Transitive Closure Algorithms Based on Graph Traversal

YANNIS IOANNIDIS, RAGHU RAMAKRISHNAN, AND LINDA WINGER
University of Wisconsin

Several graph-based algorithms have been proposed in the literature to compute the transitive closure of a directed graph. We develop two new algorithms (Basic_TC and Global_DFTC) and compare the performance of their implementations in a disk-based environment with a well-known graph-based algorithm proposed by Schmitz. Our algorithms use depth-first search to traverse a graph and a technique called *marking* to avoid processing some of the arcs in the graph. They compute the closure by processing nodes in reverse topological order, building descendent sets by adding the descendent sets of children While the details of these algorithms differ considerably, one important difference among them is the time at which descendent set additions are performed Basic_TC performs a separate depth-first traversal to obtain the topological order of nodes and does additions in a second pass. Global_DFTC performs additions whenever two sets that must be added are in memory, thereby eliminating the need to bring these sets in again later. The Schmitz algorithm is intermediate in this respect, deferring the addition of the descendent set of a child to that of a parent until the root of the strong component containing the parent is identified. Contrary to our expectations, deferring additions as much as possible, as in Basic_TC, results in superior performance. The first reason is that early additions result in larger descendent set sizes on the average over the duration of the execution, thereby causing more I/O; very often this turns out to more than offset the gains of not having to fetch certain sets again to add them The second reason is that information collected in the first pass can be used to apply several optimizations in the second pass. To the extent possible, we also adapt these algorithms to perform path computations. Again, our performance comparison confirms the trends seen in reachability queries Taken in conjunction with another performance study our results indicate that all graph-based algorithms significantly outperform other types of algorithms such as Seminaive and Warren.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*recursion*; D.4.2 [**Operating Systems**]: Storage Management—*main memory, secondary storage, swapping*; E.1 [**Data**]: Data Structures—*graphs, trees*; H.2.4 [**Database**

Some of the results in this paper appeared in a preliminary form in "Efficient Transitive Closure Algorithms," in *Proceedings of the 14th International VLDB Conference* (Long Beach, Calif., Aug. 1988). With respect to that paper, most of the algorithms have been revised, and the performance analysis has been replaced by the results of an implementation-based performance evaluation. Y. Ioannidis was partially supported by the National Science Foundation under grant IRI-8703592 and a grant from IBM. R. Ramakrishan was partially supported by the National Science Foundation under grant IRI-8804319 and a Presidential Young Investigator Award, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a grant from IBM, and an IBM Faculty Development Award.
Authors' address: Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

Management]: Systems—*query processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Depth-first search, node reachability, path computations, transitive closure

1. INTRODUCTION

Several transitive closure algorithms have been presented in the literature. These include the Warshall and Warren algorithms [28, 29], which use a bit-matrix representation of the graph, the Schmitz [25], the Ebert [10] and the Eve and Kurki-Suonio algorithms [11], which use Tarjan's algorithm [26] to identify strong components in reverse topological order, the Seminaive [5] and Smart/Logarithmic algorithms [12, 27], which view the graph as a binary relation and compute the transitive closure by a series of relational joins, and recently, a hybrid algorithm combining matrix-based algorithms and graph-based algorithms [1]. We develop two new algorithms based on depth-first traversal, and compare their performance in a disk-based environment with the well-known graph-based algorithm proposed by Schmitz. Basic_TC is the simplest of our algorithms and consists of a first pass that yields a topological sort of the nodes in the graph, and a second pass that iteratively processes nodes in reverse topological order and builds their descendent sets by adding the descendent sets of children. Global_DFTC is the second of our algorithms and seeks to combine the two passes of Basic_TC by adding two descendent sets that must be added whenever they are simultaneously in memory during the first pass, instead of waiting until the second pass to do so. Hereafter, we refer to these algorithms as BTC and GDFTC, respectively. Specialized versions of the algorithms that are applicable on acyclic graphs only are named Dag_BTC and Dag_DFTC, respectively, where the prefix "Dag_" stands for "Directed acyclic graph".

We have implemented several versions of our algorithms and the Schmitz algorithm, and compared their performance over randomly generated graphs. The result of this comparison is rather surprising, given the following characteristic difference among the algorithms: GDFTC performs additions as soon as possible, BTC performs them as late as possible, and Schmitz performs them at an intermediate stage. Counter to the intuition that early additions are better (since descendent sets that have been added together need not be brought back into memory for this addition later), BTC outperforms both Schmitz and GDFTC. The first reason is that early additions result in working with larger descendent sets for a longer time during the execution of the algorithm, and this results in buffers being filled up quicker, thereby leading to more I/O. Overall, the two effects—avoiding extra retrievals for additions and faster growth of sets—appear to balance out, with the faster growth of sets being perhaps a little more dominant. The second reason is that information collected in the first pass can be used to apply several optimizations in the second pass.

We have adapted BTC 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. We have also adapted GDFTC and Schmitz to compute such path queries, but only for acyclic graphs. (For these queries, GDFTC and Schmitz are applicable only for acyclic graphs since they do not maintain path information within a strong component. Indeed, an important optimization of BTC, which has the effect of merging nodes in a strong component for reachability, is also inapplicable.) We compare the performance of BTC, GDFTC, and Schmitz for path queries over acyclic graphs, and show that the results for reachability extend to this case as well. We also present a comparison of the various versions of BTC for path computations on cyclic graphs.

This paper differs in many respects from its preliminary version [13]. First, the algorithms have been revised and presented differently, and full proofs of correctness have been included. Second, we have increased the emphasis on BTC due to its simplicity and superior performance.[1] The most important difference, however, is that the analysis presented in the preliminary version has been replaced by a performance evaluation based upon actual implementations of the algorithms. Indeed, this has caused us to revise some of our conclusions about the relative merits of the algorithms. The performance evaluation brought out the fact that the algorithms were affected significantly by the impact on buffer management of the growth of descendent sets. This was not reflected in our analysis, which made the assumption that all strategies were affected similarly (due to the assumption that only "minimal" buffer space was available). Thus, the analysis was for worst-case performance, whereas based upon our implementation and performance evaluation, the behavior for the average case differs considerably. Also, by implementing the algorithms, we were able to experiment with specialized data organizations, which could not be captured in the analysis.

The paper is organized as follows. We introduce some notation and present a summary of the new and the existing graph-based algorithms in Section 2. Section 3 presents the new algorithms in detail, starting with some simple versions and subsequently refining them. We describe the implementation of the algorithms in Section 4, and the testbed for performance evaluation in Section 5. We present a performance comparison for reachability queries in Section 6 (acyclic graphs) and Section 7 (cyclic graphs). Path queries are considered in Section 8, and the algorithms presented earlier are adapted to compute them. In Section 9, we present a performance comparison of the algorithms for path queries. We discuss selection queries in Section 10. Graph-based algorithms are compared to nongraph-based ones in Section 11. Finally, our conclusions are presented in Section 12.

[1] In fact, we have deleted one of the algorithms in that paper, called DFTC. The algorithm, which we did implement for reachability queries, performed uniformly worse than BTC and GDFTC, and did not present any points of additional interest.

2. GRAPH-BASED ALGORITHMS

A large body of literature exists for main-memory based algorithms for transitive closure. Recently, with the realization of the importance of recursion in new database applications, transitive closure has been revisited and reexamined in a data intensive environment. In this section, we concentrate on graph-based algorithms, i.e., ones that take into account the graph structure and its properties and compute the transitive closure by traversing the graph. Almost all such algorithms have the following common characteristics: (a) they are based on a depth-first traversal of the graph, (b) they identify the strong components of the graph using Tarjan's algorithm [26], 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. Based on (c), graph-based algorithms can compute the transitive closure of a graph so that only a pointer is associated with each node in a strong component pointing to a common descendent set. In this section, we discuss graph-based algorithms by Purdom [20], Ebert [10], Schmitz [25], and Eve and Kurki-Suonio [11] and compare them with our algorithms. The focus is on algorithms that have been proposed for reachability computation of entire graphs, and so graph-based algorithms that have been primarily proposed for partial transitive closure, e.g., [14], or path computations are not being discussed. We first present some notation and basic definitions.

2.1 Notation and Basic Definitions

In this paper we use the term *graph* to refer to a directed graph, since we do not discuss undirected ones at all. We assume that the graph G is specified as follows: for each node i in the graph, there is a set of *children* $E_i = \{j | (i, j)$ is an arc of $G\}$. Without loss of generality, we assume that G has no self-loops, i.e., for all nodes $i, i \notin E_i$. For an arc (i, j), node i is called the *source* or *tail* and node j is called the *destination* or *head* of the arc. We denote the transitive closure of a graph G by G^*. The children of i in G^* are the *descendents* of i in G. The *strongly connected component* (or *strong component*) of node i is defined as $V_i = \{i\} \cup \{j | (i, j) \in G^*$ and $(j, i) \in G^*)\}$. The component V is *nontrivial* if $V_i \neq \{i\}$. The *condensation graph* of G, G_{con}, has the strong 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 set of descendents in the transitive closure for a node i is $S_i = \{j | (i, j)$ is an arc of $G^*\}$.

As mentioned in point (a) above, most graph-based algorithms perform a depth-first traversal of graphs, so we review some definitions relevant to it. Depth-first traversal induces a spanning forest on the graph based on the order in which nodes are visited. If we assume that the main routine in depth-first traversal is visit(i) for a node i, then 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). An arc (i, j) in the graph G is called a *tree arc*, if it belongs in the spanning forest. An arc (i, j) in the graph G but not in the spanning forest is called a *forward arc*, a *back arc*, or a *cross arc*, if in the spanning forest, j is

a descendant of i, j is an ancestor of i, or j is not related to i with an ancestor-descendant relationship, respectively. For every strong component, its node r on which visit(r) is first called is the *root* of the strong component.

2.2 Summary of Algorithms

The goal of this subsection is not to present any algorithm in detail but rather to give an abstract description so that the main differences among them and their implications on performance can be understood. Algorithms BTC and GDFTC are described in detail in later sections of the paper. Detailed expositions of the remaining algorithms can be found by the interested reader in the original references. In the descriptions that follow, one should pay special attention to the fact that BTC and GDFTC are two extreme points in a spectrum of possibilities for when descendent sets of children are added to those of parents, with Schmitz somewhere in the middle. This is important because it allows the conclusions of a performance evaluation of these three algorithms to be used as a basis for a qualitative understanding for how the other graph-based algorithms are likely to perform as well.

BTC. This algorithm uses Tarjan's algorithm as a first pass to construct a topological ordering of nodes and to identify the strong components of the graph. Additionally, that pass can be used to physically cluster the relation in reverse topological order with nodes in descendent sets arranged in topological order. This improves the performance of a second pass when the descendents of all nodes are found in reverse topological order. An optimization called "marking" is used to avoid the addition of a descendent set in many cases where earlier set additions are guaranteed to have added all nodes in the given descendent set. Because of the two-pass structure of the algorithm, set additions are deferred as much as possible.

GDFTC. This algorithm defines the opposite end of the spectrum from BTC in that descendent set additions are performed as early as possible. When returning from a child along a tree arc or an intercomponent cross-arc, the child's descendent set is added to the parent's set immediately, thereby eliminating the need to retrieve these sets subsequently to perform this addition. A rather complicated stack mechanism ensures that, for all nodes in a strong component, when returning from a child (along a forward or intercomponent cross-arc), the child's descendent set is added to the set of a *representative* of the strong component. Thus, additions are never deferred.

Purdom. Purdom proposed an algorithm that is similar to BTC [20]. It is based upon computing a topological sort of the condensation graph prior to computing the closure. The main difference with respect to BTC is the absence of marking; the implementation of BTC also incorporates some important optimizations that increase the effectiveness of marking and take advantage of the topological sort for physical clustering.

Eve and Kurki-Suonio. Eve and Kurki-Suonio observed that on returning to a node i after processing a child j, node j is still on the stack if and only if i and j are in the same strong component [11]. Further, after processing all the children of the root of a strong component, the nodes on the stack that are above the root comprise the nodes in the strong component. They proposed the following modifications to Tarjan's algorithm in order to compute the transitive closure. First, if i and j are in different strong components, the descendents of j are added to the descendents of i after visiting j from i (similarly to GDFTC). Second, when the root of a strong component is identified, the descendents of each node in the strong component are added to the descendents of the root (similarly to Schmitz). There are two potential redundancies in the algorithm that affect performance. First, the algorithm propagates descendent sets even when returning from forward arcs although this is unnecessary. Second, if there is an arc (j, k) such that j is in a nontrivial strong component and k is in a different component, k is added to S_j, by the first modification above, and also to the descendent set constructed for the root of j's strong component, via the addition of S_j, by the second modification above.

Ebert. Ebert suggested another modification of Tarjan's algorithm: a depth-first traversal of the graph is performed to identify strong components, but when returning from a child, if the arc is a tree arc or an intercomponent cross arc, the descendents of the child are added to the descendents of the parent [10]. This algorithm improves upon Eve and Kurki-Suonio by performing no additions on forward arcs. For acyclic graphs, the Ebert algorithm is identical to Dag_DFTC. For cyclic graphs, however, there are many redundant operations in Ebert in that descendent sets are propagated after returning from every tree arc in the component until they are eventually propagated to the descendent set of the root.

Schmitz. Schmitz's modification of Tarjan's algorithm is based upon the fact that strong components are identified in reverse topological order, and that all nodes in the strong component are on the stack above the root node when the root is identified [25]. The modification to compute the transitive closure is essentially to construct the descendent set of the root by adding the descendent sets of *all children* of nodes in the strong component when the root is identified. Schmitz's algorithm also detects forward arcs and ignores them; thus, the first redundancy of Eve and Kurki-Suonio's algorithm is avoided. Finally, Schmitz uses an optimization that is similar to marking over the condensation graph, although it is not in general as flexible as marking in BTC, due to the two-pass nature of BTC versus Schmitz's one-pass structure. Further, since no additions are made to the descendent sets of nodes in a strong component until the root is identified, the second redundancy of that algorithm is also avoided as well as the redundancy of Ebert's algorithm. Like Eve and Kurki-Suonio's algorithm, however, Schmitz has the potential cost of retrieving descendent sets that may not be in memory. For Schmitz, these are sets of children of nodes on the stack above the root node; for Eve and Kurki-Suonio, these are sets of nodes on the stack

above the root. Thus, both algorithms are intermediate between GDFTC and BTC in terms of when descendent sets are added: additions are not done eagerly, but they are not deferred to a second pass either. Instead, they are deferred until the entire strong component containing the node is identified.

Schmitz also proposed a variant of his algorithm in which what he refers to as an *arc basis* of the graph is computed and used, i.e., a minimal subset of arcs such that their transitive closure is equal to that of the original graph. We have not explicitly studied this variant. Nevertheless, one of the techniques used in the implementation of BTC achieves essentially the same effect. Thus, an upper bound on the cost improvement of this variant over the basic Schmitz algorithm can still be derived indirectly (Section 6.1).

2.3 Comparison of Algorithms

BTC can be seen as a refinement of Purdom's seminal algorithm. Both the marking optimization and the physical clustering that we propose are seen to yield significant improvements; BTC clearly dominates Purdom's algorithm.

The Eve and Kurki-Suonio algorithm performs all the additions performed by the Schmitz algorithm and usually more, and it is almost identical to it in terms of when additions are carried out. We therefore expect that the Schmitz algorithm is uniformly superior. (A comparison by Schmitz in terms of vector operations and run-times for a main memory implementation corroborates this observation.)

The Ebert algorithm is identical to Dag_DFTC (and thus GDFTC) for acyclic graphs. For cyclic graphs, it does strictly more work than GDFTC in terms of additions. While it is not clear how Ebert and GDFTC compare in terms of CPU time for cyclic graphs (since GDFTC uses more complex stack operations), we expect that the I/O performance of GDFTC is uniformly better than that of Ebert.

Based on the above observations, we have limited ourselves to a comprehensive comparison of the performance of BTC, GDFTC, and Schmitz only. The detailed results of this comparison are presented in Sections 6 and 7.

3. THE NEW TRANSITIVE CLOSURE ALGORITHMS

In this section we present in detail several new transitive closure algorithms based upon depth-first graph traversal.

3.1 A Marking Algorithm

We first present a simple transitive closure algorithm that introduces a technique called *marking*. Intuitively, if a descendent set contains a marked node, it also contains the children (but not necessarily all descendents) of that node. In the following, descendent set S_i is partitioned in two sets M_i and U_i that can be thought of as the *marked* and *unmarked* subsets of S_i.

proc Closure (G)
Input: A graph G represents by children sets E_i, $i = 1$ to n.

Output: $S_i = M_i$ ($U_i = \emptyset$), $i = 1$ to n, denoting G^*.

(1) {**for** $i = 1$ to n **do** $U_i := E_i$; $M_i := \emptyset$ **od**
(2) **for** $i = 1$ to n **do**
(3) **while** there is a node $j \in U_i - \{i\}$
do $M_i := M_i \cup M_j \cup \{j\}$; $U_i := U_i \cup U_j - M_i$ **od**
(4) **od**}

LEMMA 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 $M_i = \emptyset$ and $U_i = E_i$, and that for all i, $M_i \cup U_i$ is monotonically increasing. □

THEOREM 3.2. *Algorithm Closure correctly computes the transitive closure of a graph G.*

PROOF. If $j \in M_i \cup U_i$, then $j \in E_i$ or there is some node k such that $k \in E_i$ and $j \in M_k \cup U_k$. 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 when the algorithm terminates, for all i, $U_i = \emptyset$. The proof is completed by noting that initially $U_i = E_i$ and that $M_i \cup U_i$ is monotonically increasing for all i, and by applying Lemma 3.1. □

We note that Schmitz's algorithm contains an optimization that achieves the effect of marking over the condensation graph. (For the interested reader, this is the optimization that derives from Lemma 2 of [25].)

3.2 Depth-First Traversal to Number Nodes

In the algorithms that we introduce in the following sections, we need to obtain a numbering of the graph nodes with the property that all descendants of a node numbered m have a lower number than m, i.e., a topological order numbering. In the presence of cycles, an approximation to such a numbering is obtained, by ignoring back arcs. That is, in the acyclic graph obtained from G by ignoring back arcs, all descendants of a node numbered m have a lower number than m. The depth-first numbering algorithm is presented below.

proc Number (G)
Input: A graph G represented by children sets E_i, $i = 1$ to n.
Output: Graph G with nodes numbered. The numbering is stored in a global array *popped*[ ].

(1) {*vis* = 1;
(2) **for** $i = 1$ to n **do** *visited*[i] := 0; *popped*[i] := 0 **od**
(3) **while** there is some node i s.t. *visited*[i] = 0 **do** visit(i) **od**
}
(4) **proc** visit(i)
(5) {*visited*[i] := 1;
(6) **while** there is $j \in E_i$ s.t. *visited*[j] = 0 **do** visit(j) **od**
(7) *popped*[i] := *vis*; *vis* := *vis* + 1;
}

The following lemma identifies an important property of the spanning forest induced by algorithm Number.

LEMMA 3.3 [4]. *Let G_1 be a strong component of a graph G. Then, the vertices of G_1 together with those of its arcs that are common to the spanning forest form a tree.*

Note that the node in the strong component that is the root of this tree is the root of the strong component. Tarjan's algorithm for identifying the strong components of a graph [26] is easily modified to compute the array *popped*, and it can also identify the root (of the strong component of) each node in the graph, for example, in another array *root*. While we have presented the simpler algorithm Number for ease of exposition, in the sequel, we use Tarjan's algorithm, suitably modified, to compute the arrays *popped* and *root*. We refer to this modified algorithm as Modified_Tarjan.

