i := \phi$
 **while** $U_i \neq \phi$ **do**
      choose $j \in U_i$ s.t. $L_{ij} \leq L_{ik}$ for all $k \in U_i$;
      **for** all $k \in S_j$ **do**
           $L_{ik}^{old} := L_{ik}$; $L_{ik} :=$ AGG ($L_{ik}$, CON ($L_{ij}, L_{jk}$));
           **if** $L_{ik}^{old} \neq L_{ik}$ **then** { $U_i := U_i \cup \{ k \}$; $M_i := M_i - \{ k \}$ }
      **od**
      $U_i := U_i - \{ j \}$; $M_i := M_i \cup \{ k \}$
 **od**
}
```

We observe that for the special case of the single source shortest path problem, this is Dijkstra's algorithm [Dijkstra 59]. Other examples of path computations with selection include the shortest path from node i to node j, the maximum flow path from node i to node j, etc. At first sight, it appears that we can use the algorithm corresponding to the selection "column1 = i *and* column 2 = j". However, we would then terminate as soon as we discovered some path from node i to node j. This is incorrect since we wish to find *all* paths from i to j. Thus, we must still use the algorithm corresponding to the selection "column1 = i".

6.2. Analysis

One can see that DFTC and Path_DFTC are different only in the way they treat the various successor lists. DFTC simply adds lists to others while removing duplicates. Path_DFTC adds lists also, but whenever there are duplicate nodes, it chooses exactly one of them, according to the semantics of AGG. The complexity of the two operations is exactly the same. Hence, the (worst-case) complexity of both algorithms (reachability and path computations) is the same. It is repeated here for clarity.

$$\text{cpu}(\text{path\_DFTC}) = O(n + e + \sum_{c \in SCC} e^{out_c} + [e + \sum_{(v,w) \in E} t_w + \sum_{c \in SCC} n_c t_c]). \tag{9}$$

$$\text{i\_o(path\_DFTC)} = O(n + e + \sum_{c \in SCC} (n_c + e_c) + [e + \sum_{c \in SCC} n_c]). \quad (10)$$

7. One-Sided Recursions

We can extend the algorithm to deal with recursive queries that are extensions to transitive closure called *one-sided recursions*, which were introduced by Naughton [Naughton 87]. The following example illustrates this.

```
t(X,Y) :- s(X,Z), a(Z,Y).
a(X,Y) :- p(X,Z), a(Z,Y).
a(X,Y) :- q(X,Y).
```

The relation t can be expressed as $s \cdot p^* \cdot q$. If we wish to compute the entire relation t, we proceed as follows. Suppose s contains a tuple ($c1$,$c2$). Then, we compute p^* with the selection "column1 = $c2$". We do this for every such tuple in s. Once, we compute the closure for some node, we can make use of this in computing the closure of other nodes (through closed additions). To maximize this, we could order the values in the second column of s by the frequency of their occurrence in successor lists of p. Finally, we join all the successor sets computed in this fashion with q. (We will see later how this algorithm can be arrived at in the general context of a one-sided recursion.)

If we wish to compute t with a selection of the form "column1 = c", we proceed as follows. We start with the successor set of c in relation t. Next we extend this set using the transitive closure algorithm. This gives us $\sigma(s) \cdot p^*$. Finally, we join with q. We deal with a selection of the form "column2 = c" similarly, but we must first construct predecessor sets.

We now sketch the extension of this evaluation method to general one-sided recursions. Our presentation is informal, and we refer the reader to [Naughton 87] for formal definitions. Consider a predicate p defined by a Datalog program that contains a single, linear recursive rule. By substituting the body of the rule for the occurrence of the recursive predicate, we can produce a series of strings of predicate instances. >From each string of predicate instances in this series, we can generate a string of predicate instances containing only base predicates (predicates which are stored in the database and are not defined by any rules) by using the non-recursive rules. The answer to a query can be obtained by generating these strings of predicate instances and evaluating the resulting join expression over the base predicates. To do this efficiently, we must try to combine the effort of generating (and evaluating) successive strings in the series. In the above example, the series contained the following strings:

$s(X,Z),\ a(Z,Y)$
$s(X,Z),\ p(Z,Z1),\ a(Z1,Y)$
$s(X,Z),\ p(Z,Z1),\ p(Z1,Z2),\ a(Z2,Y)$
...

Using the non-recursive rule, these sets generate the following join expressions which must be evaluated over the base predicates:

$s(X,Z),\ q(Z,Y)$
$s(X,Z),\ p(Z,Z1),\ q(Z1,Y)$
$s(X,Z),\ p(Z,Z1),\ p(Z1,Z2),\ q(Z2,Y)$
...

Informally, a definition is one-sided if there is exactly one connected set in each string of the series after removing the instance of the recursive predicate. (We say that a predicate instance p is *connected* to another predicate instance q in a string if they share a common variable, or if there is a predicate instance m which is connected to both p and q. A *connected set* is a maximal set of connected predicate instances.)

Naughton has observed the following. If a recursion is one-sided, the join expressions to be evaluated are all of the form $s \,.\, p^i \,.\, t$, where:

1. s is a (possibly empty) initial segment of predicate instances.
2. p^i denotes i repetitions of p, where p represents a group of predicate instances and $i \geq 0$. The adjacent instances of p are connected by shared variables, and the pattern of shared variables is the same between each pair of adjacent instances of p.
3. t represents some predicate instances due to the non-recursive rules.

This regular pattern in the series of expressions to be evaluated allows us to develop efficient evaluation algorithms. The bulk of the work is in evaluating the joins denoted by the subexpression p^i. Naughton presents an efficient algorithm for evaluating queries containing selections. This algorithm works by evaluating each expression in the series by starting from the end which contains the selection and performing a series of joins. Further, the regular structure of the expressions implies that in evaluating each expression we can re-use the results obtained in evaluating the previous expression. The algorithm is presented below:

```
initialize $carry$, $seen$, $ans$;
while $carry$ not empty do
        $carry := f(carry)$; $seen := seen \cup carry$; $carry := carry - seen$
od
$ans := g(seen)$
```

The initialization of the sets *carry*, *seen* and *ans*, and the functions f and g are determined by the query and the program. We note that this has a strong relationship to the selection algorithm presented earlier for transitive closure. To make this relationship more apparent, we present below a slight variant of algorithm *Basic_TC*.

```
proc Basic_TC2 ( $G$ )
Input: A digraph specified as a binary relation $G$.
Output: $G^*$ represented as a binary relation $ans$.
{ $U_i := M_i := \pi_2(\sigma_{1=i}(G))$; $T := \phi$;
  for $i = 1$ to n do
        while $U_i \neq \phi$ do
              $U_i := U_i \bowtie G$; $M_i := M_i \cup \pi_2(U_i \bowtie T) \cup U_i$; $U_i := U_i - M_i$
        od
        $T := T \cup (\{ i \} \times M_i)$;
  od
  $ans := T$
}
```

This is essentially algorithm *Basic_TC* (using a different representation for the relations) except that the choice of $j \in U_i$ is constrained a little. (Note that any execution of *Basic_TC*2 is also an execution of *Basic_TC*, but the converse is not true. Some slight changes are also introduced because Naughton initializes *seen* to *carry*, whereas our earlier algorithm initialized it to ϕ.) This leads naturally to the following selection algorithm:

```
proc Select ( $G$ )
Input: A digraph specified as a binary relation $G$.
Output: $\sigma_{1=i}\ (G^*)$, represented as a binary relation $T$.
{ $U_i := M_i := \pi_2 ( \sigma_{1=i}\ (G)); T := \phi$;
 while $U_i \neq \phi$ do
      $U_i := U_i \bowtie G; M_i := M_i \cup \pi_2 (U_i \bowtie T) \cup U_i;\ U_i := U_i - M_i$
 od
 $T := (\{ i \} \times M_i)$;
 $ans := T$
}
```

The statement in the while-loop can be simplified a little by noticing that $T = \phi$ when it is executed:

$U_i := U_i \bowtie G; M_i := M_i \cup U_i;\ U_i := U_i - M_i$

To see the similarity to Naughton's algorithm, observe that U_i and M_i correspond, respectively, to *carry* and *seen*. Suitably generalizing algorithm *Select* by introducing abstract functions f and g in the place of the expressions defining U_i and T (which are specific to the given query on the transitive closure program), and generalizing the initialization statements, we obtain precisely Naughton's algorithm for evaluating selection queries.

Generalizing *Basic_TC*2 similarly in terms of f and g, we obtain an algorithm for computing a query with no selections over a relation defined using one-sided recursion. In the example we considered earlier in this section, the algorithm obtained in this way is indeed the algorithm informally presented earlier. To illustrate this point, we consider that example again:

```
t(X,Y) :- s(X,Z), a(Z,Y).
a(X,Y) :- p(X,Z), a(Z,Y).
a(X,Y) :- q(X,Y).
```

The initialization step is as follows:

$U_i := M_i := \pi_2 (\sigma_{1=i}\ (s));$
$T := \phi$

The statement in the while-loop is as follows:

$U_i := U_i \bowtie p; M_i := M_i \cup \pi_2 (U_i \bowtie T) \cup U_i;\ U_i := U_i - M_i$

Finally, the definition of *ans* is as follows:

$ans := \pi_{1,3} (T \bowtie q)$

This concludes our discussion of one-sided recursion. We hope that we have conveyed the intuition behind the generalization of the transitive closure algorithm to the case of one-sided recursion, although we have not described how to automatically arrive at the appropriate initializations and definitions of the functions f and g. While we have not presented an analysis of this algorithm in comparison to Seminaive (in particular, in computing the entire relation without selections), we observe that this is analogous to the performance of *Basic_TC* with respect to Seminaive for the reachability case of transitive closure. Thus, we expect a significant improvement on the average.

8. 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 application, 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.

8.1. Schmitz

In all the relevant literature, the algorithm by Schmitz [Schimtz 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]. 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 the other members of the component as well. With this modification the cost of Schmitz's algorithm is

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

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,

- Schimtz'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, an additional $O(\sum_{c \in SCC} n_c)$ lists have to be brought in during this phase.

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

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

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 a factor of at least 1/3.

8.2. 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.

```
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 ( $G$ ) {

$U_i := E_i; M_i := \Phi$
for $i := 1$ to $n$ do
    while there is $j \in U_i - \{i\}$ do $M_i := M_i \cup \{j\}$; $U_i := U_i \cup E_i - M_i$ 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, howev , 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:

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

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 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:

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

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.

† 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, focussing 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.

8.3. 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 $j \in S_i$ then $S_i := S_i \cup S_j$; od
od
for $i := 1$ to $n$ do
    for $j := i+1$ to $n$ do if $j \in S_i$ then $S_i := S_i \cup S_j$; 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

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

Comparing even against (13), (15) makes the Warren algorithm look even worse than Seminaive, let alone Global_TC. 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

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

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_TC for the acyclic case. Although this is simply an indication and not a proof, it seems that Global_TC 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:

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

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

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 ($n+e$ versus $n+2e$). 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. However, there is 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.

8.4. Other Work

Besides Seminaive, another popular algorithm that has been proposed for general recursion is the *Smart* or *Logarithmic* algorithm [Valduriez and Boral 86] and [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.

A limited amount of work has been done on the shortest path problem between two given nodes of a graph using QUEL* as the coding language [Kung et al. 86]. Four algorithms, two based on breadth-first search, and two heuristic algorithms, based on best first search [Rich 83], were implemented and their performance was compared against the same programs implemented in Fortran. The conclusion was that for large graphs, one of the breadth-first versions was the algorithm of choice, even as compared to a main memory implementation. Both the scope of that work and its conclusions are only marginally relevant to the work presented in this paper.

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).

9. 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.

A preliminary study of the potential for parallel execution and performance improvements through good blocking and paging heuristics shows much promise. However, as we progress from *Basic_TC* to the more sophisticated algorithms, it appears that there is a tradeoff between these factors and the benefits of a strictly depth-first search order.

Finally, if the graph G is updated relatively infrequently, it might be feasible to store the successor sets according to (at least, approximately) the reverse topological order (*popped*). In this case, directly running algorithm *Basic_TC* to compute the transitive closure should outperform the other strategies, and thus, the order in which data is stored is exploited in a unique way to optimize the computation of the transitive closure.

10. 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.

[Hecht 77]
Hecht, M. S., *Flow Analysis of Computer Programs*, Computer Science Library, Elsevier North-Holland, New York, NY, 1977

[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.

[Kung et al. 86]
Kung, R. et al., "Heuristic Search in Database Systems", in *Expert Database Systems, Proc. from the 1st International Workshop*, L. Kerschberg (ed.), Benjamin-Cummings, Menlo Park, CA, 1986, pp. 537-548.

[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.

[Rich 83]
Rich., E., *Artificial Intelligence*, McGraw-Hill, New York, NY, 1983.

[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.

[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.