3.3 Algorithm BTC

A simple-minded version of our first algorithm is a straightforward combination of the two ideas presented in the previous sections; algorithm Closure is simply run after numbering the nodes in reverse topological order (modulo back arcs). This version is denoted by BTC'.[2] For ease of presentation, we assume the existence of a procedure called *node_popped*(i), which looks at the global array *popped* constructed by Modified_Tarjan when applied on a graph G and returns the node k such that *popped*$[k] = i$. Note that, when Closure is run on an acyclic graph G following the *popped* ordering, after a descendent set S_j is added to a set S_i, the equalities $S_j = M_j$ and $U_j = \emptyset$ hold. This is not true, however, when Closure is run on a cyclic graph following such an ordering. For example, when S_j is added to S_i and (i, j) is a back arc, there exists a node k such that $k \in U_j$ and k is an ancestor of i in the spanning forest imposed by Number.

proc BTC' (G)
Input: A graph G represented by children sets E_i, $i = 1$ to n.
Output: $S_i = M_i$ ($U_i = \emptyset$), $i = 1$ to n, denoting G^*.
(1) {Modified_Tarjan(G); /*First Pass*/
(2) **for** $i = 1$ to n **do** $U_i = E_i$; $M_i = \emptyset$ **od** /*Second Pass*/
(3) **for** $i = 1$ to n **do**
(4) I = node_popped(i);
(5) **while** there is a node $j \in U_I - \{I\}$
do $M_I = M_I \cup M_j \cup \{j\}$; $U_I = U_I \cup U_j - M_I$ **od**
(6) **od**}

Modified_Tarjan also computes the array *root*, which enables an important optimization: since all nodes in a strong component have the same set of descendents, we can construct the descendent set for the root node alone.

[2] In our earlier paper on these algorithms [13], we referred to algorithm Closure as algorithm BTC, and proposed the use of algorithm number to order the nodes appropriately; the resulting algorithm is essentially what we have presented above as algorithm BTC'. The only difference is the use of algorithm Modified_Tarjan rather than algorithm Number.

Consider the processing of a node I in the algorithm above. Instead of adding the descendent set of a child j and the child itself (i.e., $M_j \cup U_j \cup \{j\}$) to S_I, we can add it to $S_{root[I]}$. This addition is carried out in BTC′ when $j \in U_I - I$; by excluding $j = I$, we avoid the addition of descendent set to itself. After processing a root node (a node I such that $I = root[I]$), we copy its descendent set to all nodes in its strong component.

If we carry out the addition of a child j and its descendent set for $j = I$ also, an additional optimization is made possible: we can ignore the marked/unmarked distinction for nodes in a descendent set.[3] In essence, whenever we add a descendent set S_j to another, the set U_j is either empty or can be ignored, based upon the following observations. In considering a child j of node I, if j and I are in different strong components, j must have already been processed; thus $S_j = M_j$ and $U_j = \emptyset$. If j is in the same strong component as I, the nodes in U_j are all in the same component too. Further, every nonroot node in the strong component is processed before the root, and is therefore in the marked subset of the root's descendent set when we process the root. The root node itself is reachable from some node in the strong component, and is added to the marked subset of the root's descendent set when we process the first such node. Thus, the addition of U_j has no effect on the descendent set computed for the root, which is subsequently propagated to all nodes in the strong component. The only other use for unmarked subsets of descendent sets is to control the **while** loop; i.e., to determine for what nodes j the addition is to be carried out. Here, we can use $E_I - M_{root[I]}$ instead of U_I since U_I is initialized to E_I and is not added to (based upon the preceding discussion), and all nodes for which the addition has already been carried out are included in $M_{root[I]}$.

We refer to the above changes collectively as the "root optimization"; they can improve performance significantly. For reachability computations, we henceforth consider only algorithm BTC, presented below, which is BTC′ with the root optimization. Based on the above observations, S_i is used directly in the algorithm, since there is no need to distinguish between its subsets M_i and U_i. On the contrary, for path computations, nodes in a strong component cannot be treated identically, and the root optimization is not applicable. Thus, when we consider path computations, we adapt BTC′, rather than BTC (Section 8).

proc BTC (G)
Input: A graph G represented by children sets E_i, i = 1 to n.
Output: S_i, i = 1 to n, denoting G^*.

```
(1) {Modified_Tarjan(G);                /*First Pass*/
(2) for i = 1 to n do S_i := ∅ od       /*Second Pass*/
(3) for i = 1 to n do
(4)    I := node_popped(i);
```

[3]This results in adding the descendent set of the root to itself for nontrivial strong components or singleton strong components with self-arcs. This unnecessary addition can be avoided, but we have chosen to keep the presentation simple instead.

(5) **while** there is a node $j \in E_I - S_{root[I]}$
do $S_{root[I]} := S_{root[I]} \cup S_j \cup \{j\}$ **od**
(6) **if I** = root[I] **then for all** $k \neq I$ s.t. root[k] = I **do** $S_k := S_I$ **od**
od}

Note that in the above algorithm, S_j is empty in line (5) whenever j is a nonroot node in the same component as I; further, S_j contains all descendents of j if j and I are in different components.

THEOREM 3.4. *Algorithm Basic_TC computes the transitive closure of a directed graph G.*

PROOF. We prove the theorem by showing that upon termination of execution of the algorithm, for each node m in every strong component, S_m contains all descendents of m in the transitive closure of G. Note that a node may constitute a trivial strong component by itself. The proof is by induction upon *popped*[r], where r is the root of a strong component.

Basis. Consider the strong component with root r such that *popped*[r] is the least over all strong components. If this is a trivial strong component, i.e., r is the only node in it, the initial value of $S_r = \emptyset$, which is not modified by the algorithm. Thus, the inductive claim holds.

If r is the root of a nontrivial strong component, every node k that is reachable from a node in the component must also be in the component; otherwise, k would belong to a strong component with a root l such that *popped*[l] < *popped*[r]. In addition, every node in the component, including the root, is a child of some other node in the component. By definition of the root of the strong component, *popped*[r] > *popped*[m] for any node $m \neq r$ in its strong component. Thus, after processing the **while** loop for $I = r$ in the algorithm, every node in the strong component has taken the role of j in statement (5) at least once since the beginning of execution. Hence, S_r contains every node m in the strong component, and by statement (6), this set is propagated to every node $m \neq r$ in the strong component. Thus, the inductive claim holds for this case as well.

Induction Step. Consider a strong component with root r such that *popped*[r] = P, and let the claim hold for all strong components with root r' such that *popped*[r'] < P. As in the basis proof, we must examine two cases. If this is a trivial strong component, every child of r is in a strong component with a root that has a *popped* number less than P. By the induction hypothesis, the descendent set for each child m of r is contained in S_m. By statement (5), this set is added to S_r while processing $I = r$. Thus, the inductive claim holds.

If r is the root of a nontrivial strong component, we can show as in the basis proof that after processing the **while** loop for $I = r$, S_r contains all nodes in its strong component. Further, if node k is not in the strong component but is a child of a node j in the strong component, S_r contains $S_k \cup \{k\}$. The reason is that the root of the strong component containing k is processed before j, which is processed before r. Since k is in a strong

component with root r' such that $popped[r'] < N$, by the induction hypothesis, S_k includes all descendents of k when we process node j. Thus, $S_k \cup \{k\}$ is added to $S_{root[j]}$, i.e., to S_r, when j is processed. This concludes the proof of the theorem. □

3.4 Algorithm Dag_DFTC

In algorithm BTC, we numbered the nodes in a first pass before computing the closure. In this and the following subsection, we attempt to improve performance by combining some of the work from the two passes, i.e., by adding the appropriate descendent sets when they are simultaneously in memory during the numbering pass. The following simple algorithm illustrates the idea, although it only works for acyclic graphs.

proc Dag_DFTC (G)
Input: An acyclic graph G represented by children sets E_i, $i = 1$ to n.
Output: S_i, $i = 1$ to n, denoting G^*.
(1) {**for** $i = 1$ to n **do** $visited[i] := 0$; $S_i := \emptyset$ **od**
(2) **while** there is some node i s.t. $visited[i] = 0$ **do** visit(i) **od**
}
(3) **proc** visit (i)
(4) {$visited[i] := 1$;
(5) **while** there is some $j \in E_i - S_i$
do if $visited[j] = 0$ **then** visit(j); $S_i := S_i \cup S_j \cup \{j\}$ **od**
}

We state the following simple theorem without proof.

THEOREM 3.5. *Algorithm Dag_DFTC computes the transitive closure of an acyclic graph G.*

The basic intuition in the above algorithm is that, when we pop up from a tree arc, the descendent set of the child is complete and must be added to the descendent set of the parent, and moreover both descendent sets are in memory. If the descendent set of the parent is paged out during the processing of the child, we must nonetheless retrieve it to identify the next child to visit, if any. Hence, by performing this addition at that time, we avoid possibly fetching one or both of these descendent sets later, in the second phase of BTC, to perform the addition. The above intuition is used to derive an "eager addition" algorithm for arbitrary graphs as well, which is presented in the next subsection.

3.5 Algorithm GDFTC

In this section we develop an algorithm that generalizes Dag_DFTC to work on arbitrary graphs. Like BTC, it avoids duplication of effort by essentially generating the descendents of only one of the nodes in a strong component. Subsequently, the sets of all the other nodes are updated, if any. In this version of the algorithm, a stack mechanism[4] is used to construct the

[4]Our stack differs from the stacks used in other graph-based algorithms for transitive closure in that it is a stack of descendent sets of nodes in nontrivial strong components, as opposed to a stack of nodes.

descendent set for (the root of) a strong component. During the process of the algorithm, each stack frame is associated with some nontrivial strong component. If we discover that some of these (potentially distinct) "components" are in fact part of the same component, then stack frames are merged to reflect this. Every stack frame f maintains two sets of nodes. The set $nodes[f]$ contains nodes that are known to be members of the strong component associated with f. The set $list[f]$ contains nodes that do not belong in $nodes[f]$ and are descendents of the members of $nodes[f]$. When the root is identified, $list[top]$ is assigned to all the nodes in $nodes[top]$ and the stack is popped, concluding the processing of the corresponding strong component.

The algorithm works roughly as follows. It traverses the graph in depth-first order, visiting each node once. The action on each traversed arc (i, j) depends on its type with respect to the spanning tree of calls to the visit( ) routines, i.e., on whether it is a tree, cross, or back arc (forward arcs are ignored). The arc type is identified with the help of the values of $visited[\,]$ and $popped[\,]$ for i and j. (Note that the array $visited$ contains integer elements in this algorithm.) In all cases, however, action is differentiated based on two additional pieces of information: first, whether i and j are in the same or different strong components, and second, in case they are in the same (nontrivial) component, whether j is the first child of i to pass the information to the latter that it is part of a nontrivial strong component. Both questions are resolved based on the values of $root[\,]$ for i and j. For any node i, $root[i] \leq n$ while i is known to be part of a strong component whose processing has not finished yet, whereas $root[i] = n + 1$ otherwise. Thus, for the first question, the value of $root[j]$ should be equal to $n + 1$ if i and j are in different strong components (processing of the component of j is over). For the second question, the value of $root[i]$ should be equal to $n + 1$ if i is not known to be in a strong component.

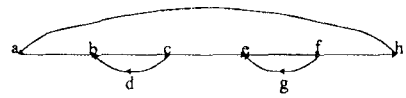
Based on the specific case identified from the above pieces of information, the algorithm takes the following actions. For all tree and cross arcs, if i and j are in different strong components, the descendents of j are propagated to the descendents of i. This is the action when operating on acyclic parts of the graph and is straightforward. The bulk of the algorithm, which involves stack manipulation, addresses the case when i and j are in the same strong component. Tree arcs are the most interesting in this case. The top stack frame always corresponds to j. If j is the first child of i through which i is detected to be part of a strong component, then i is incorporated in the top stack frame. Otherwise, the second stack frame from the top corresponds to i and is merged with the top frame. In both cases, $root[i]$ is updated appropriately. Cross and tree arcs are treated almost identically. If j is the first child of i through which i is detected to be part of a strong component, then a new stack frame is pushed on the stack and becomes associated with i. Otherwise, only $root[i]$ is updated appropriately (the top stack frame is the one corresponding to i), in slightly different ways for cross and back arcs.

Algorithm GDFTC is given below. 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 GDFTC(G)
Input: A graph G represented by children sets E_i, i = 1 to n.
Output: S_i, i = 1 to n, denoting G*.
/*   list[f]      descendents of nodes in the strong comp. of stack frame f.  */
/*   nodes[f]     nodes in the strong comp. of stack frame f.                  */
/*   top          pointer to the top of the stack.                             */
/*   visited[i]   order in which visit(i) is called.                           */
/*   root[i]      potential root of the strong comp. in which i belongs.       */
/*   popped[i]    1 if the call to visit(i) has returned.                      */
(1)  {vis := 1; top := 0; visited[n + 1] := n + 1;
(2)  for i := 1 to n do visited[i] := popped[i] := 0; root[i] := n + 1; list[i] :=
     nodes[i] := S_i := ∅ od
(3)  while there is some i s.t. visited[i] = 0 do visit(i) od}
     proc visit(i)
(4)  {visited[i] := vis; vis := vis + 1;
(5)    for each j ∈ E_i do
     /*each j considered exactly once.*/
(6)      if j ∉ S_i then
     /*body of loop not executed when j ∈ S_i*/
(7)        if visited[j] = 0 then {
     /*(i, j) is a tree arc.*/
(8)          visit(j);
(9)          if root[j] = n + 1 then S_i := S_i ∪ S_j ∪ {j}
     /*i, j in different strong components.*/
(10)         elseif root[i] = n + 1
     /*first detection of i being in a strong comp. (through j)*/
(11)           then add_in_top_frame(i, j)
(12)           else merge_top_two_frames(i, j)}
(13)       elseif popped[j] = 1 then
     /*(i, j) is a cross arc.*/
(14)         if root[j] = n + 1 then S_i := S_i ∪ S_j ∪ {j}
     /*i, j in different strong components.*/
(15)         else {if root[i] = n + 1 then push_new_stack_frame(i, j);
     /*first detection of i being in a strong comp.*/
(16)               update_root_non_back(i, j)}
(17)       elseif popped[j] = 0 then {
     /*(i, j) is a back arc.*/
(18)         if root[i] = n + 1 then push_new_stack_frame(i, j);
     /*first detection of i being in a strong comp.*/
(19)         update_root_back(i, j)}
       od
```

Fig. 1. A strongly connected graph.




```
(20)   if i = root[i] then {
       /* Propagate descendents of root to the rest
       of the nodes in the strong comp. */
(21)       root[i] := n + 1;
           for each j ∈ nodes[top]
             do S_j := S_i•nodes[top]; root[j] := n + 1 od;
           top = top − 1}
(22)   popped[i] := 1
       }
       proc add_in_top_frame(i, j)
(23)   {list[top] := list[top] ∪ S_i; S_i := •list[top];
(24)   nodes[top] := nodes[top] ∪ {i}; root[i] := root[j];}
       proc merge_top_two_frames(i, j)
(25)   {list[top − 1] := list[top − 1] ∪ list[top];
(26)   nodes[top − 1] := nodes[top − 1]•nodes[top]; top = top − 1;
(27)   update_root_non_back(i, j)}
       proc push_new_stack_frame(i, j)
(28)   {top = top + 1; list[top] := •S_i; nodes[top] := {i}}
       proc update_root_non_back(i, j)
(29)   {if visited[root[j]] < visited[root[i]] then root[i] := root[j]}
       proc update_root_back(i, j)
(30)   {if visited[j] < visited[root[i]] then root[i] = j}
```

We prove that GDFTC is correct in an appendix. As mentioned above, an important aspect of the algorithm is that duplication of effort is avoided by constructing the descendent list of just one node (the root) of a strong component and subsequently copying this list for each node in the component. One of the reasons for the complexity of this algorithm is the need to keep track of strong component information while constructing descendent lists on the fly. We illustrate the operation of the algorithm on an example in which a single strong component is discovered in a piecemeal fashion. Figure 1 shows the input graph. The whole graph is one strong component. Assume that the nodes are visited in the order a, b, c, d, e, f, g, and h. Thus, the back arcs (d, b) and (g, e) are discovered before (h, a) is. This results in two potentially independent components to be pushed on the stack, namely, $\{b, c, d\}$ and $\{e, f, g\}$. After (h, 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 f again, statement (12) in the algorithm is executed, and the two frames at the top (corresponding to a and e respectively) are merged into one. When c is reached, similar actions are taken, so that when a, the root, is reached, all its descendents are correctly found in the top list.

4. IMPLEMENTATION OF ALGORITHMS

This section describes the main aspects of our implementation of the algorithms, analyzing the specific choices that we made when multiple alternatives were available so that the results of a performance evaluation may be clearly interpreted. These aspects include storage structures for graphs, physical clustering of descendent lists, memory management, and duplicate elimination. Some of the techniques that we present below, or closely related ones, have also been used by others for implementing transitive closure algorithms [3, 14, 22].

4.1 Storage Structures

We represent and store graphs in several forms. First, both the input and output graphs of the algorithms are stored in a plain tuple format, as compactly as possible. Tuples with the same source attribute (arcs with the same tail) are stored consecutively in the file, but otherwise no special structure is assumed.

Second, during the course of the execution of all algorithms, graphs are represented as descendent lists. The restructuring from arc-tuples to descendent lists occurs as part of the first pass of BTC, whereas it is the first step of all other algorithms, and we refer to it as the *restructuring* phase. To accommodate descendent lists, every page is divided into some number of *blocks*. Each block can store a constant number of node names (equal to the *blocking factor*), representing arcs from a common source to the stored nodes. There is a pointer to an additional block if there are more arcs with a common source than the blocking factor. In addition, each page contains an array index with one entry for each block; the entry contains the common source of arcs in the block, and a bit indicating whether the block is empty or not. Given a fixed size page, increasing the blocking factor implies that fewer blocks can be stored in a page. Thus, choosing the perfect blocking factor depends on the following trade-off: a high blocking factor saves space for a long descendent list, since its source is factored out and stored only once for each set of descendents that fit in each block; on the other hand, a high blocking factor wastes space for a short descendent list, since a large portion of a block remains empty and unused. This trade-off will become clear from the results of our experiments.

Third, whenever a descendent list is processed in memory, i.e., whenever nodes are copied from it or into it, its contents are also replicated in the form of an adjacency vector. The vector has an entry for every node in the graph, which is equal to 1 if the corresponding node has been identified as a descendent of the source of the corresponding descendent list and is equal to 0 otherwise. This allows for fast duplicate elimination, since the descendent list does not have to be searched before adding a node to it: a straight lookup at the adjacency vector is enough (Section 4.4). The size of the adjacency vectors is calculated in the first steps of each algorithm, when the graph is

transformed from tuples to descendent lists, at which time the number of nodes is counted.

In addition to the above, an array containing some useful information is maintained in memory, with an entry for each node in the graph. Each such entry contains the following items: (a) the outdegree of the node, (b) the root of the strong component to which the corresponding node belongs, (c) the rank of the node in the topological order obtained by the depth-first traversal of the graph, (d) an indication of whether the node has been visited and processed or not, (e) a pointer to adjacency vector of the node (if it is memory), and (f) the page number of the file on disk where the descendent list of the node is stored. For leaves, the last entry is equal to a particular reserved value, making it unnecessary to store empty descendent lists for them, thus saving space and also many useless accesses to disk.

4.2 Descendent List Ordering

In the implementation of the BTC algorithm, we take advantage of its two pass structure and use much information from the first pass to expedite the second pass, in which the transitive closure is computed. This is not possible with the remaining algorithms that we have considered. For cyclic graphs, their strong components are identified in the first pass, and this allows us to essentially compute the transitive closure of the condensation graph, improving performance significantly. The effect of computing the closure of the condensation graph is also achieved by Schmitz and GDFTC, in other ways. Inter- and intradescendent list ordering are the two other important optimizations made possible by the ordering of nodes obtained in the first pass of BTC. These result in significant performance improvements, and for the most part, their effect cannot be realized by Schmitz or GDFTC. We elaborate on these optimizations below.

In BTC, the descendent lists of a graph are constructed during the first pass. At that time, of course, they only contain the children of each node. We store the descendent lists in the reverse topological order of their source nodes. This has the effect that many nodes that are close together in that order, and are therefore likely to be close in the graph as well, have their descendent lists stored on the same page. For example, a parent and its children are often on the same page. Also, since nodes are processed in reverse topological order, nodes processed consecutively are likely to be on the same page as well. Hence, the above interdescendent list ordering results in very high hit ratios in the buffer pool and thus in less I/O. The above technique does not help when the number of children of each node in the graph is so large that only one children list or less fits on a page.

Another benefit of the first pass of BTC is that the topological ordering of the nodes can be used to reduce the production of duplicates. Specifically, consider an arc (i, k) in G and assume that there is also a path between i and k whose first arc is (i, j). Clearly, the inequalities $i < j < k$ hold in the topological order of G. If j is processed before k when dealing with the children list of i (statement (5) of BTC), then k will be found in S_i when its turn comes, and no action will be taken on it. If k is processed first, however,

then j will have to be processed as well, and the descendents of k will essentially be derived twice for i. To avoid this unnecessary computation, the nodes in each descendent list produced by the first pass of BTC are stored (and processed) in topological order, i.e., j is stored first in the above example. This intradescendent list ordering has a considerable effect on I/O and CPU performance. The above ordering has also been used by Agrawal and Jagadish in their Hybrid algorithm [3].

As we mentioned earlier, the above data orderings cannot be used in Schmitz or GDFTC, because of their "on-the-fly" type of processing before the necessary information is available. The effect of the intradescendent list ordering, however, can also be achieved by computing the arc basis of a graph and using that for the actual transitive closure computation. As mentioned in Section 2, Schmitz proposed that as a variant of his algorithm. Such a preprocessing step, however, adds some nontrivial cost to the overall execution, as opposed to the intradescendent list ordering of BTC, whose cost is negligible, since it is a by-product of the first pass of the algorithm (and is accounted for in the numbers that we present). We should also note that adding the first pass of BTC to either Schmitz or GDFTC is not very meaningful: the added complexity of these algorithms with respect to the second pass of BTC gains nothing, and potentially costs more.

4.3 Memory Management

Several data structures are assumed to remain in main memory throughout the execution of all algorithms. The most important such structure is the array mentioned in Section 4.1. This is in addition to the buffer pool, which is used to store the following: (a) arc-tuples, for the initial input and final output of the algorithms, (b) descendent lists and (c) adjacency vectors. For all algorithms, depending on the execution phase, a buffer pool of size M is divided among the above types of data as follows.

Restructuring
- $M - 1$ pages for input arc-tuples
- 1 page for the constructed descendent lists

Main algorithm
- 1 page for output arc-tuples
- M-2 pages for descendent lists
- 1 page for adjacency vectors

During restructuring, LRU is used as the page replacement policy among the arc-tuple pages. During the main algorithm, LRU is used among the adjacency vectors to manage the space in the single page devoted to them. With respect to the pages storing descendent lists, we have experimented with two replacement algorithms: LRU and a specialized algorithm that we introduce below called *Least Unprocessed Node Degree* (*LUND*).

LUND works as follows. The descendent lists that are in main memory at any point are divided into two classes. The first class contains lists, called

complete lists, whose source is a node that has been processed already, i.e., all its descendents have been found. Clearly, any future reference to such a list S_j is via an arc pointing to j, with the goal of copying S_j into the list of the source of that arc. The second class contains all the other lists, called *incomplete lists*, i.e., those whose source is either still being processed or has not yet started being processed. Future references to such a list can be due to both arcs coming into the node and arcs going out of the node. (For every outgoing arc, the list is requested once so that the descendents of the head of the arc are added to the list. For every incoming arc, the list is requested once so that it can be added to the descendents of the tail of the arc.) In LUND, the candidate pages for replacement in the buffer pool are chosen from among those that contain only complete lists and a fraction f of the least recently used pages with incomplete lists. For each list on a candidate page, its *Unprocessed Node Degree* (*UND*) is computed as the sum of the in- and out-degree of the source node for the list minus the number of times that the list has already been requested. Thus, the UND of a list is the number of requests for the list that are yet to be made, or equivalently, the number of unprocessed arcs incident on the corresponding node. LUND adds the UNDs for all lists in each page and then chooses the page with the least sum as the victim for replacement. The intuition behind the LUND policy is twofold. First, the most recently used incomplete lists are likely to be needed in memory again soon, and should not be paged out. These are included in the fraction of incomplete lists that are not considered for replacement. Second, among candidate lists, without any further information about the graph structure, the algorithm assumes that the fewer the arcs that need to be processed for the source of a list, the further away in the future that list will be requested. This algorithm has been justified by the results of several experiments, some of which are discussed in Section 6.2.2.

A final issue to consider is related to page splits. If all blocks of a page are occupied and one of them is full and needs to be expanded, the page must split into two pages. At that point, a decision must be made on how descendent lists will be divided between the new pages. We have experimented with two approaches. The first one is to randomly divide them between the pages; the second one is to take into account the UND of the source nodes of the lists in the original page and separate those with small UND from those with large UND. The specific criterion for the UND-based separation is not important—we have experimented with several of them with no major effects on the performance. The intuition behind the second approach is that nodes with high UND are expected to be accessed frequently in the future. Hence, combining all of them together in a page increases the chances that the page will stay in main memory long enough for much of the processing of these lists to be done without additional I/O. Results of some initial experiments showed that the first approach is the preferred one when using LRU, whereas as expected, the second approach is the preferred one when using LUND. Therefore, all experiments presented in the results sections are for these combinations.

4.4 Duplicate Elimination

In this section, we briefly describe the algorithm that is used for duplicate elimination at linear CPU cost. As mentioned in Section 4.1, when copying nodes from a descendent list S_j to another list S_i, adjacency vectors for both lists exist in main memory. For every node k in S_j, the corresponding entry in the adjacency vector of S_i is checked: if it is equal to 0, then k is added to S_i and the bit is switched to 1; otherwise no action is taken on k, since it already exists in S_i. This corresponds to O(1) cost for each node in S_j, i.e., to a cost that is linear in the length of S_j. The cost of constructing the adjacency vectors is also linear in the length of the lists, so duplicate elimination is very efficient.

5. PERFORMANCE EVALUATION TESTBED

We implemented several versions of all three algorithms BTC, GDFTC, and Schmitz, using C on a VAXstation 3200 under UNIX. The file page size and the buffer page size in our implementation were chosen to be 2 Kbytes to match with the corresponding sizes of the machine. With this size, each page can fit 256 arc-tuples for the input and output representation of graphs, and 30 and 72 blocks with block size 15 and 5, respectively, for the descendent list representation.

Although all experiments have been run with no other users on the machine, since we do our own buffer management the UNIX-provided elapsed times are not meaningful. In our experiments, we relied on UNIX-provided CPU times and our own counting of I/O based on the implemented page replacement strategy and the amount of available memory in each case. For all graph-based algorithms, there is an option in writing the output: the descendent set of the root of a strong component can be copied once for each node in the component, or a single copy can be written out, with pointers to it from each node in the strong component. We assume that the same choice is made by all the algorithms; this means that the cost of reading the input and writing the output are the same for all algorithms, and unavoidable. With respect to I/O cost, therefore, the numbers presented in this paper do not include the initial cost of reading in the original graph once and the final cost of writing out the transitive closure once. (The only exception is in Section 11.3, where we compare graph-based algorithms with nongraph-based algorithms.) All other reads and writes performed during the execution are included in the numbers presented.

There are several interesting parameters that affect the performance of the algorithms. They can be divided into parameters of the implementations of the algorithms and parameters of the data. They are discussed in the following two subsections. The third subsection explains how our input graphs were generated.

5.1 Parameters of Algorithm Implementations

There are three interesting parameters of the algorithm implementations: the number of buffer pages, the buffer replacement policy, and the blocking

Table I. Parameters of Algorithm Implementations and their Tested Values

Parameter	Symbol	Values
Buffer size (pages)	M	10, 20, 50
Buffer replacement policy	-	LRU and LUND (f=0.25)
Blocking factor	B	5 and 15

factor. The buffer size (denoted by M) was varied considerably. In the results sections, the only values discussed are $M = 10$, 20, and 50, because all algorithms need at least 10 pages of memory to run, and no interesting phenomena occur beyond 50 pages (the I/O cost drops sharply, as expected). A minimum of ten pages are necessary because all algorithms require that at least two descendent lists can fit in memory at the same time. For 2000 node graphs, this accounts for eight pages in the worst case. In addition, two more pages are needed, one for the adjacency vectors and one for input or output arc-tuples. We experimented with several buffer replacement policies, in particular, LRU and four versions of LUND with the fraction f being equal to $f = 0.25$, 0.5, 0.75, and 1.0, respectively. Among these versions of LUND, the one with $f = 0.25$ was almost always either the best or close to it. Hence, we show the results for LRU and LUND with $f = 0.25$ only. In each case, the presented costs are for the best of the two policies for that case. Finally, we experimented with two blocking factors, $B = 5$ and 15. The effect of the value of B depended on the input graph type. This is discussed in detail in Section 6.2.3. However, the relative performance of the algorithms remained unaffected by B, so the results in all other sections are for $B = 15$. The above space of parameters and tested values is summarized in Table I.

5.2 Parameters of Data

All relations used in our experiments contained integer node identifiers, which represent the best case for efficiency. This is without any loss of generality, however, because even if a given relation is not in this form, it can be transformed to it by a single pass over its tuples. In addition, the integers used were random numbers in a specific range, so that the actual values that represented the nodes would not bias the performance of the algorithms. Also, for any specific setting of the values of the parameters described below, all algorithms under comparison were run on the same input graph, so that no differences in the specific choice of node identifiers or other secondary characteristics could affect the results.

We show results for both acyclic and arbitrary graphs below. We also experimented with trees, but since those tests did not offer any additional insights beyond what was observed for acyclic graphs, we do not present them. The following are the parameters that were used to characterize graphs, with the symbols that denote them in parentheses: the number of nodes (N), the outdegree or branching factor of each node (b), and the depth (d). Preliminary experiments with several values of N showed that the main conclusions of this study seem to be unaffected by N. Hence, we only present

results for the value $N = 2000$. (We also studied a complete suite of graphs with $N = 1000$, with values of other parameters being varied exhaustively as with $N = 2000$. The trends and analysis for $N = 1000$ are identical to those for $N = 2000$.) We experimented with five values of b, in particular $1 \leq b \leq 5$, and we present results for all of them. Higher values of b were not tested extensively, because in many cases, the generated graphs have transitive closures that are almost complete, and such extremes are rather uninteresting.

Finally, the depth of a graph is defined to be equal to the maximum length of any *simple* path in the graph. Its importance in our study can be understood as follows. Consider some point during the execution of an algorithm and let the nodes for which processing has started for computing their descendent lists but has not finished yet to be the *active* nodes at that point in time. For example, in GDFTC, the active nodes are those for which visit( ) has been called but has not yet returned. The number of active nodes has a direct influence on performance, because if there is not enough memory to hold the descendent lists for all these nodes, some of them have to be paged out and brought back in again in the future. The importance of depth stems from the fact that it represents the maximum number of nodes that can ever be simultaneously active. However, depending on the order in which nodes are processed, the actual maximum number of simultaneously active nodes that is encountered may be much smaller than the depth. This "operational" parameter is called the *observed depth* and is clearly more helpful than the actual graph depth in understanding the performance of algorithms. Of course, the two parameters are in general closely related, and high (low) values in one are usually related to high (low) values in the other. Hence, in the rest of this paper, we use the observed depth instead of the actual depth as one of the main graph parameters that affect performance. For simplicity, however, we continue to use the term depth. In attempting to experiment with a range of possible graph depths, we realized that we were not aware of any way to generate random graphs with a specific value of d. Hence, we used another parameter, called *locality factor* and denoted by l, that gave us the ability to generate and experiment with both shallow and deep graphs. The definition of l is better understood in conjunction with the algorithm used to generate the graphs for our experiments, so it is presented in the next subsection.

Table II summarizes the above space of parameters and tested values, including l for completeness. For every set of values of these parameters examined in our experiments, five graphs with the corresponding characteristics were generated and tested. In all cases, the results presented in subsequent sections represent averages of those five graphs.

5.3 Graph Generation Process

For all experiments, graphs were generated randomly based on values of their parameters. As mentioned above, the specific values representing the nodes were chosen randomly as well. In all cases an arbitrary ordering was imposed on the nodes. For an acyclic graph, the children of the ith node in

Table II. Parameters of Data and their Tested Values

Parameter	Symbol	Values
Number of nodes	N	1000 and 2000
Outdegree	b	1, 2, 3, 4, 5
Locality factor	l	$N/100$ and N

that order were generated by randomly choosing b nodes among those whose rank in the order was in the range $[i + 1, \min(i + l, N)]$. There are several comments that need to be made on the above. Acyclicity is achieved by the fact that nodes are always connected to ones that are further down in the ordering, which is essentially some topological order of the generated graph. Also, note that the locality factor is defined as the size of the subset of nodes from which neighbors are chosen (except for the boundary conditions, where nodes are too close to the end of the ordering). Clearly, l can range up to the total number of nodes, in which case a truly random graph is generated. For the same number of nodes and outdegree, the smaller the locality factor is, the greater the depth of the graph. In our experiments, the locality parameter affected the depth of the graph in precisely this way. Note that the locality factor captures a characteristic of the graph itself, as opposed to the homonymic parameter of Agrawal and Jagadish [2], which captures a characteristic of the integer values assigned to the graph nodes.

For a cyclic graph, the children of the ith node in the arbitrary order mentioned above were generated by randomly choosing b nodes among those whose rank in the order was in the range $[\max(1, i - 1), \min(i + l, N)]$. Given the above range, the possibility for cycles should be clear. In this case, however, the locality factor does not affect the depth of graphs as in the acyclic case. To a large extent, the depth remains unaffected by the value of l. Hence, we mostly concentrated on the results that were obtained with a single locality factor for cyclic graphs.

6. COMPUTING THE TRANSITIVE CLOSURE OF ACYCLIC GRAPHS

In this section, we present the results of the performance evaluation of the studied algorithms on acyclic graphs. The first section presents the results for a typical example and discusses the general trends that were observed throughout the experiment. The following two subsections elaborate on the details of how the various parameters of the algorithm implementations and of the graphs affect the performance of the algorithms. For reasons that will become clear later, we present numbers for two versions of BTC. The second version is called BTC- and is like BTC except that it does not use the inter- and intradescendent list orderings that were described in Section 4.2 to store the data for the second pass. In the next subsection only, we also present numbers for a third version of BTC, called BTC-+, which is between BTC and BTC-, in the sense that it uses inter- but does not use intradescendent list orderings. We should emphasize again that the numbers presented in this paper do not include the initial cost of reading in the original graph once and

Table III. Performance Results on Acyclic Graphs with $N = 2000$, $b = 5$, and $d = 14$

Algorithm	I/O Cost (pages)	I/O Ratio over winner	CPU Time (sec)	CPU Time Ratio over winner
BTC	6685	1.00	61.5	1.00
BTC-+	7294	1.09	67.6	1.10
BTC-	10872	1.63	68.3	1.11
GDFTC	10506	1.57	68.4	1.11
Schmitz	10761	1.61	69.8	1.13

the final cost of writing out the transitive closure once, which are the same for all algorithms.

6.1 General Trends

Table III shows a typical example of the number of I/O operations and CPU time (in seconds) of all the algorithms. The specific setup has $B = 15$ and $M = 50$, and the numbers represent the best buffer replacement strategy between LRU and LUND. The corresponding graph parameters are $N = 2000$, $b = 5$, and $d = 14$, which was generated by using $l = 2000$. Each graph with these characteristics contains approximately 5000 arcs, and its transitive closure contains approximately 667000 arcs.

This example is typical of the relative performance of the algorithms on acyclic graphs and indicates the following trends. With respect to I/O, BTC is the clear winner. This was observed throughout the experiments, for all types of graphs and with all setups examined. This came as a surprise in the beginning, because BTC is the simplest of all algorithms and the expectation was that it would be the poorest performer. As we can see, however, from the difference between the cost of BTC and BTC-, the performance is affected significantly by the proper inter- and intradescendent list ordering that is possible after the first pass of BTC.

If we ignore the advantage of such orderings, i.e., consider BTC-, the performance is very similar to both Schmitz and GDFTC. This was again another observation that was evident throughout most of our experiments, with all three of these algorithms having relatively close I/O costs and being interchanged in superiority depending on the graph characteristics. What primarily distinguishes BTC-, Schmitz, and GDFTC is the different timing of the addition of nodes to descendent lists. As mentioned in earlier sections, the three algorithms fall on a spectrum, where BTC- delays additions until the whole graph has been explored in the first pass and all strong components have been identified, Schmitz is a bit more eager and does additions after each strong component has been identified, whereas GDFTC is the most eager of all and does additions every time an arc is explored. In all other major respects, the algorithms are identical, e.g., they all use depth-first traversal to explore the graph and generate a single descendent list for each strong component. As can be seen from the results, however, their sole significant difference does not have any major effect on performance. The

earlier additions to descendent lists are made, the higher the chances that the relevant lists are in main memory, but also the larger the lists that need to be manipulated for a large part of an algorithm's execution. There seems to be a mutual balance between these two conflicting trade-offs, which results in the similar performance observed for the three algorithms. On the other hand, taking advantage of the first pass of BTC to order the data appropriately has a significant payoff.

A comparison of the costs of BTC and BTC-+ identifies the benefits of the intradescendent list orderings that allow BTC to take the maximum possible advantage of the marking optimization. It can be seen that with respect to both CPU and I/O cost, BTC is about 10% more expensive if these orderings are not used. For other algorithms, e.g., Schmitz, computing the arc basis would have the same effect with respect to maximizing the set of arcs that can be ignored. Hence, since Schmitz would have the additional overhead of computing the arc basis, the difference between the costs of BTC- and BTC-+ provides an upper bound for the improvement in Schmitz's algorithm through computing the arc basis. As we mentioned earlier, the sole purpose for the introduction of BTC-+ was the analysis in this paragraph; we discuss this version of BTC nowhere else in the paper.

With respect to CPU time, there is no major difference among the algorithms, with BTC again being slightly more efficient than the remaining algorithms, primarily because of the intradescendent list ordering that it employs, which allows it to avoid traversing several arcs in the graph. In addition, even assuming a relatively fast disk with 25msec page access time, all algorithms are heavily I/O bound. Again, both of these observations held throughout the experiments. Hence, we are mostly concerned with the I/O cost for the rest of the section.

6.2 Effect of Parameters of Algorithm Implementations

In the subsections below, we discuss how the performance of the various algorithms is affected by the buffer pool size M, the buffer replacement policy, and the blocking factor B.

6.2.1 *Number of Available Buffers.* Figure 2 shows a typical example of the I/O cost of the examined algorithms as a function of the number of available buffers M. This is for acyclic graphs with $N = 2000$, $b = 5$, and depth $d = 14$, which were again generated using $l = 2000$, i.e., for the same data discussed in Section 6.1. Thus, the minima exhibited by each algorithm in Figure 2 correspond to the results presented in Table III.

As expected, for all algorithms, the I/O cost decreases as the buffer size increases. An interesting observation is that both versions of BTC benefit from an increase in the buffer size much more than GDFTC or Schmitz. This becomes apparent when we compare the results for $M = 10$ and $M = 50$. In the first case, BTC is slightly less expensive than GDFTC and Schmitz, whereas BTC- is slightly more expensive than them. In the second case, BTC is considerably cheaper than GDFTC and Schmitz, whereas the cost of BTC- has dropped to almost that of the other algorithms. Breaking the algorithm

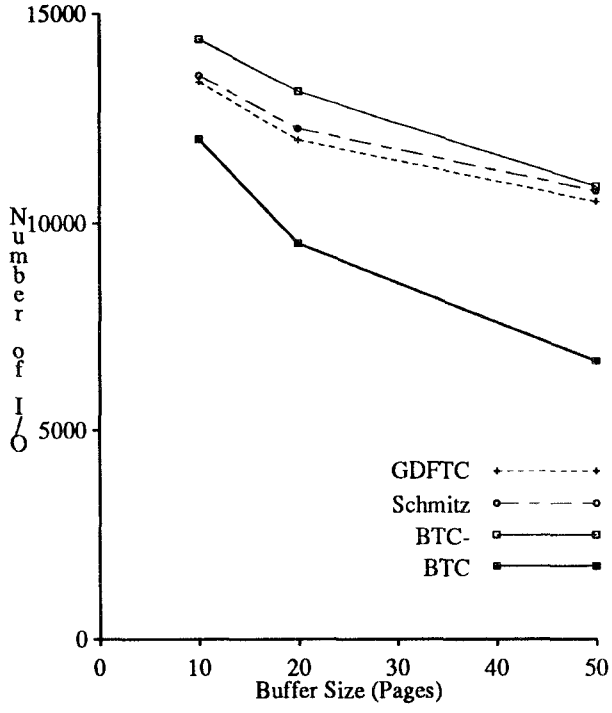


Fig. 2. I/O cost as a function of buffer size when the algorithms are applied on acyclic graphs.

into two passes seems to be the main reason for this sensitivity. In each pass, BTC has to deal with smaller amounts of data than the other two algorithms. Hence when the buffer size increases, maintaining all necessary information in memory at any point is easier than with the other algorithms.

6.2.2 *Buffer Replacement Policy.* All algorithms have exhibited very similar patterns of behavior when experimenting with the two buffer replacement policies studied. For the same graph, almost always, the same policy was the preferred one for all algorithms. Figure 3 shows the winning regions for the two policies in the space of the parameters that characterize graphs, i.e., outdegree and depth (number of nodes does not affect the relative ranking of the policies). Two separate figures are shown, one for a small and one for a large buffer size.

From Figure 3, one concludes that both policies are good, each one having its strengths and weaknesses. In general, LRU is better in large and deep graphs, whereas LUND is better in smaller and shallower graphs. For BTC, this can be explained as follows, but similar arguments hold for the other algorithms as well. With respect to the graph depth, cross arcs are much more likely to have their ends being far apart in the reverse topological order in a deep graph than in a shallow one. The reason is that in deep graphs, there tend to be longer paths in the spanning forest imposed by the depth-first traversal, which implies that cross arcs into a node from nodes in other

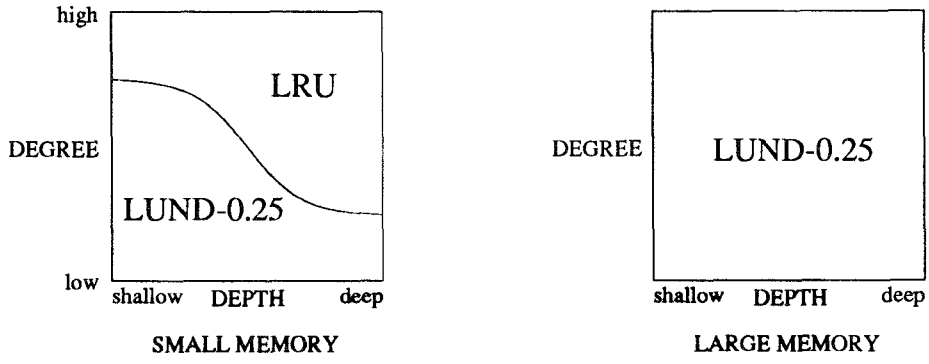


Fig. 3. Winning regions of LRU and LUND in the space of b and d of acyclic graphs.

branches of the forest are processed much later than the node itself. Hence, nodes that are deep in such a graph and have not been accessed for some time are good candidates for having their descendent lists be replaced, even if they still have arcs pointing to them that have not been processed, because these will likely be processed quite far in the future. LRU will indeed favor pages with such lists for replacement and therefore perform better than LUND, which will try to keep them in memory if their UNDs are high. The opposite argument holds for shallow graphs, for which LUND is the preferred policy. With respect to the graph size, increasing the outdegree of nodes slightly increases the graph depth but also the number of cross arcs. Hence, the problem mentioned above is intensified in larger graphs, which is why LRU wins in those, but loses in smaller ones.

The relative performance of the two policies is also affected by the size of the buffer pool. As one can see from Figure 3, as the buffer pool size increases, the performance of LUND improves as well and eventually dominates LRU for all graph sizes and depths that we tried. This is again related to the above argument. With a larger buffer pool, more descendent lists can be held in memory, which implies that the list of a not recently used node with high UND becomes a better candidate for being kept in memory, because the chances that it will be accessed are higher. Therefore, LRU loses its appeal and LUND dominates.

In addition to the previous analysis, we would also like to mention that in terms of actual cost, the difference between LRU and LUND is never dramatic (except for a few cases with small graphs). Hence, even if a system does not implement LUND, which is a specialized algorithm, but adopts the straightforward LRU, its performance will not suffer much, and the conclusions of this paper are still valid. To illustrate the above, we present two examples in Table IV with the I/O cost of BTC for the two different policies. The first example is for deep graphs (d = 124–129) with a small number of buffers (M = 10), which should favor LRU. The second one is for shallow graphs (d = 8–14) with a larger number of buffers (M = 50), which should favor LUND. The remaining parameters are B = 15 and N = 2000, whereas

Table IV. Comparison of LRU and LUND on Shallow and Deep Acyclic Graphs with $N = 2000$

Depth and Number of Buffers	Outdegree	LRU		LUND	
		I/O Cost (pages)	I/O Ratio over winner	I/O Cost (pages)	I/O Cost Ratio over winner
Deep and $M = 10$	1	1797	1.020	1762	1.000
	2	12477	1.008	12373	1.000
	3	18050	1.000	18140	1.005
	4	21055	1.000	21312	1.012
	5	22145	1.000	22411	1.012
Shallow and $M = 50$	1	171	1.336	128	1.000
	2	2289	1.225	1868	1.000
	3	3882	1.059	3666	1.000
	4	5222	1.012	5161	1.000
	5	6764	1.012	6685	1.000

the node outdegree varies from $b = 1$ to 5. Note that, if the case of small shallow graphs is excluded, the worst policy is always within 6% of the best one, which should be considered an insignificant difference. Nevertheless, LUND proves to be the best algorithm overall, being superior in most cases, and losing by no more than 1% when it is inferior to LRU.

6.2.3 *Blocking Factor.* The effect of blocking factor on performance is very similar for all algorithms. Hence, for clarity of presentation, we only present actual numbers for BTC in this section. Figure 4 shows a typical example of the I/O cost of BTC as a function of the node outdegree for the two different values of blocking factor that we tested, i.e., $B = 5$ and 15. Again, this is with $M = 50$ buffers and for acyclic graphs with $N = 2000$ and depth that varies from $d = 10$ to 14 (locality factor $l = 2000$). The results presented in the figure offer no surprises. Depending on whether the average descendent list in the transitive closure is large or small, $B = 15$ and $B = 5$ is the preferred value for the blocking factor, respectively. In Figure 3, small descendent lists are found in the graphs with outdegree $b = 1$, whereas large ones are found in the graphs with $b \geq 2$. Likewise, in the experiments with graphs generated with a small locality factor, the graphs are much deeper, which results in large descendent lists even for outdegree $b = 1$ and renders $B = 15$ the preferred value for all graphs tested in that case.

6.3 Effect of Parameters of Data

Recall that for comparing the various transitive closure algorithms, the important graph parameters are b and d, since differences in N have shown no influence on the major conclusions of this study. Each of the following sections discusses the effect that one of b and d has on performance.

6.3.1 *Outdegree.* The effect of b on the behavior of the algorithms when N and d remain relatively constant is shown in Figure 5. The specific setup has $B = 15$ and $M = 50$. The graphs have $N = 2000$ and d varying between 10 and 14, while the size of the corresponding transitive closures grows from approximately 14000 arcs to 667000 arcs. As expected, as b increases, all

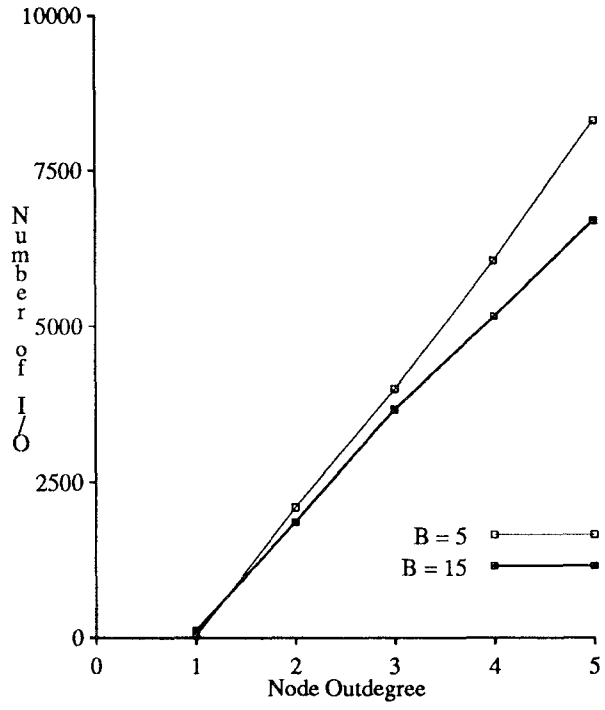


Fig. 4. I/O cost as a function of node outdegree when BTC uses different blocking factors on acyclic graphs.

algorithms need to deal with larger graphs, so both their I/O and CPU costs increase as well. With respect to CPU cost, all algorithms have very similar costs, so we do not discuss this any further. With respect to I/O cost, BTC is superior to all other algorithms by a large margin, whose absolute value increases and its relative value decreases with b. Again, a comparison of BTC and BTC- shows that the main factor for the algorithm's superiority is the inter- and intradescendent list ordering that is possible after the first pass. BTC-, Schmitz, and GDFTC have very similar costs, with BTC- being always the worst and GDFTC being always the best. The reasons for this are explained in the following section, when the effect of the graph depth on the performance of algorithms is discussed.

6.3.2 *Depth.* The effect of d on the behavior of the algorithms when N and b remain relatively constant is seen when comparing Figure 6 with Figure 5 from the previous section. For the results of Figure 6, the specific setup has $B = 15$ and $M = 50$. The graphs have $N = 2000$ and d varying between 124 and 129, which was generated by using $l = 20$, while the size of the corresponding transitive closures grows from approximately 183000 arcs to 1971000 arcs. Note the dramatic increase in the depth and the transitive closure size compared to the graphs of the previous section. The increase in the cost of all algorithms with b, the similarity of CPU costs among all

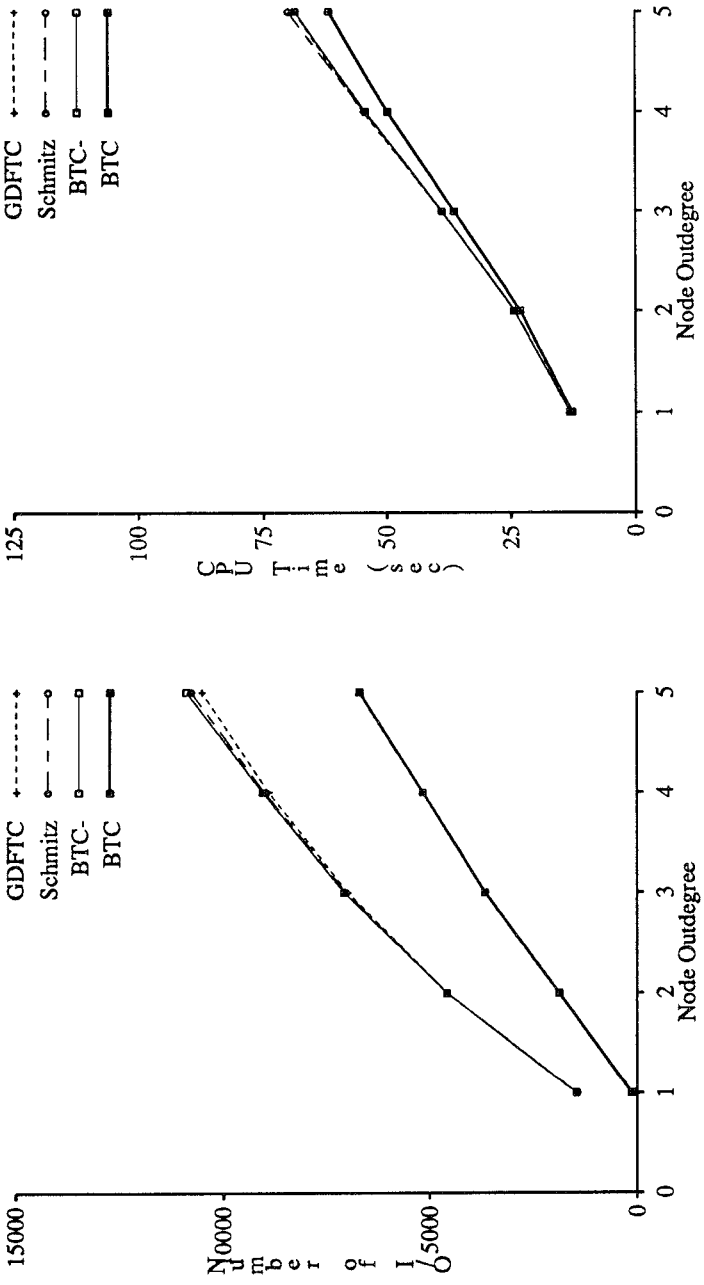


Fig. 5. I/O and CPU cost as a function of outdegree when the algorithms are applied on shallow acyclic graphs.

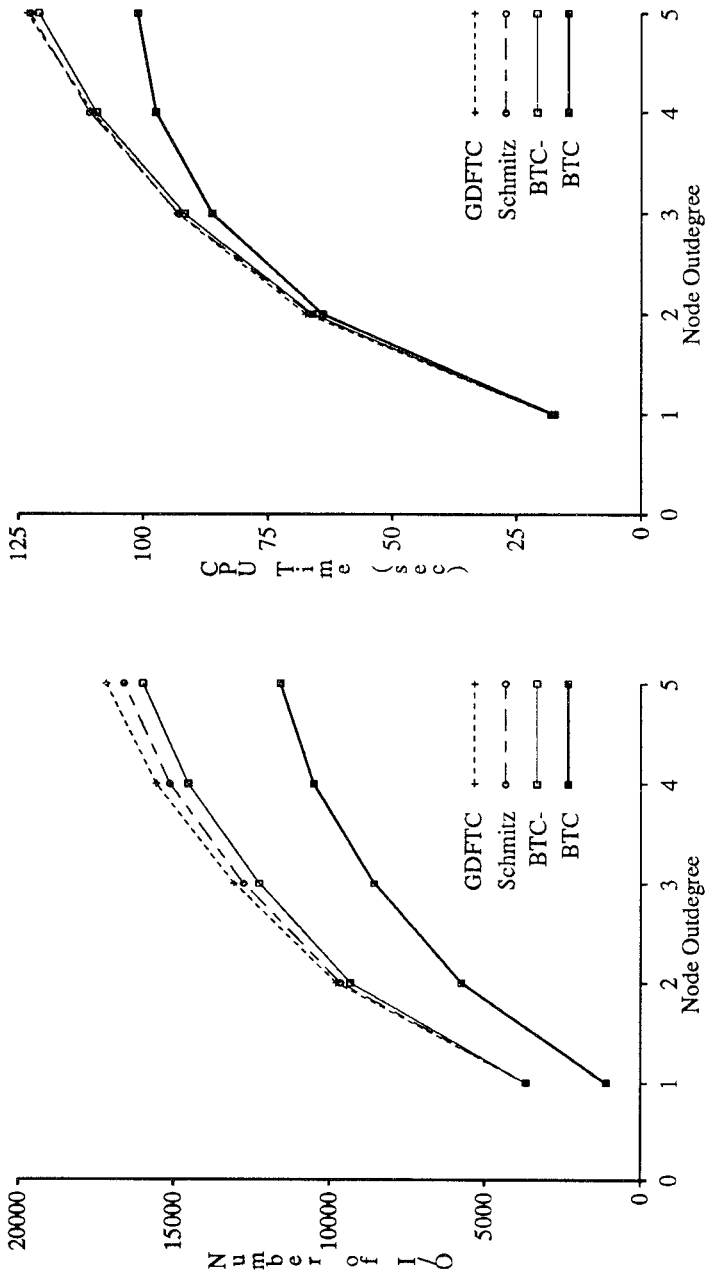


Fig. 6. I/O and CPU cost as a function of outdegree when the algorithms are applied on deep acyclic graphs.

algorithms, and the clear superiority of BTC in I/O cost hold for these deep graphs just as for the shallow graphs corresponding to Figure 5. Thus, we comment no further on these aspects, but concentrate on two issues that are affected by the graph depth.

First, it is clear that both the I/O and the CPU costs of all algorithms increase with depth. This is primarily due to two facts. The first fact is that, with deep graphs, the time between two requests for the same descendent list is in general longer, which increases the chances of the corresponding page being paged out. This has a direct effect on the I/O cost but also an indirect effect on the CPU cost, since the increased paging of descendent lists results in an increase in the adjacency vector calculations and descendent list traversals. The second fact is that with deep graphs, the size of descendent lists is in general much larger. This increases the I/O cost because it reduces the number of lists that can remain in memory at any one time and also increases the CPU cost because longer lists need to be manipulated.

Second, although BTC-, Schmitz, and GDFTC again have very similar I/O costs, their performance order is reversed compared to what it was for shallow graphs. That is, it appears that for shallow graphs it pays off to add descendent lists eagerly (as GDFTC does), whereas for deep graphs it pays off to delay that addition as much as possible (as BTC- does). The reason is that in deep graphs the descendent lists tend to be larger than in shallow graphs, e.g., the descendents of the root of a deep path are the descendents of all the nodes on the path. Thus, eagerly adding nodes into descendent lists for deep graphs fills up pages quickly and results in poor performance. For such graphs, the approach of BTC- is the best. For shallow graphs, the situation is reversed and the approach of GDFTC is the best.

7. COMPUTING THE TRANSITIVE CLOSURE OF GRAPHS WITH CYCLES

In this section, we present the results of our performance evaluation of the studied algorithms on graphs with cycles. Again, the first section presents the results for a typical example, whereas the remaining sections elaborate on how the various parameters of the algorithm implementations and the graphs affect performance.

7.1 General Trends

Table V shows a typical example of the number of I/O operations and CPU time (in seconds) of all the algorithms. The only difference here from the typical case presented for acyclic graphs is that the buffer pool size is much smaller: $M = 10$. The reason is that for larger sizes, BTC is at least an order of magnitude faster than the other algorithms, and such a large difference can be misleading. Thus, the specific setup for the results presented has $B = 15$ and $M = 10$, whereas the corresponding graph parameters are $N = 2000$, $b = 5$. Also, although the graphs were generated with the same value of $l = 2000$ as before, the resulting depths were approximately $d = 1390$, which is much larger than in the acyclic case. This also had the effect that the size of the complete transitive closure was much higher, containing approximately

Table V. Performance Results on Graphs with Cycles with $N = 2000$, $b = 5$, and $d = 1390$

Algorithm	Number of I/O	I/O Ratio over winner	CPU Time (sec)	CPU Time Ratio over winner
BTC	4321	1.00	25.1	1.15
BTC-	6745	1.56	25.3	1.16
GDFTC	5640	1.31	38.4	1.76
Schmitz	6661	1.54	21.8	1.00

3946000 arcs. In spite of the extremity of the values of some of these parameters, the example is typical of the relative performance of the algorithms on graphs with cycles.

As for acyclic graphs, BTC is the clear winner with respect to I/O, for the same reasons. Unlike for acyclic graphs, however, there are significant differences among the remaining three algorithms, BTC-, Schmitz, and GDFTC, with each performing better than the others for some setups and graphs. The details of these differences in behavior are discussed in the individual sections that deal with the effect of the various parameters on performance.

With respect to CPU time, cyclic graphs behave differently from acyclic ones. For cyclic graphs, Schmitz is always the best performer, with the two versions of BTC being slightly more expensive, and GDFTC being significantly more expensive. The reason for the latter is the excessive stack manipulations that are necessary in GDFTC when it identifies strong components in a piece-meal fashion. All algorithms, however, are heavily I/O bound for cyclic graphs as well. Hence, most of the forthcoming discussions are concerned with I/O cost only.

7.2 Effect of Parameters of Algorithm Implementations

The effect of the buffer replacement policy and the blocking factor on the performance of algorithms is not discussed in this section, because the conclusions for cyclic graphs and their explanations are exactly the same as for acyclic ones. Hence, we concentrate on the effect of the number of buffer pages M.

Figure 7 shows a typical example of the I/O cost of the examined algorithms applied on graphs with cycles as a function of the number of available buffers M. The specific graphs have the same characteristics as those in Section 7.1, and therefore, the maxima exhibited by each algorithm in Figure 7 correspond to the results presented in Table V.

Most conclusions that can be drawn from Figure 7 are not very different from what holds for acyclic graphs. The only noteworthy observation that is different for graphs with cycles is that GDFTC benefits much less from an increase in the buffer pool size than Schmitz. This is due to the extensive stack manipulations that GDFTC performs, which occasionally result in several stack frames being brought into memory for merging. Avoiding excessive I/O for such operations requires much more memory than the corresponding operations in Schmitz, which are the consecutive additions of

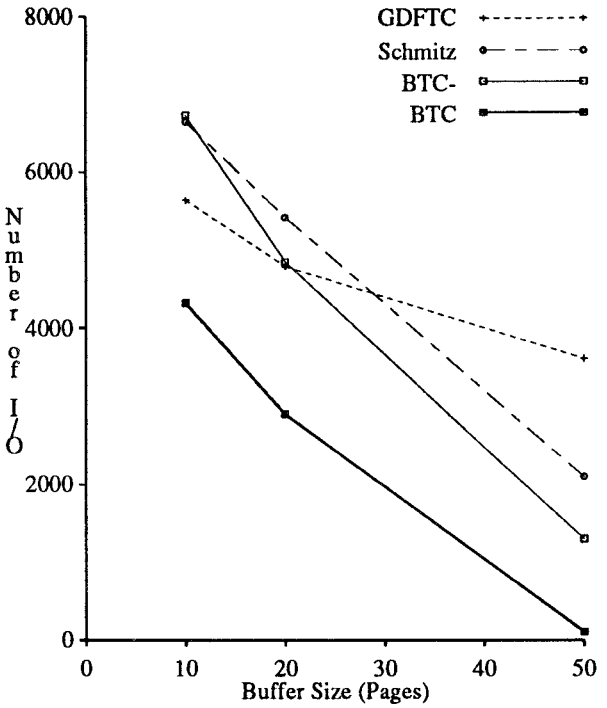


Fig. 7. I/O cost as a function of buffer size when the algorithms are applied on graphs with cycles.

nodes to the descendent list of the root of a strong component after that is discovered.

7.3 Effect of Parameters of Data

As for acyclic graphs, the number of nodes does not affect the main characteristics of the behavior of any of the algorithms when applied on acyclic graphs, and so we concentrate on b and d. The results presented in this section are for a setup that has $B = 15$ and $M = 10$ and for graphs that have $N = 2000$ and outdegree that varies from $b = 1$ to 5. As b grows, however, so does the depth d (from 66 to 1389) and the corresponding transitive closure size (from 109000 arcs to 3946000 arcs). This is an important difference between acyclic graphs and graphs with cycles, and prevents us from treating the outdegree and depth independently in this case.

The effect of b (and indirectly of d as well) on the behavior of the algorithms is shown in Figure 8. There are several very interesting observations to be made. First, with respect to CPU cost, Schmitz is always the cheapest algorithm, whereas GDFTC is always much more expensive than all others, for reasons explained in Section 7.1. Second, with respect to I/O cost, BTC remains superior to all other algorithms, as with acyclic graphs, for the same reasons.

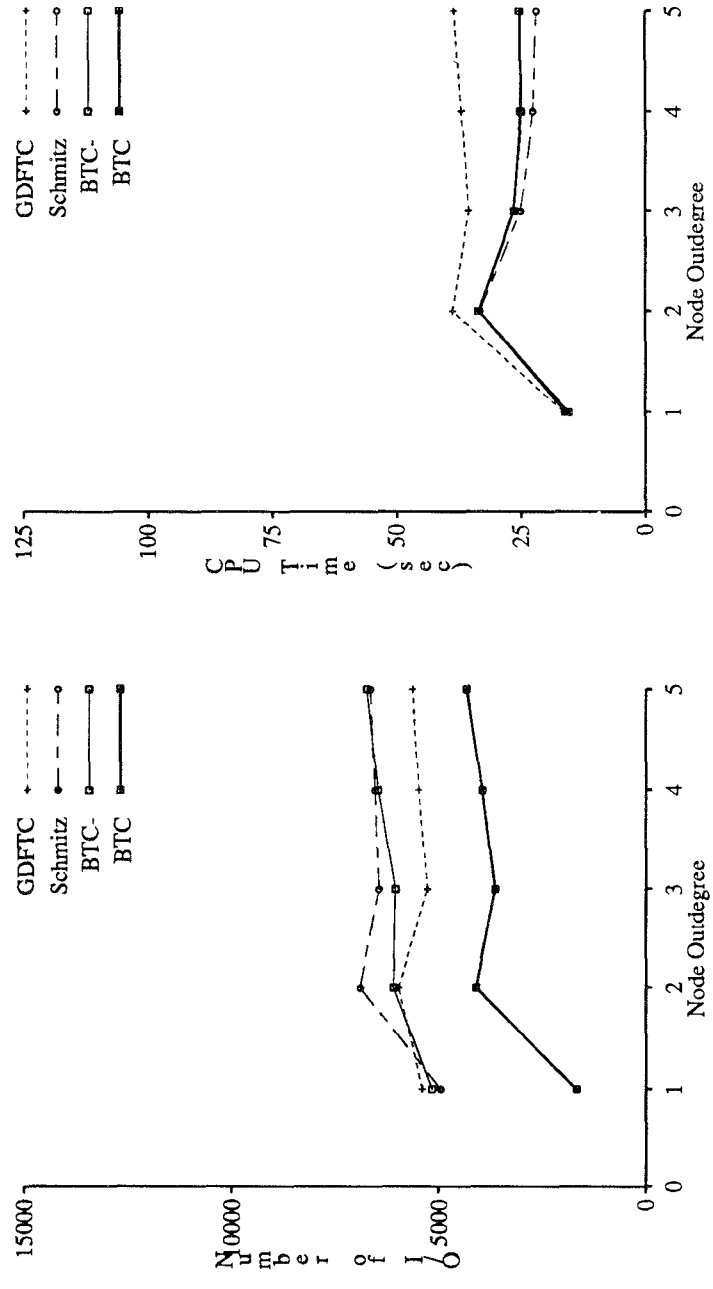


Fig. 8. I/O and CPU cost as a function of outdegree when the algorithms are applied on graphs with cycles.

Table VI. Number of Strong Components and their Sizes for Graphs with Cycles having $N = 2000$

Outdegree	Number of Strong Components	Size of Strong Components (Number)			
1	1953	41 (1)	6 (1)	3 (1)	1 (1950)
2	531	1470 (1)	1 (530)		
3	158	1843 (1)	1 (157)		
4	61	1940 (1)	1 (60)		
5	26	1975 (1)	1 (25)		

Third, all algorithms present a local maximum in their cost for $b = 2$. This can be explained based on random graph theory and the formation of strong components. As a point of reference, the number of strong components and their sizes are given in Table VI for a representative graph among those tested in our experiments for each value of b.

As the outdegree of nodes increases, very quickly most of them become part of a single huge strong component, whereas the remaining nodes are scattered among very small components. It is quite clear from Table VI that a large strong component is already formed when $b = 2$. In addition, the size of that component has a big increase when moving from graphs with $b = 2$ to $b = 3$, whereas beyond that point it absorbs the remaining nodes relatively slowly. Since all algorithms generate a single descendent list for each strong component, which is later propagated to all the nodes in it, the significant drop in the number of the components between $b = 2$ and $b = 3$ results in a drop in the overall I/O cost. The even more dramatic increase in the large strong component size experienced between $b = 1$ and $b = 2$ does not have the same effect, because graphs with $b = 1$ tend to have many strong components that are mutually disconnected, e.g., at least all nontrivial strong components. Therefore, for such graphs, descendent lists tend to be small and processing takes a small amount of time. On the other hand, beyond $b = 3$, since the increase in the size of the large strong component is minimal, the I/O cost increases simply because of the increase in the number of arcs that must be manipulated.

Fourth, as b grows, the first pass cost of BTC and BTC- becomes quite high because of the associated growth in d. With deep graphs and only 10 buffers, tuples of nodes visited early in the depth-first traversal are paged out very often, and this results in a poor buffer hit ratio. Fifth, for the small buffer size that was used for these experiments, in general GDFTC is less expensive than Schmitz, despite the fact that most of the tested graphs are very deep. Apparently the cost paid by Schmitz of revisiting the descendent lists of all the children of the nodes in a strong component is much higher than the cost paid by GDFTC of merging stack frames excessively. As discussed in Section 7.2, however, for a large buffer size, the trade-off is reversed and Schmitz is less expensive.

8. PATH COMPUTATIONS

8.1 Path Algebras

We have considered how to compute the transitive closure of a graph. 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 by Carre [6], and subsequently refined by Rosenthal et al. [24] and Agrawal et al. [1], which we present below. There is a *label* L_{ij} associated with each arc (i, j). A *path* p_{ij} from node i to node j is an ordered set of arcs $\{(source_k, destination_k)\}$, $k = 1, \ldots, n$, such that $i = source_1$, $destination_1 = source_2, \ldots, destination_n = j$. We sometimes specify a path by the sequence of nodes on it. A label can be associated with the path p_{ij}. Intuitively, this *path label* is computed as a function, called CON (for *concatenate*), of the sequence of labels of the arcs in p_{ij}. A label can be associated with a *path set* P as well. This *path set label* is computed as a function, called AGG (for *aggregate*), of the path labels of the paths in P.

Formally, AGG and CON are defined as binary functions over path labels. We now introduce several conditions over AGG and CON that enable us to characterize the problems for which a path algorithm is applicable. We begin by presenting a set of core requirements that we assume to hold henceforth: CON must be associative (condition (1)), AGG must be associative and commutative (conditions (2) and (3)), and there must be identities for both CON and AGG (condition (4)). The identity for CON is denoted by θ and that for AGG is denoted by ϕ. CON is required to distribute over AGG (condition (5)). To summarize, for all labels L_1, L_2 and L_3:

(1) $\text{CON}(L_1, \text{CON}(L_2, L_3)) = \text{CON}(\text{CON}(L_1, L_2), L_3)$,
(2) $\text{AGG}(L_1, \text{AGG}(L_2, L_3)) = \text{AGG}(\text{AGG}(L_1, L_2), L_3)$,
(3) $\text{AGG}(L_1, L_2) = \text{AGG}(L_2, L_1)$,
(4) $\text{AGG}(\phi, L_1) = L_1$; $\text{CON}(\theta, L_1) = \text{CON}(L_1, \theta) = L_1$, and
(5) $\text{AGG}(\text{CON}(L_1, L_2), \text{CON}(L_1, L_3)) = \text{CON}(L_1, \text{AGG}(L_2, L_3))$.

Conditions (1)–(4) allow us to view AGG and CON as functions over multisets of path labels and sequences of path labels, respectively. The motivation behind condition (5), proposed by Agrawal et al. [1] and Carre [6], is that it permits efficient path computations. Consider the case where there are two paths p_1 and p_2 from node j to node k with associated labels L_1 and L_2, respectively. Let there also be a path p_3 from node i to node j with label L. Let P 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 paths $p_3 \bullet p_1$ and $p_3 \bullet p_2$. In particular, suppose that AGG is applied to p_1 and p_2, and the label $L' = \text{AGG}(L_1, L_2)$ is simply recorded with the ordered pair (j, k), instead of L_1 and L_2. In this case, instead of both paths from i to k, only a single path is generated, with the label $\text{CON}(L, L')$. If AGG and CON are distributive, then this is indeed the correct path set label for the set of paths from i to k.

When considering cyclic graphs, we will require *absorptiveness*, which is an additional condition that allows us to ignore cyclic paths. Absorptiveness is defined as

(6) $\mathrm{AGG}(L_1, \theta) = \theta$.

Dar has points out that the above six conditions together imply idempotence of AGG [8], which is defined as

$$\mathrm{AGG}(L_1, L_1) = L_1.$$

Thus, when (6) holds, we can view AGG as a function over sets of path labels rather than as a function over multisets. Finally, we also introduce the following condition, called *choice*, which enables the marking technique to be used to further optimize the computation:

(7) For every pair of labels $L_1, L_2, \mathrm{AGG}(L_1, L_2) = L_1$ or L_2.

As the following example due to Dar illustrates [8], condition (7) does not follow from (1)–(6). Consider the domain of (positive) natural numbers augmented with the special value ∞, with CON = Multiply, AGG = GCD, $\theta = 1$ and $\phi = \infty$. It is straightforward to verify that this algebra satisfies conditions (1)–(6), but not (7).

Conditions (1)–(7) form the entire framework with which we study path computations. We should note that Cruz and Norvell [7] have independently proposed a different framework for the same purpose. Dar has noted that conditions (1)–(7) above are equivalent to their notion of maximizing semirings [8].

Table VII shows how several familiar problems can be described in this terminology [1, 7]. The path set in each of the above definitions is the set of all paths in the graph. Note that Critical Path and Bill of Materials do not have absorptive AGG functions; the latter does not even have an idempotent AGG. However, these problems are typically ill-defined over cyclic graphs.

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 (i, j). Let A^k denote the kth power of A. Then, L^k_{ij} 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, $A^k = A^{k+1}$. If $\langle G, \mathrm{AGG}, \mathrm{CON}\rangle$ is absorptive, then (the adjacency matrix associated with) it is stable [6].

8.2 Algorithm Path_BTC

The distributive property is easily exploited to generalize transitive closure algorithms presented earlier to also do path computations. First, we must augment the descendent set representation to also record labels. Consider descendent 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 descendent lists denotes the graph G, this is the label associated with the arc (i, j) in G. 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 algorithms. Algorithms GDFTC and Schmitz cannot be used, however, since they lose

Table VII. Formulation of Several Problems as Path Computations

Problem	Set of Labels	CON	AGG	θ	ϕ
Reachability	{0,1}	AND	OR	1	0
Shortest Path	$\mathbb{R}^+ \cup \{\infty\}$	Add	Min	0	∞
Critical (Longest) Path	$\mathbb{R} \cup \{-\infty,\infty\}$	Add	Max	0	$-\infty$
Maximum Capacity Path	$\mathbb{R}^+ \cup \{\infty\}$	Min	Max	∞	0
Most Reliable Path	[0,1]	Multiply	Max	1	0
Bill of Materials	$\mathbb{R}^+ \cup \{\infty\}$	Multiply	Add	1	0

path information in strong components; similarly, the root optimization is not applicable for algorithm BTC, and thus we must consider BTC′ instead.

We present algorithm Path_BTC, which is a straightforward adaptation of BTC′ to do path computations. 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 label ϕ. One obvious difference in the path version of BTC is that we now have to compute path labels as we add descendent sets. Another, much more important difference, however, stems from the fact that each node in a descendent set now carries a label. Thus, we must be concerned with not just the membership of a node in a descendent set, but also whether the value of the associated label changes with a given descendent set addition. (The addition of a node to a descendent set for the first time can be thought of as a special case that promotes the label from the worst-case value ϕ, which intuitively denotes nonreachability.) The difference can be understood in terms of the marking technique. If node k appears in a descendent list S_i, let us refer to the addition of all children of k to S_i as a *one-step expansion* (or just *expansion*) of k in S_i. For reachability, if node k appeared marked in S_i, this meant that a one-step expansion of k had been carried out in S_i. For path computations, it is possible that sometime after such a one-step expansion is carried out, a better value is found for k in S_i. Specifically, we could add descendent set S_j, containing node k, to S_i, also containing j. The new label, which takes into account the path from i to k through j, could be different. When this happens, we have to examine the children of k again to see if their labels (in S_i) also change, due to the new label for k. This means that k should be *unmarked* in S_i when we compute a different label for it.

However, if k was marked in S_j, it can be marked in S_i after the addition as well, provided that condition (7) is also satisfied. Intuitively, this condition ensures that AGG essentially just keeps the "best" label values; since the label changed, the label $\text{CON}(L_{ij}, L_{jk})$ is the "best" label found so far from i to k. For example, if we are computing shortest paths, AGG just retains the shortest path; all other path label information is irrelevant. Since k is marked in S_j, the children of k are in S_j, and therefore, the labels computed for them in the addition of S_j to S_i take into account the new "best" label for the path from i to k via j. The labels, however, that are associated with old paths from i to k are not explicitly considered; thus, k cannot be treated as marked in S_i unless condition (7), which allows us to consider just the "best" path, holds.

The complete algorithm, including the optimization that takes advantage of condition (7), is given below.

proc Path_BTC(G)
Input: A graph G represented by children sets E_i, $i = 1$ to n, with labels.
Output: $S_i = M_i \cup U_i$ ($U_i = \emptyset$ or $\{i\}$), $i = 1$ to n, denoting G^*, with correct path labels.

```
(1)  {Modified_Tarjan(G);
(2)  for i = 1 to n do U_i := E_i; M_i := ∅ od
(3)  for i = 1 to n do
(4)    I := node_popped(i);
(5)    while there is a node j ∈ U_I − {I} do
(6)      M_I := M_I ∪ {j}; U_I := U_I − {j};
(7)      for all k ∈ S_j − {j} do
(8)        L_Ik^old := L_Ik; L_Ik := AGG(L_Ik, CON(L_Ij, L_jk));
(9)        if L_Ik^old ≠ L_Ik
(10)         then if k ∈ M_j then {M_I := M_I ∪ {k}; U_I := U_I − {k}}
(11)                         else {U_I := U_I ∪ {k}; M_I := M_I − {k}}
         od
(12)   od
(13) od}
```

We use the following notation. If P is some path (set of paths), $L(P)$ denotes the set of associated path label(s). L_{ik} denotes the label associated with arc (i, k), and L^1_{ik} denotes the initial value of this label (i.e., the label for the arc from i to k). We begin by establishing an important property of nodes in a marked set.

LEMMA 8.1. *If* $\langle G, AGG, CON \rangle$ *satisfies conditions* (1)–(7), *the following invariant holds after each iteration of the* **while** *loop of Path_BTC: if* $k \in M_I$, *then for each* $l \in E_k$, $L_{Il} = AGG(L)$, *where* L *is a set of path labels that includes the label* $CON(L_{Ik}, L^1_{kl})$.

PROOF. The **while** loop is iterated several times for each value of I, in general. We number iterations of the **while** loop over the entire execution of the algorithm. For example, if there are five iterations for I = popped(1), the first iteration for I = popped(2) is iteration number 6. The proof is by induction over the iteration number.

Basis. Initially, M_i is empty, for all i, and so the claim holds trivially.

Induction Step. Suppose that the claim holds after iteration T-1. We show that the claim is preserved after iteration T. There are two ways in which a node can be added to M_I in iteration T. First, the node is the child j of I being processed in iteration T. In this case, the execution of the **for** loop ensures that the lemma holds with respect to node j, since for each child k of j, L_{Ik} is updated to reflect $\text{CON}(L_{Ij}, L_{jk})$. In particular, assume that the old value of label L_{Ik} is equal to AGG(L) for some set of labels L. Then, from conditions (2)–(4) and (6), which allow us to treat AGG as a function over sets of labels, statement (8) ensures that the new value of L_{Ik} is equal to

$AGG(L \cup \{CON(L_{Ij}, L_{jk})\})$. Since L_{jk} was initially $AGG(\{L^1_{jk}\}, \phi)$, this further implies that $L_{Ik} = AGG(L')$ where L' includes $CON(L_{Ij}, L^1_{jk})$.

The second way in which a node can be added to M_I is if a node k is found in M_j such that there is a change in the value computed for L_{Ik} in statement (8). We must show that for every child l of k, $L_{IE} = AGG(L)$ where L is a set of path labels that includes $CON(L_{Ik}, L^1_{kl})$. We observe that since L_{jl} cannot be altered in the current iteration, and k is in M_j, by the induction hypothesis, $L_{jl} = AGG(L')$, where L' includes $CON(L_{jk}, L^1_{kl})$. Further, since $l \in S_j$, after iteration T, by statement (8), L_{Il} will be equal to $AGG(L'')$, where L'' includes the label $CON(L_{Ij}, L_{jl})$. Since AGG can be viewed as a function over sets of labels, by conditions (2)–(4) and (6), this implies that L'' includes the label $CON(L_{Ij}, CON(L_{jk}, L^1_{kl}))$, and thus, by condition (1), the label $CON(CON(L_{Ij}, L_{jk}), L^1_{kl})$. As we showed in the previous paragraph, $L_{Ik} = AGG(L \cup CON(L_{Ij}, L_{jk}))$. Further, this value is different from $AGG(L)$. By condition (7), it follows that $L_{Ik} = CON(L_{Ij}, L_{jk})$. Therefore, L'' includes the label $CON(L_{Ik}, L^1_{kl})$. This concludes the proof of the lemma. □

THEOREM 8.2. *If $\langle G, AGG, CON \rangle$ satisfies conditions* (1)–(7), *algorithm Path_BTC terminates with $M_i \cup U_i$ equal to the set of all descendents of i, and $L_{ij} = AGG(L(P_{ij}))$, where P_{ij} is the set of paths from i to j, for all pairs of nodes i, j.*

PROOF. The algorithm terminates, since by the conditions on statement (5) and (7), cyclic paths are never extended. Further, when the algorithm terminates, for all i, the only node in U_i, if any, is node i. It remains to be shown that, for any pair of nodes i and k, $L_{ik} = AGG(L)$, where L is the set of labels of all paths from i to k. Since we assume that $\langle G, AGG, CON \rangle$ is absorptive, condition (6), we only need to show that L includes the labels of all acyclic paths.

Let D be a parameter that represents path lengths. We prove by induction on D that for any pair of nodes i and k, $L_{ik} = AGG(L)$, where L is the set of labels of all paths from i to k of length less than or equal to D.

Basis. For $D = 1$, we need only consider arcs in the input graph, and the claim holds trivially.

Induction Step. Let the claim hold for all paths of length less than D. Consider a path p of length D from i to k; without loss of generality, let (j, k) be the last arc on the path p. From Lemma 8.1, since $j \in M_i$ when the algorithm terminates, $L_{ik} = AGG(L')$, where L' includes $CON(L_{ij}, L^1_{jk})$. By the induction hypothesis, $L_{ij} = AGG(L'')$, where L'' is the set of labels of all paths from i to j of length less than D. In particular, it includes the label $L(\{p_{ij}\})$, where p_{ij} is the prefix of path p from i to j. From the properties of AGG and CON, it follows that L' includes $CON(L(\{p_{ij}\}), L^1_{jk})$, which is the label for path p. □

Note that algorithm Path_BTC 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 is also possible to compute the path set label for the set

of all paths in the transitive closure. Thus, for example, we can find the longest path in a graph. (In this case, CON would be concatenation and AGG would be the function that picks the longer path.)

We also remark that the following condition, called *pairwise label extensibility* (or just *extensibility*), can be used to derive condition (7), given conditions (1)–(6):

(7′) For every pair of labels L_1, L_2, there exists some label L_3 such that $L_1 = \text{CON}(L_2, L_3)$ or $L_2 = \text{CON}(L_1, L_3)$.

Indeed, substituting $L_3 = \theta$ in condition (5) and simplifying by using conditions (4) and (6) yields, for all L_1, L_2, the following:

$$\text{AGG}(\text{CON}(L_1, L_2), L_1) = L_1.$$

From condition (7′), for any L_1 there is some L_3 such that $L_1 = \text{CON}(L_2, L_3)$ (or symmetrically $L_2 = \text{CON}(L_1, L_3)$). Therefore, the following holds:

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

Extensibility appears to be a simple property, independent of AGG, and is satisfied by all the problems listed in Table VII. Condition (6), however, does not hold for Critical Path and Bill of Materials; therefore, extensibility cannot be used to establish condition (7), choice, for these problems. As it happens, condition (7) does not hold for Bill of Materials, but it does hold for Critical Path. We note that extensibility does not follow from conditions (1)–(7).

We now present a variant of Path_BTC that works correctly even when condition (7) does not hold, although it is less efficient than Path_BTC when condition (7) also holds. The modification to Path_BTC is as follows. Lines (9)–(11) should be replaced by the following:

$$\textbf{if } L_{Ik}^{old} \neq L_{Ik} \textbf{ then } \{U_I := U_I \cup \{k\};\ M_I := M_I - \{k\}\}$$

This variant of Path_BTC in fact works even if condition (6) does not hold as long as the data is acyclic.

THEOREM 8.3. *If* $\langle G, AGG, CON \rangle$ *satisfies conditions* (1)–(6), *or conditions* (1)–(5) *and* G *is acyclic, algorithm Path_BTC with the modification described above terminates with* $M_i \cup U_i$ *equal to the set of all descendents of* i, *and* $L_{ij} = AGG(L(P_{ij}))$, *where* P_{ij} *is the set of paths from* i *to* j, *for all pairs of nodes* i, j.

PROOF OUTLINE. The proof is similar to that of Theorem 8.3; however, we need to prove a lemma equivalent to Lemma 8.1, but without assuming conditions (6) for acyclic graphs and (7) in general. The proof of such a lemma is identical to the proof of Lemma 8.1, with the following additional observations: (a) The proof of Lemma 8.1 dealt with two ways in which a node can be added to M_I in iteration T. A node can now be added in only the first way discussed in the proof of Lemma 8.1, given the modification to lines (9)–(11) of the algorithm. (b) Condition (7) was used in the proof of Lemma 8.1 only in

the case that a node was added in the second way discussed there. (c) For acyclic graphs, since condition (6) may not hold, idempotence may not hold either, so we have to treat AGG as a function over multisets of labels; associated with a path set, we now have a multiset of labels, with the cardinality of a label equal to the number of times it appears as a label of a path in the path set. Examining the proof of the lemma, we see that essentially the same argument as before still serves. The key observation is that each path from a node I to a node k is constructed in exactly one way—by extending some path from I to a node j with an arc from j to k. The label for this path is considered when the path is constructed. If this results in a new label for the set of paths from I to k, then extensions of this path obtained by appending some arc going out of k must be considered; therefore, k must be unmarked in S_i, and this is indeed done, by modified line (9). If the label of the new path from I to k does not change the label of the set of paths from I to k, by condition (5), we need not reconsider paths obtained by extending the new path from I to k.

The proof of the theorem itself is again similar to the corresponding proof in Theorem 8.2. The only additional observation is observation (c) above. □

We now turn to the behavior of Algorithm Path_BTC on acyclic graphs. Since condition (6) was only used to exclude cyclic paths from consideration, it is easy to see that it is not required for acyclic graphs. In that case, idempotence may not hold either, but this was only simplified proofs by allowing us to treat AGG as a function over sets. If condition (7) holds, however, AGG can still be treated as a set since a single label is always selected when we apply AGG; the cardinality of labels in the input to AGG does not matter. Thus, we have the following corollary of Theorem 8.2:

COROLLARY 8.4. *If* $\langle G, AGG, CON \rangle$ *satisfies conditions* (1)–(5) *and* (7), *on acyclic graphs algorithm Path_BTC terminates with* $M_i \cup U_i$ *equal to the set of all descendents of* i, *and* $L_{ij} = AGG(L(P_{ij}))$, *where* P_{ij} *is the set of paths from* i *to* j, *for all pairs of nodes* i, j.

The above result allows us to apply Path_BTC to compute critical paths over acyclic graphs. However, it is not applicable to problems like Bill of Materials, even on acyclic graphs. We must use Path_BTC with the modifications described above for lines (9)–(11) of the algorithm.

8.3 Algorithm Path_Dag_DFTC

We can also adapt algorithm Dag_DFTC to do path computations. We will only consider the extension under the assumption that conditions (1)–(5) and (7) hold. Since we consider only acyclic graphs, the difference with respect to Dag_DFTC can be explained very simply. If $E_i - S_i$ contains node k, it can be moved to S_i through the addition of S_j to S_i only if $k \in S_j$ *and* the new label for k in S_i is better than the old label. The reason is that, for path computations, if $k \in S_i$, the children of k also appear in S_i and their labels reflect the path from i to k that resulted in the latest change to the label of k in S_i. (Recall that we discussed this point in the previous section as we

developed algorithm Path_BTC.) In terms of the marking technique, we note that, as in the reachability case, all nodes in S_j are marked, with marking interpreted as for path computations. It is straightforward to establish that this algorithm computes the transitive closure of G with correct labels; we omit a formal claim and proof of correctness.

proc Path_Dag_DFTC(G)
INput: A graph G represented by children sets E_i, $i = 1$ to n, with labels.
Output: S_i, $i = 1$ to n, denoting G^*, with correct path labels.

(1) {**for** $I = 1$ to n **do** *visited*$[i] := 0$; $S_i := \emptyset$ **od**
(2) **while** there is some node i s.t. *visited*$[i] = 0$ **do** visit(i) **od**
}
(3) **proc** visit(i)
(4) {*visited*$[i] := 1$;
(5) **while** there is some $j \in E_i - S_i$ **do**
(6) **if** *visited*$[j] = 0$ **then** visit(j);
(7) **for** all $k \in S_j$ **do**
(8) $L_{ik}^{old} := L_{ik}$; $L_{ik} := \text{AGG}(L_{ik}, \text{CON}(L_{ij}, L_{jk}))$;
(9) **if** $L_{ik}^{old} \neq L_{ik}$ **then** $S_i := S_i \cup \{k\}$
od
od
}

We note that since algorithms Dag_DFTC and GDFTC behave identically on acyclic graphs, the above path algorithm can be seen as an adaptation of both of these algorithms.

8.4 Algorithm Path_Dag_Schmitz

Like GDFTC, the Schmitz algorithm loses path information in strong components and cannot be adapted to do path computations on cyclic graphs. Also, recall that the Schmitz algorithm contains an optimization that is essentially equivalent to marking for acyclic graphs. Below we present Path_Dag_Schmitz, which is an adaptation of Schmitz that works for acyclic graphs, under the assumption that conditions (1)–(5) and (7) hold. We observe that this algorithm differs from Path_Dag_DFTC in only the following minor respect: all children of a node are visited before any of their descendent lists is added to the parent's list. Again, since it is straightforward to establish that this algorithm computes the transitive closure of G with correct labels, we omit a formal claim and proof of correctness.

proc Path_Dag_Schmitz(G)
Input: A graph G represented by children sets E_i, $i = 1$ to n, with labels.
Output: S_i, $i = 1$ to n, denoting G^*, with correct path labels.

(1) {**for** $i = 1$ to n **do** *visited*$[i] := 0$; $S_i := \emptyset$ **od**
(2) **while** there is some node i s.t. *visited*$[i] = 0$ **do** visit(i) **od**
}
(3) **proc** visit(i)
(4) {*visited*$[i] := 1$;

```
(5) for all j ∈ E_i do if visited[j] = 0 then visit(j) od
(6) while there is some j ∈ E_i − S_i do
(7)   for all k ∈ S_j do
(8)     L_ik^old := L_ik; L_ik := AGG(L_ik, CON(L_ij, L_jk));
(9)     if L_ik^old ≠ L_ik then S_i := S_i ∪ {k}
      od
    od
    }
```

9. PERFORMANCE EVALUATION OF ALGORITHMS FOR PATH COMPUTATIONS

Several algorithms for path computations have been proposed in the literature, e.g., [1, 7, 15]. We have not attempted to conduct a comprehensive performance evaluation of all these algorithms. Instead, we have studied the performance of the algorithms presented in the previous section, so that we can evaluate how the original reachability algorithms behave when adapted for path computations.

9.1 Implementation and Performance Evaluation Testbed

We have implemented Path_BTC, Path_Dag_Schmitz, and Path_Dag_DFTC for path computations on acyclic graphs. Path_BTC- has also been implemented so that the effect of inter- and intradescendent list orderings on performance can be measured. Below, we present the main results of our experiments, focusing on the issues where there are differences between path computations and reachability. These results show that, in this case as well, Path_BTC dominates all other algorithms in terms of performance. For this reason, we have implemented only Path_BTC and Path_BTC- for path computations on graphs with cycles, since the former appears to be the algorithm of choice for the whole range of computations. Some results on the performance of these algorithms are presented at the end of this section as well. We have not implemented the variant of Path_BTC that does not take advantage of condition (7). However, based upon some preliminary experiments with a variant of Path_BTC that utilized condition (7) only partially, we expect that the improvement due to utilizing this condition is significant, perhaps resulting in a reduction of I/O by a small integer factor.

All the important features of the implementations of the algorithms for reachability have been maintained in their versions for path computations as well. These include the external and internal storage structures, the inter- and intradescendent list orderings employed at the end of the first pass of Path_BTC, and the memory management techniques. The only difference in the above is that, in all structures, every arc is associated with a label that may be updated during the course of the execution. Another important difference is that, for path computations, duplicate arcs are not eliminated, but instead their labels are compared appropriately and one of them is chosen as the optimal one for the arc. Even this process, however, takes advantage of

the adjacency vectors constructed for descendent lists and only costs O(1) time for each node in a list that is merged into another one.

In our experiments with path computations, the exact same testbed was used as with reachability. Due to the increased cost of the algorithms, however, we mostly applied the algorithms on smaller graphs than before, i.e., we extensively tested acyclic graphs with at most $N = 1000$ nodes and graphs with cycles with at most $N = 200$ nodes. Nevertheless, our limited experiments with larger graph sizes showed that our conclusions do not depend on the specific sizes chosen. For all our tests, we used the shortest path problem as our path computation. The arc labels were generated randomly from a uniform distribution in the range between 1 and 10.

9.2 Computing the Shortest Paths between Nodes of Acyclic Graphs

In this section, we present the results of our performance evaluation of the studied algorithms when computing shortest paths between nodes of acyclic graphs. Our analysis is not as detailed as for reachability, because as mentioned above, most of the conclusions for that case carry over for path computations as well. Our focus is on the effect of outdegree and depth.

9.2.1 *Outdegree.* Figure 10 shows typical examples of the number of I/O operations and CPU time (in seconds) of all the algorithms as a function of the outdegree b of the graph nodes. It corresponds to the usual setup, i.e., $B = 15$ and $M = 50$, and contains numbers for acyclic graphs with $N = 1000$ nodes and depth that varies between $d = 8$ and 13, which were generated by using $l = 1000$. As b grows from 1 to 5, the transitive closure size for these graphs grows from 6600 to 193000. For comparison, we also show the corresponding diagrams for reachability in Figure 9.

The only noteworthy difference between reachability and path computations is the significant increase in the absolute value of the I/O and CPU costs of all algorithms. This is to be expected, due to the presence of the labels that occupy a large percentage of the used space and require considerably more manipulation than the simple duplicate eliminations and list mergings of reachability. Given that the space overhead incurred by the labels is about 50%, the observed increase of about a factor of 2 in I/O cost was to be expected. The relative performance of algorithms with respect to I/O remains as before. Path_BTC is the most efficient algorithm to be used for path computations, primarily due to its ability for appropriate inter- and intradescendent list orderings. The other three algorithms have almost identical I/O costs. On the other hand, the CPU cost is dominated by the label computations, and so all four algorithms are almost identical in that respect.

9.2.2 *Depth.* In this section we present the results of the same experiment as in the previous one, with the only difference that locality factor $l = 10$ was used to generate deeper graphs, i.e., with depths ranging from $d = 86$ to 90. Figure 11 presents the associated results. Most major conclusions that can be drawn are identical to the reachability case. We especially want to emphasize that the trade-offs on I/O cost between making

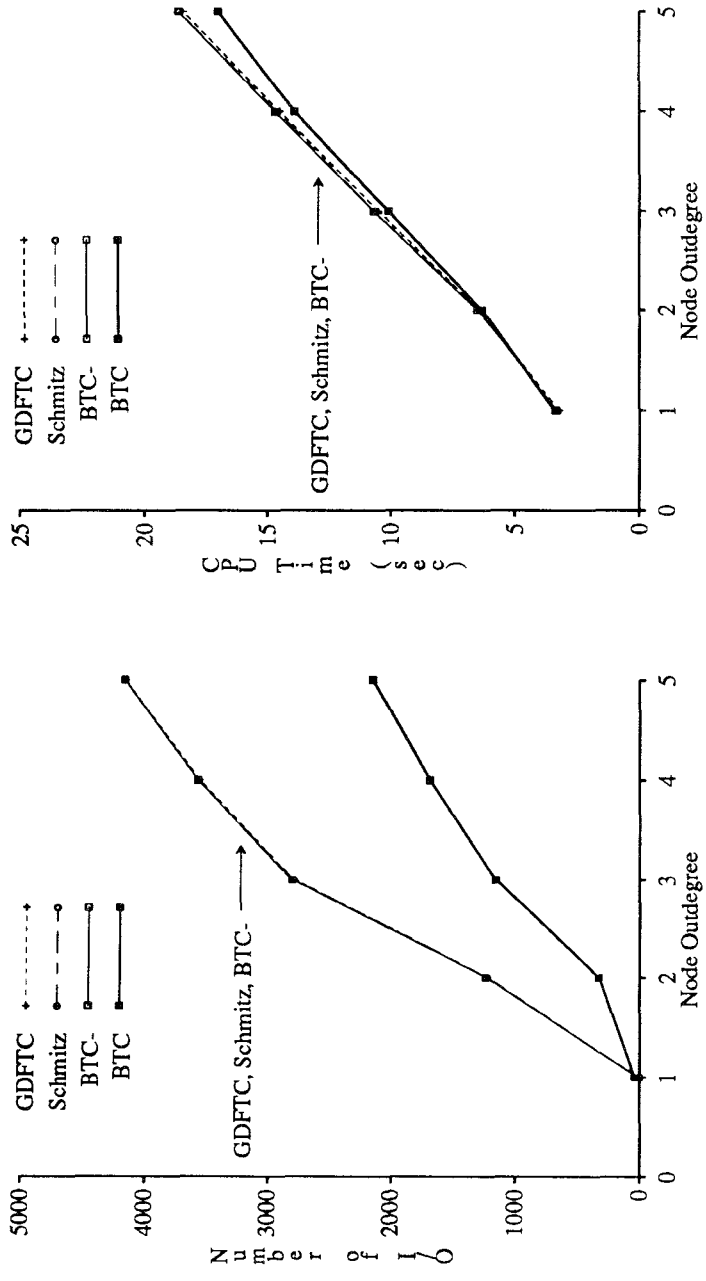


Fig. 9. I/O and CPU cost as a function of outdegree for reachability on shallow acyclic graphs.

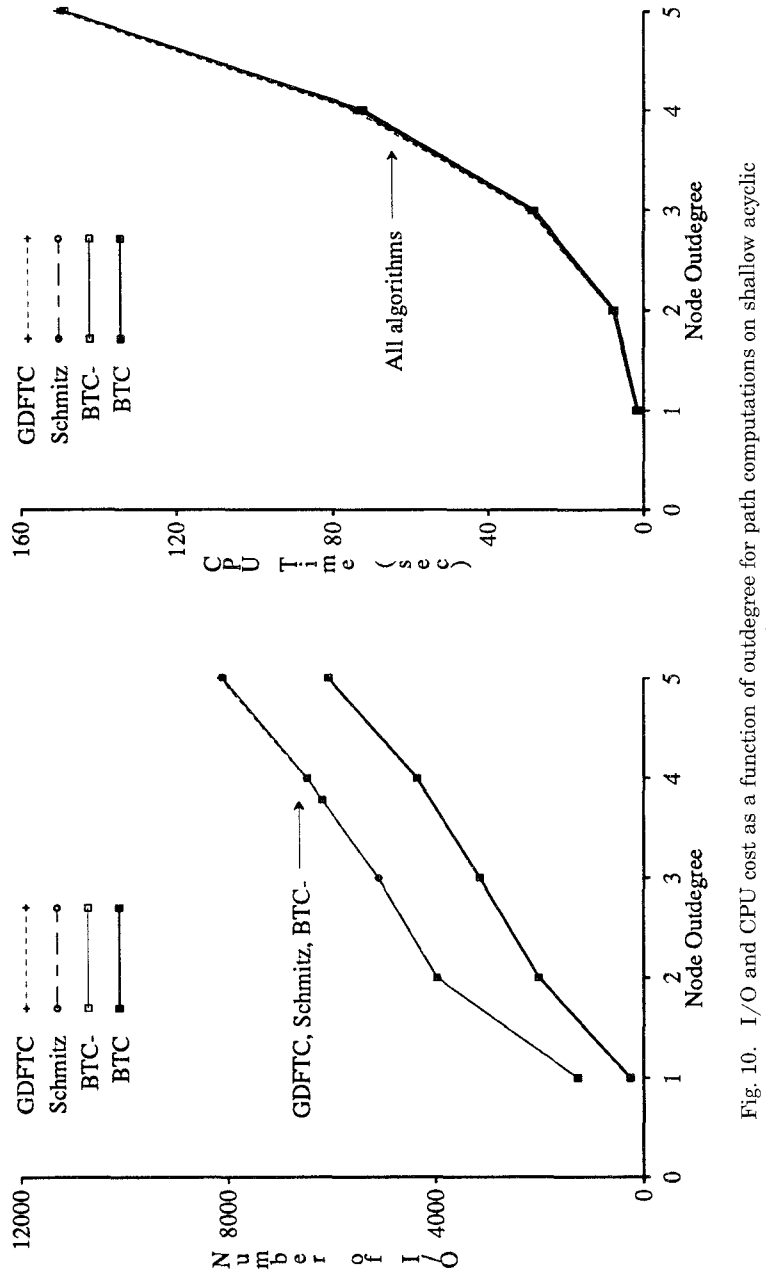


Fig. 10. I/O and CPU cost as a function of outdegree for path computations on shallow acyclic graphs.

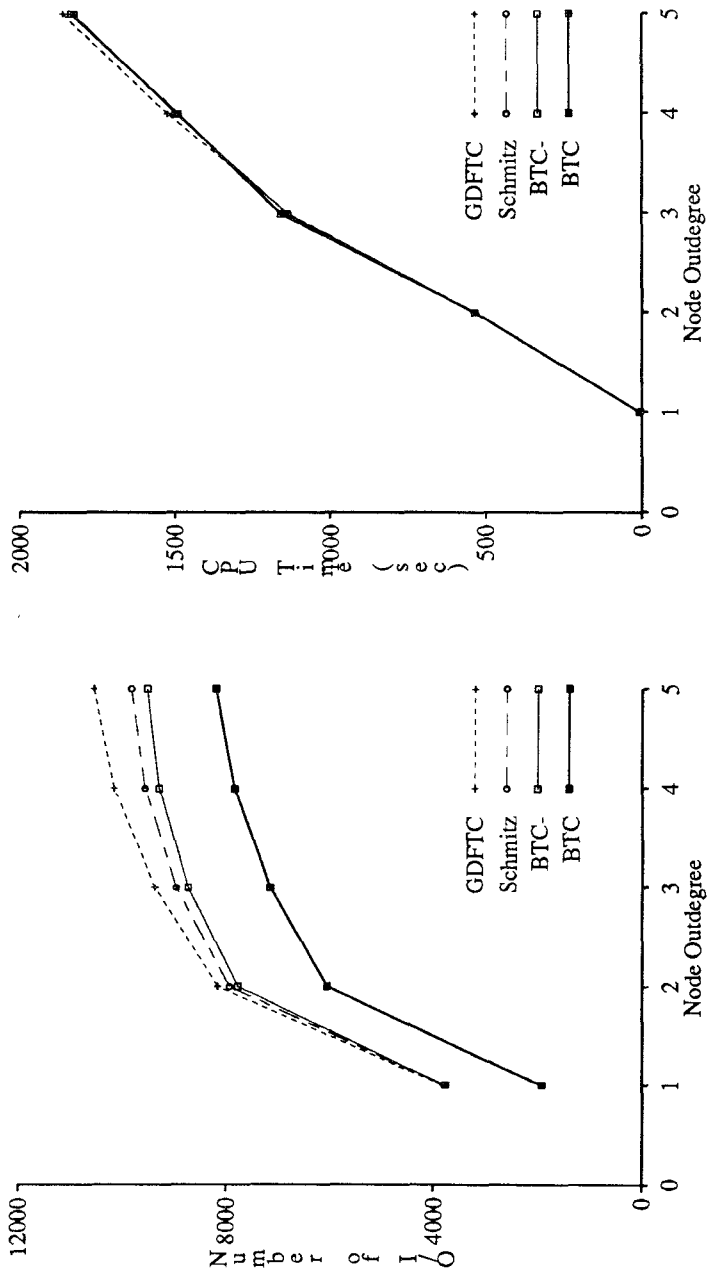


Fig. 11. I/O and CPU cost as a function of outdegree for path computations on deep acyclic graphs.

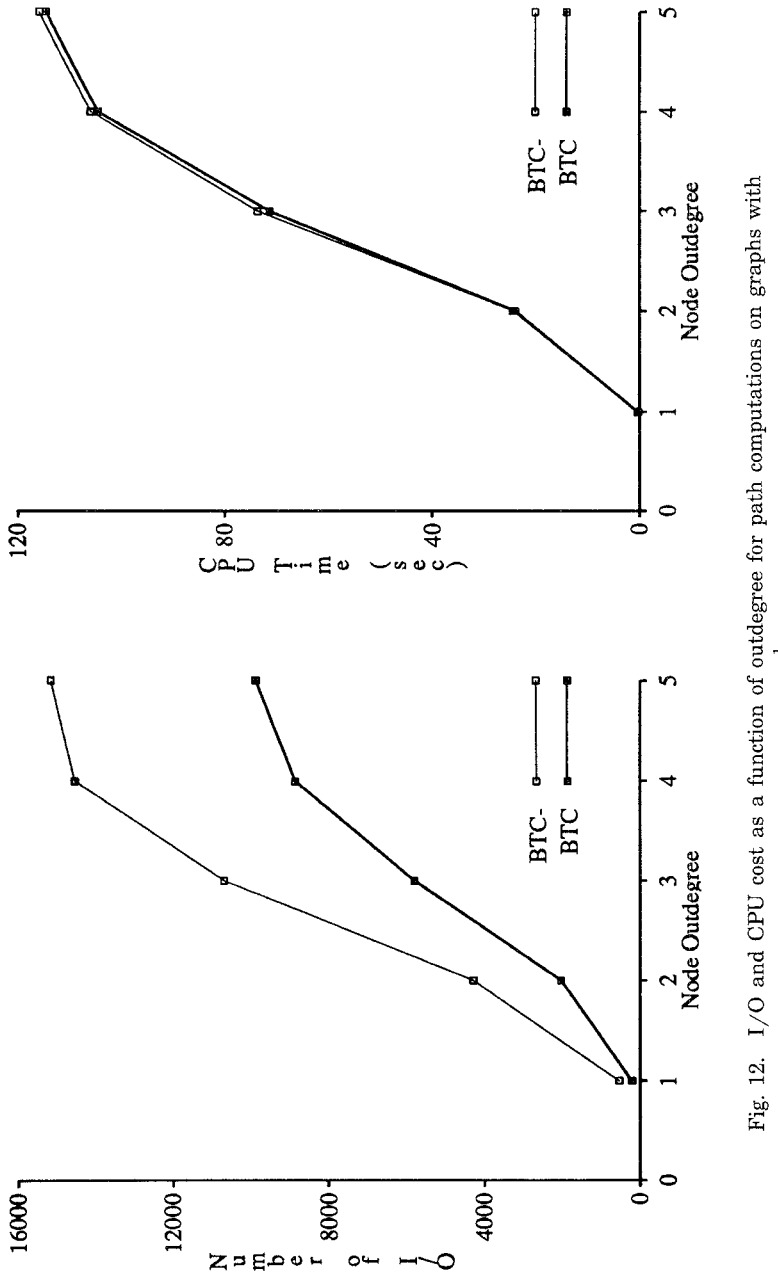


Fig. 12. I/O and CPU cost as a function of outdegree for path computations on graphs with cycles.

descendent additions eager versus lazy are exactly the same for path computations as they were for reachability. This becomes clear when one compares Path_BTC-, Path_Dag_Schmitz, and Path_Dag_DFTC in Figures 10 and 11. In the former, for relatively shallow graphs, the three algorithms have almost identical costs, indicating a balance between the two opposing trends. In the latter, for deep graphs, Path_BTC- is clearly the most efficient and Path_Dag_DFTC is the least efficient, indicating again that for the more complex graphs, lazy additions are the preferred approach. Another interesting observation is the extreme increase in the CPU cost of the algorithms, compared to the shallow graphs. The number of potential paths are so many in this case, that the label comparisons dominate the cost and the algorithms become CPU-bound.

9.3 Computing the Shortest Paths between Nodes of Graphs with Cycles

In this section we present the results of our experiments with Path_BTC and Path_BTC- on graphs with cycles. As mentioned above, due to the expense of such computations, we only present results for graphs with $N = 200$ nodes. Figure 12 shows the I/O and CPU costs of the two algorithms as a function of the node outdegree b, for a setup that has $B = 15$ and $M = 10$ and for graphs that have $N = 200$. As b grows, the depth d grows as well from 19 to 142.

There are two points that need to be emphasized about the results in Figure 12. First, the increase in the cost of the algorithms compared to reachability is dramatic. As a point of comparison, the cost of Path_BTC for the tested 200 node graphs is almost twice as high as the cost of BTC for similar graphs with 2000 nodes (Figure 8)! This is because, unlike for acyclic graphs, nodes can become unmarked, and in the worst case, every single path in the graph may need to be examined. This results in the observed increase in the cost. Second, unlike in the case of reachability, there is no local maximum in the cost presented for $b = 2$. The reason is that in path computations, the strong component optimization of Section 3.3 does not apply, and therefore, the number of strong components does not affect the cost of the algorithms. Hence, increasing the outdegree of nodes, monotonically increases the size of the graph and the number of paths in it, which results in a monotonically increasing cost as well.

10. SELECTIONS

In this section we briefly describe how our path algorithms can be adapted to compute selections for both reachability and path computations. We present the algorithms below as adaptations of Closure or BTC, our simplest algorithms. However, they can also be viewed as special cases of some other algorithms.

10.1 Reachability

When a selection of the form "column1 = c" is specified, no numbering of nodes is necessary, and so algorithm Closure can be run directly on the

original graph. Furthermore, its first loop is no longer necessary; the inner loop (line (3)) can simply start executing from the selected node c. We observe that M_j is always empty, and so marking does not yield any improvements in this case.

On the other hand, a selection of the form "column2 = c", which requests all tuples in the transitive closure of the form $(?, c)$, requires that the graph be first transformed so that it is represented using *predecessor sets*. The algorithm above can then be used.

Finally, consider a selection of the form "column1 = c_1 **and** column2 = c_2", which simply requests to test if (c_1, c_2) is in the transitive closure. This can be processed as in the case of selection "column1 = c_1", with the difference that the algorithm can stop when c_2 is added to S_{c_1}.

10.1.1 *Multisource Reachability.* A variant on the selection problem is to ask for the sets of nodes reachable from each of a given set of source nodes. (We only consider selections on the first column in this section.) If the source set is small, it is probably best to simply run the selection algorithm repeatedly, at the potential cost of some duplicate computation for subgraphs that are reachable from more than one source node. This problem has been studied by Jiang [14], who has presented an interesting algorithm that avoids duplication of effort. Algorithm BTC can also be adapted in a simple way to deal with multisource reachability: in the numbering phase, use each of the given source nodes as a root until no new nodes are encountered. This yields a topological sort of the subset of the original graph that is reachable from at least one of the given source nodes. Compute the closure on this subset; this yields the closure for each of the source nodes. There is no duplicate work, and marking does improve the performance, but the closure of every reachable node is computed. Thus, there is a tradeoff with respect to repeatedly running the single-source selection algorithm. We have not explored this tradeoff or compared these algorithms to Jiang's.

10.2 Path Computations

It is possible to specify path computation queries with selections. For example, the longest path from node i can be requested, in which case only the set of paths from node i are of interest. This is computed using essentially the inner loop of Path_BTC. We present the following algorithm:

```
proc select_path_TC(i)
{U_i := E_i; M_i := ∅;
while U_i ≠ ∅ do
  choose j ∈ U_i s.t. L_ij ≤ L_ik for all k ∈ U_i;
  M_i := M_i ∪ {j}; U_i := U_i − {j}
  for all k ∈ S_j − {j} do
    L_ik^old := L_ik; L_ik := AGG(L_ik, CON(L_ij, L_jk));
      if L_ik^old ≠ L_ik
        then {U_i := U_i ∪ {k}; M_i := M_i − {k}}
  od
od
}
```

We observe that for the special case of the single source shortest path problem, this is Dijkstra's algorithm [9]. The heuristic used in the choice of j is motivated by this algorithm. Other examples of path computations with selection include the shortest path from node i to node j and the maximum flow path from node i to node j. At first sight, it appears that, for these problems, the above algorithm can be adapted for the selection "column1 = i **and** column 2 = j", so as to terminate as soon as some path from node i to node j is found. This is incorrect, however, since even in these problems, *all* paths from i to j must be explored. Thus, we must still use the algorithm corresponding to the selection "column1 = i". We note that the above algorithm assumes that conditions (1)–(7) are satisfied, or that the graph is acyclic and conditions (1)–(5) and (7) are satisfied. If condition (7) is not satisfied, we can modify this algorithm as before, but at the cost of some efficiency. Thus, marking can yield improvements even for path computations with single-source selections.

Finally, multisource path computations can be also dealt with along the lines outlined for multisource reachability.

11. RELATED WORK

In Section 2, we reviewed a significant subset of the existing graph-based algorithms for transitive closure. In this section, we briefly discuss nongraph-based algorithm and compare them with the ones studied in this paper. In particular, we compare our algorithms with the traditional matrix-based Warshall and Warren algorithms, a hybrid algorithm proposed by Agrawal and Jagadish, and the iterative Seminaive and Smart algorithms, which are applicable for arbitrary fixpoint computations and not just for transitive closure. We also discuss some other related work on the topic.

11.1 Iterative Algorithms

The Seminaive algorithm [5] is a well-known algorithm for general recursive queries, and can be applied to transitive closure as well. Unlike graph-based algorithms, information specific to the transitive closure problem cannot be used, so several techniques that improve performance, such as the root-optimization for BTC, are not applicable. In addition, the cost of duplicate-elimination is nontrivial, since facts that correspond to entries in many descendent lists are produced in each iteration, and we may have to retrieve many lists to check if the facts produced in an iteration are already known.

The *Smart* or *Logarithmic* algorithm [12, 27] first computes all the pairs of nodes that are a number of arcs apart that is a power of 2, and then computes the remaining arcs performing much fewer operations than would otherwise be needed (i.e., if Seminaive were used). Regarding the transitive closure of trees, it has been shown that Smart outperforms Seminaive in most cases under varying assumptions about storage structures and join algorithms. On the other hand, on nontree graphs, Smart tends to generate many more duplicates than Seminaive, and usually loses in performance. The algorithm relies heavily on computing sets of arcs, so it is hard to formulate it in a way

that it can be directly compared with the algorithms presented in this paper. However, the remarks on Seminaive also apply to the Smart algorithm.

Lu proposed another algorithm for reachability that uses hash-based join techniques to compute the transitive closure of a relation [19]. Its basic structure is that of Seminaive, but it employs two interesting optimizations 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.

11.2 Matrix-Based Algorithms

A straightforward disk-based implementation of Warren's algorithm was proposed and tested against Smart [19]. 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. The main results of the analysis were that the Warren algorithm works better than Smart when there is ample main memory available and when there is a great variation in the lengths of the various paths in the graph.

Another implementation of the Warran algorithm, much better suited to disk-based data, was developed by Agrawal et al. [1, 2]. They used blocking to improve the performance and provided empirical evidence that the algorithm outperforms both Seminaive and Smart almost uniformly. An important aspect of their study was the demonstration of the importance of duplicate elimination costs; much of the difference between the iterative algorithms and their implementation of Warren's algorithm derives from the lower cost of duplicate elimination in their scheme.

11.3 Comparison of Graph-Based with Iterative and Matrix-Based Algorithms

In this section we draw some qualitative conclusions on the relative performance of graph-based algorithms when compared to iterative and matrix-based algorithms. For this we use the results of a performance evaluation study [16] that compared the last two classes of algorithms and, for the most part, confirmed the results of the work of Agrawal and Jagadish [1]. Specifically, we considered the subset of the graphs that were used in that study and could be identified by the graph parameters that we have used, i.e., N, b, and d (via l). We then ran BTC, GDFTC, and Schmitz on graphs with the same characteristics, using the same buffer sizes reported in that study. This allowed a meaningful comparison of the results of the two studies, since they both use the same hardware and system software platforms (Section 6). Table VIII shows the results of a subset of these experiments and the corresponding results from the earlier study. In particular, for each case, Table VIII shows the cost of the most efficient graph-based algorithm obtained in our experiments and the cost of the most efficient nongraph-based algorithm as reported in that study.

Table VIII. Aggregate Cost of the most Efficient Graph-Based and Nongraph-Based Algorithms on a Variety of Graphs

Graph Characteristics					Graph-Based	Non-Graph-Based	Ratio of
Type	N	b	d	M	Algorithm (I)	Algorithm (II)	(II) over (I)
Tree	4094	2	11	50	39.96	130.00	3.25
Acyclic	600	2	9	10	38.06	167.01	4.39
				20	20.58	103.05	5.01
				100	5.44	50.25	9.24
	500	4	14	50	32.74	167.51	5.11
Arbitrary	100	10	90	50	1.86	53.29	28.65
	400	10	333	150	27.84	2154.20	77.38

The nongraph-based algorithms present the output in the form of a descendent set per node. To permit a uniform comparison, we implemented the graph-based algorithms to do this as well; thus, for each node in a strong component, the output included a copy of the descendent set for the strong component rather than a pointer to it. (Of course, our implementations of these algorithms still maintain a single copy of the descendent set for a strong component during the computation of the closure.) The numbers in Table VIII reflect this; for all algorithms, the cost of reading the input and writing the output is included.

BTC was the most efficient graph-based algorithm in all these experiments, whereas each one of Blocked Warren, Seminaive, and Smart was the most efficient nongraph-based algorithm at least once. In presenting the cost of algorithms, we have followed the approach of the earlier study, so the I/O and CPU cost are combined into an *aggregate cost*, which is the sum of the CPU time plus 40 milliseconds for each disk I/O access. Thus, the numbers in Table VIII represent seconds.

From the above table, the superiority of graph-based algorithms should be clear. Although the specific numbers are for BTC, the magnitude of the difference between the two types of algorithms is far greater than the differences that we have observed among any graph-based algorithm. Note that when the buffer size increases, the relative difference between the graph-based and the nongraph-based algorithms increases as well. Also note the tremendous impact that the root optimization has in the performance of the graph-based algorithms on cyclic graphs, to the point that they become one or two orders of magnitude more efficient than the nongraph-based algorithms, which cannot make use of this optimization. Since the two studies were done independently, there may be some differences in the implementation details of the algorithms that do not allow us to take the actual numbers in Table VIII as absolutes. The magnitude of difference between the two families of algorithms, however, is far greater than what can be attributed to possible implementation details of the algorithms. Further, given that even relatively sparse cyclic graphs tend to have large strong components, the graph-based algorithms are likely to do even better if it is acceptable to

present the output with a single copy of the descendent set for each strong component. Hence, at least qualitatively, we can safely reach the conclusion that graph-based algorithms are superior to nongraph-based ones.

11.4 A Hybrid Algorithm

Recently, Agrawal and Jagadish [1] have proposed a hybrid algorithm that seeks to combine the advantages of graph-based and matrix-based algorithms. Their algorithm modifies the Warren algorithm with three important optimizations borrowed from graph-based algorithms. First, Tarjan's algorithm is run in a first pass to identify strong components and create a condensation graph. Second, only matrix elements in a row corresponding to children (of the node corresponding to that row) generate descendent set additions. Third, columns are ordered similarly to our intradescendent list sorting in the implementation of BTC. In addition, the marking technique presented in [13] (and also discussed in this paper) is adapted to the matrix-based computation, essentially achieving the same effect as in BTC. The matrix-based formulation can be seen as generalizing BTC by offering a range of blocking alternatives. Whereas a single descendent list is processed at a time in BTC, the Hybrid algorithm allows the matrix to be partitioned into blocks of rows and proceeds a block at a time. Thus, if a descendent list is fetched to be added to another list, it is added at that time to all lists in the current block as needed.

BTC is similar to the Hybrid algorithm for the case of blocksize 1. Our implementation of BTC physically clusters descendent lists in reverse topological order. While it is not clear whether this is done in the implementation of Hybrid algorithm [1], the algorithm itself allows it. However, there are some important differences between BTC and Hybrid. First, BTC does not actually construct the condensation graph. In particular, while it identifies the root of the strong component to which each node belongs, it does not actually create a descendent list for the strong component by adding the descendent lists of all nodes in the component in the first pass of BTC. Thus, some additions are deferred in BTC, relative to the construction of the condensation graph in the Hybrid algorithm. Second, as we noted earlier, the Hybrid algorithm offers many blocking alternatives that must be explored relative to the specific choice of blocksize 1 (using "block" as in the description of Hybrid) in BTC. As we saw earlier, physical clustering in topological order yields excellent buffer performance with both LRU and LUND; it would be interesting to see whether alternating the blocksize improves or degrades the performance.

Agrawal and Jagadish show that the Hybrid algorithm is superior to the Warren algorithm. They also study a pure graph-based algorithm, and conclude that it is inferior to Hybrid. While the differences in the testbed, performance methodology and metrics, and implementations differ sufficiently to make a close comparison of the results very meaningful, it appears that the results of the two papers are consistent insofar as they overlap.

11.5 Other Related Work

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 [17]. Four algorithms, two based on breadth-first traversal, and two heuristic algorithms, based on best first traversal [23], 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* [24]. Traversal recursion was formally specified using path algebras [6], 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).

12. CONCLUSIONS

We have presented and evaluated a family of graph-based transitive closure algorithms. Our results indicate that the simplest algorithm, comprised of a first pass that is essentially Tarjan's algorithm and an iterative second pass, is the most promising in terms of both simplicity and efficiency. The superior performance of this algorithm relative to other, more complicated, graph-based algorithms is surprising and is partly explained by the fact that many optimizations can be employed in its second pass to expedite it based on information collected in its first pass. Our results also suggest that graph-based algorithms are superior to nongraph-based algorithms due to their ability to use particular information about the graph structure, e.g., the form of the condensation graph.

If the graph G is updated relatively infrequently, it might be feasible to store the descendent sets according to (at least, approximately) the reverse topological order (*popped*). In this case, directly running algorithm Closure 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.

Finally, it is possible to extend BTC for larger classes of problems, such as one-sided recursions (see [13, 21]).

APPENDIX A: CORRECTNESS OF GDFTC

Before we prove the correctness of GDFTC, we prove some intermediate lemmas. In what follows, we use *top* to denote a variable that points to the top of the stack in the algorithm. Let *frame*$[i]$, for a node i, be defined as

follows:

$$frame[i] = \begin{cases} f & \text{if } i \in nodes[f] \\ n+1 & \text{otherwise} \end{cases}.$$

LEMMA A.1. *The array* frame *is well defined. That is, at any time there is at most one* f *such that* $i \in$ nodes$[f]$.

PROOF. Within the call visit(i), direct updates to *nodes* happen only at statements (11), (15), and (18). At most one of these commands will be executed, and this at most once, for any i, since control reaches these statements only when $root[i] = n + 1$. When these updates happen, $root[i]$ is updated also to something other than $n + 1$, so control never returns to statement (11), (15), or (18). The only other manipulation of *nodes* is at statement (12), which simply merges two existing lists into one. Thus, at any one time, there is at most one f such that $i \in nodes[f]$. Hence *frame* is well defined. □

LEMMA A.2. *Consider the call visit(i) in Algorithm GDFTC. If* $j \in E_i$, *then the following hold.*

(1) $visited[j] = 0$ *at statement* (7) *if and only if* (i, j) *is a tree arc.*
(2) $visited[j] = 1$ *at statement* (7) *and* $popped[j] = 0$ *at statement* (13) *if and only if* (i, j) *is a back arc.*
(3) $visited[j] = 1$ *at statement* (7) *and* $popped[j] = 1$ *at statement* (13) *if and only if* (i, j) *is a cross arc.*

PROOF. (1) The proof follows directly from the definition of tree arcs, since there is a subsequent call visit(j) in visit(i).

(2) The call visit(j) has been made, but has not yet returned, since $visited[j] = 1$ and $popped[j] = 0$. So there must be a path composed of tree arcs from j to i, and thus, (i, j) is a back arc. In the other direction, if (i, j) is a back arc, there must be a path from j to i composed of tree arcs. Thus, visit(j) is invoked before visit(i), and so $visited[j] = 1$. Further, since visit(i) has not returned, there is some child k of j for which the call visit(k) in visit(j) has not returned, and so visit(j) cannot have returned. Thus, $popped[j] = 0$.

(3) From (1) and (2), (i, j) is not a tree or back arc. Since j is not in S_i (by the condition of statement (5)), it cannot be a forward arc. Thus, it must be a cross arc. In the other direction, if it is a cross arc, the call visit(j) must have returned, and so $visited[j] = 1$ and $popped[j] = 1$. □

Several sets related to node i are of interest and are defined below. In what follows, V is a set of nodes such that $V \subseteq E_i$.

D_i^V The set of descendents of i via i's children in V.
T_i^V The set of descendents of i via i's children in V in the spanning forest of calls to visit.

B_i^V The set of descendents of i via i's children in V and via paths containing only tree and cross arcs, except for the last arc, which is a back arc. (Thus, the members of this set are heads of back arcs.)

POP_i^V The set of descendents of i via i's children in V for which the call to visit returned before the call visit(i).

LEMMA A.3. *During the execution of GDFTC the following hold.*

(a) *For every node i, after the call visit(i) returns, one of the following holds:*

(a1) $frame[i] = n + 1$, $root[i] = n + 1$, *and* $S_i = D_i^{E_i}$, *or*

(a2) $frame[i] = top$
$root[i] = r$, *such that* $visited[r] = min_{k \in B_i^{E_i}}\{visited[k]\}$.
$nodes[frame[i]] = \{j | j \in T_i^{E_i}$ *and* $frame[j] \neq n + 1\} \cup \{i\}$
$S_i = list[frame[i]] = \{j | j \in POP_i^{E_i}$ *and* $frame[j] = n + 1\}$

(b) *For every tree arc (i, j), after the call visit(j) returns, the following holds:*

(b1) $frame[i] = n + 1$, *if* $frame[i] = n + 1$ *before the call visit(j).*

(b2) $frame[i] = TOP$, *if* $frame[i] = TOP$ *before the call visit(j).*
Further, $frame[j] = TOP + 1$ *or* $frame[j] = n + 1$.

PROOF. Note that initially $frame[i] = root[i] = n + 1$ and $S_i = \emptyset$, for all nodes i. Let $pop[i]$ be the pop order of i, i.e., the order in which visit(i) returned. We prove the lemma by induction on $pop[i]$. (This induction is referred to in the sequel as the *outer induction.*)

Basis. Let $pop[i] = 1$, i.e., i is the first node for which visit(i) returned. Clearly, either i is a node with zero outdegree, so the loop of statements (5)–(19) is never entered, or for all children j of i, $visited[j] > 0$ and $popped[j] = 0$ when they are examined, i.e., all arcs (i, j) are back arcs. (In both cases $B_i^{E_i} = E_i$ and $POP_i^{E_i} = T_i^{E_i} = \emptyset$. Moreover, in the former case $E_i = \emptyset$.) Part (b) of the lemma does not apply here, so we only prove part (a). Within the call visit(i), let V denote the set of children of i that have been iterated through in statements (5)–(19) of the algorithm at any time. Modify (a) in the statement of the lemma into (a′) so that it reads as follows.

(a′) For every node i, after examining all children of i in $V \subseteq E_i$, one of the following holds:

(a1′) $frame[i] = n + 1$, $root[i] = n + 1$, and $S_i = \emptyset$, or

(a2′) $frame[i] = top$
$root[i] = r$, such that $visited[r] = \min_{k \in V}\{visited[k]\}$.
$nodes[frame[i]] = \{i\}$
$S_i = list[frame[i]] = \emptyset$

We prove the above by induction on the size of V, i.e., the number of children of i that have been examined at any time.

Basis. Let $|V| = 0$. Then, the **for**-loop of statements (5)–(19) has not been executed at all, and therefore *frame*[i] and *root*[i] remain as they were before, i.e., equal to $n + 1$. Moreover, $S_i = \emptyset$. Thus, (a1′) holds.

Induction Step. Assume that the claim is true after examining $c \geq 0$ children. We prove it for $c + 1$, i.e., $|V| = c + 1$. Let j be the $(c + 1)$th child of i, i.e., $V^{new} = V^{old} \cup \{j\}$. Then, (i, j) is a back arc. By the induction hypothesis, before examining j in statement (5), either (a1′) or (a2′) holds. In the former case (j is the first child of i to be examined and $c + 1 = 1$), the condition in statement (18) is satisfied and its **then** part is executed, establishing the following.

frame[i] = *top*
nodes[*frame*[i]] = *nodes*[*top*] = {i}
S_i = *list*[*frame*[i]] = $\emptyset$

In addition, statement (19) is executed, and because *visited*[$n + 1$] = $n + 1$ and $0 < visited[j] \leq n$, the **then** part of statement (30) establishes

(*root*[i] = j, where vacuously (since V is singleton)
$visited[j] = \min_{k \in V} \{visited[k]\}$.

Thus (a2′), and therefore (a′), holds. In the latter case (j is not the first child of i to be examined), the test in statement (18) fails, and only statement (19) is executed, possibly updating *root*[i] as (a2′) requires. The remaining clauses of (a2′) remain valid by the induction hypothesis. Thus, in this case also, (a2′), and therefore (a′) holds.

After examining all the children of i, (a′) holds for $V = E_i$. Since i is the first node for which the call visit(i) returned, by (a′) $i \neq root[i]$, thus statement (21) is skipped, and (a′) still holds after visit(i) returns. We have already mentioned that in this case, $B_i^{E_i} = E_i$ and $POP_i^{E_i} = T_i^{E_i} = \emptyset$. Thus, (a) reduces to (a′) and the basis case of the outer induction is proved.

Induction Step. Assume that the lemma is true for all nodes i such that $pop[i] \leq pop$ for some $pop \geq 1$, i.e., for the first *pop* nodes i for which visit(i) returned. We prove it for the (pop + 1)th. Let h be he *pop*th node for which the call to visit returned and let i be the (pop + 1)th such node. By the depth-first traversal structure of the algorithm, either i is a leaf in the spanning forest of calls to visit (a descendent of a sibling of h or a member of a different tree in the forest) or i is the father of h and visit(h) was called from within visit(i). We examine the two cases separately.

Assume that i is a leaf in the spanning forest. Then all arcs (i, j), if any, are either back arcs or cross arcs; thus, $T_i^{E_i} = \emptyset$. As in the basis case, since no call visit(j) is issued for any child j of i, part (b) of the lemma does not apply, so we only prove part (a). Within the call visit(i), let V denote the set of children of i that have been iterated through in statements (5)–(19) of the algorithm at any time. Modify (a) in the statement of the lemma into (a″) so that it reads as follows:

(a″) For every node i, after the call visit(i) returns, one of the following holds.

(a1″) $frame[i] = n + 1$, $root[i] = n + 1$, and $S_i = D_i^V$

(a2″) $frame[i] = top$
$root[i] = r$, such that $visited[r] = \min_{k \in B_1^V}\{visited[k]\}$.
$nodes[frame[i]] = \{i\}$
$S_i = list[frame[i]] = \{j | j \in POP_i^V$ and $frame[j] = n + 1\}$

As in the basis case, we prove the above by induction on the size of V, i.e., the number of children of i that have been examined at any time.

Basis. Let $|V| = 0$. Then, the **for**-loop of statements (5)–(19) has not been executed at all, and therefore $frame[i]$ and $root[i]$ remain as they were initially, i.e., equal to $n + 1$. Moreover, $S_i = D_i^{\emptyset} = \emptyset$. Thus, (a1″) holds.

Induction Step. Assume that the claim is true after examining $c \geq 0$ children. We prove it for $c + 1$, i.e., $|V| = c + 1$. Let j be the $(c + 1)$th child of i, i.e., $V^{new} = V^{old} \cup \{j\}$. Arc (i, j) can be a back arc or a cross arc. We treat the two cases separately. Assume that (i, j) is a back arc, i.e., the condition of statement (17) is satisfied, by Lemma A.2. By the induction hypothesis, before examining j in statement (5), either (a1″) or (a2″) holds for i. If (a1″) holds (j is the first child of i to be examined and reveal that i is a member of a nontrivial strong component), the condition in statement (18) is satisfied and its **then** part is executed, establishing the following.

$frame[i] = top$
$nodes[frame[i]] = nodes[top] = \{i\}$
$S_i = list[frame[i]] = \{j | j \in POP_i^V$ and $frame[j] = n + 1\}$

The last equality is justified by the fact that for a back arc (i, j), the addition of j into V does not affect the contents of POP_i^V. In addition, statement (19) is executed, and because $visited[n + 1] = n + 1$ and $0 < visited[j] \leq n$, the **then** part of statement (30) establishes

$root[i] = j$, where vacuously (since B_i^V is singleton)
$visited[j] = \min_{k \in B_i^V}\{visited[k]\}$.

Thus, (a2″), and therefore (a″), holds. If (a2″) holds before examining j, the test in statement (18) fails, and only statement (19) is executed, possibly updating $root[i]$ as (a2″) requires. The remaining clauses of (a2″) remain valid by the induction hypothesis. (The value of S_i and $list[frame[i]]$ have to remain the same, since the addition of a back arc in V does not affect the contents of POP_i^V.) Thus, in this case also, (a2″), and therefore (a″) holds.

Assume that (i, j) is a cross arc, i.e., the condition of statement (13) is satisfied, by Lemma A.2. Since $popped[j] = 1$, the call to visit(j) has already returned. Thus, by the induction hypothesis of the outer induction the lemma holds for j. If $root[j] = n + 1$, (a) holds for j and the complete set of descendents of j is stored in S_j and propagated to S_i at statement (14). If (a1″) holds for i before examining j, S_i is correctly updated to D_i^V (with V

containing j also). If (a2″) holds for i before examining j, since all nodes in S_j have been popped before i, they are members of POP_i^V. Thus, S_i and $list[frame[i]]$ are updated correctly also. The values of $frame[i]$, $root[i]$, and $nodes[i]$ correctly remain unchanged. (Specifically for $root[i]$, the addition to V of the head of a cross arc that is in a different strong component ($root[j] = n + 1$), cannot have any effect on the contents of B_i^V.) On the other hand, if $root[j] \neq n + 1$, the test in statement (14) fails, and control reaches statement (15). Recall that, by the induction hypothesis, before examining j in statement (5), either (a1″) or (a2″) holds for i. If (a1″) holds, the condition in statement (15) is satisfied and its **then** part is executed, establishing the following:

$$frame[i] = top$$
$$nodes[frame[i]] = nodes[top] = \{i\}$$
$$S_i = list[frame[i]] = \{j | j \in POP_i^V \text{ and } frame[j] = n + 1\}$$

The last equality is justified by the induction hypothesis of the outer induction, since for a cross arc (i, j), if $root[j] \neq n + 1$ then $frame[j] \neq n + 1$. Thus the addition of j to V does not affect $\{j | j \in POP_i^V \text{ and } frame[j] = n + 1\}$. In addition, the following will be established:

$$root[i] = j, \text{ where vacuously (since } B_i^V \text{ is singleton}$$
$$visited[j] = \min_{k \in B_i^V} \{visited[k]\}.$$

Thus, (a2″), and therefore (a″), holds. If (a2″) holds for i before examining, j, statement (16) is executed, possibly updating $root[i]$ as (a2″) requires. The remaining clauses of (a2′) remain valid by the induction hypothesis, since again the contents of $\{j | j \in POP_i^V \text{ and } frame[j] = n + 1\}$ are not affected by the addition of j to V. Thus, in this case also, (a2″), and therefore (a″) holds.

Note that (a″) reduces to (a) when V becomes equal to E_i. (Recall that for a leaf of the spanning forest, $T_i^{E_i} = \emptyset$.) For a leaf of the spanning forest of calls to visit, it can never be true that $i = root[i]$. This is because $root[i]$ is either made equal to a child of i (statement (19)) (but i is never examined as a child of itself (statement (5))), or it is made equal to the root of a child of i (statements (11), (12), and (16)). By the induction hypothesis of the outer induction, (a) holds for j. Statements (11), (12), and (16) are only executed when $root[j] \neq n + 1$, thus (a2) must hold for j. If $i = root[j]$ at some point, this means that there is a path from i to j of tree and cross arcs only starting with a tree arc from i (and finishing with a back arc to i). This, however, contradicts the hypothesis that i is a leaf in the spanning forest. Thus i cannot be equal to $root[i]$, statement (21) is skipped, and after the return of visit(i), (a) holds. This completes the proof of the lemma for the case that i is not the father of h.

Assume that i is the father of h. For the first time, since the call to visit(h) was taken, we need to prove both (a) and (b) for i. We first prove (b). Node i is never placed in an entry of *nodes*, except within the top level call of visit(i)

(statements (11), (12), and (16)). Thus, if $frame[i] = n + 1$ before visit(j) is called for a child j of i, this will not be changed after the return of the call to visit(j). When i is inserted into some entry of *nodes* it is always true that $frame[i] = top$ (statements (11), (15), and (18)). Thus, consider the case where $frame[i] = top = TOP$, for some value TOP, before the call to visit(j), for some child j of i. During the call visit(j), top may be increased and decreased multiple times. Consider the last time within any recursive call of visit(j) that top was increased from TOP to $TOP + 1$ without being decreased to TOP before the call to visit(j) returns. Assume that this happened within the call visit(l). Clearly, l is a descendent of i and j in the spanning forest of calls to visit. We prove that for any node k in the path from j to l in the forest, when call visit(k) returns, $frame[k] = TOP + 1$. This will be done by induction on the distance of k from l.

Basis. Consider l itself. Consider any call visit(m) to a child m of l, after top is increased to $TOP + 1$ and $frame[l]$ is set to $TOP + 1$ (statement (15) or (18)). Since l and m have been popped before i, by the induction hypothesis of the outer induction, (a) and (b) hold for l and m. Thus, when visit(m) returns, either $root[m] = frame[m] = n + 1$, statement (9) is executed, and $frame[l] = top = TOP + 1$, or $root[m] \neq n + 1$ and $frame[m] = top = TOP + 2$, in which case, since $root[l] \neq n + 1$, statement (12) is executed, and $frame[l]$ (also $frame[m]$) is set to $TOP + 1 = top$. This covers the basis case.

Induction Step. Assume that the claim is true for an arbitrary node k' in the path between j and l. We prove it for its father in this path k. Again, by the induction hypothesis of the outer induction, (a) and (b) hold for both k and k'. When the call *visit*(k') returns, by the induction hypothesis of the inner induction, $frame[k'] = TOP + 1 = top$. Moreover, $root[k] = frame[k] = n + 1$. Otherwise, it should be $frame[k] = TOP$ (outer induction hypothesis (b) for k). If that were the case, statement (12) would be executed, and top would be decreased to TOP, which contradicts our assumption that this would not happen between the call visit(l) and the return of the call visit(j) to visit(i). Thus, $root[k] = frame[k] = n + 1$, statement (11) is executed, which set $frame[k]$ equal to $frame[k'] = TOP + 1$. After any other call visit(k''), following the return of visit(k'), within the call visit(k), by the outer induction hypothesis for k, either $frame[k''] = n + 1$, in which case $frame[k]$ remains equal to $top = TOP + 1$, or $frame[k''] = TOP + 2 = top$, in which case after the execution of statement (12), $frame[k]$ is set back again to $top = TOP + 1$. Thus, in all cases the claim holds for k.

By the above induction, after visit(j) returns within visit(i), $frame[j] = TOP + 1 = top$. This concludes the proof of (b) for i.

The proof of (a) is straightforward. Recall that i is the next node for which visit(i) returns after the return of visit(h). If $root[h] = n + 1$, by the induction hypothesis (a1), S_h contains all the descendents of h, which are correctly propagated to S_i or both S_i and $list[frame[i]]$ (statement (9)). Nothing else is modified: $frame[i]$ should remain $n + 1$ or top; $root[i]$ should retain its

value, because $root[h] = n + 1$; $nodes[frame[i]]$ should also retain its value, since for any member $k \in T_i^{\{h\}}$, which is the set of new members of T_i^V, $frame[k] = n + 1$ by the induction hypothesis and the fact that $root[h] = n + 1$. Thus, in this case, (a) holds. If $root[h] \neq n + 1$, either statement (11) will be executed or statement (12). Addressing each case is similar to previous parts of this proof and is omitted. In all cases, (a) is seen to hold. Node i, may have other children to examine after h, all of which must be heads of cross or back arcs. It is easily seen that the claim still holds after examining these nodes also, as was done before. If (a1) holds when control reaches statement (20); i.e., $root[i] = n + 1$, then statement (21) is skipped, and (a1), and therefore (a) also, holds after visit(i) returns. If (a2) holds when control reaches statement (20), i.e., $root[i] \neq n + 1$, but $root[i] \neq i$, then again (a2) remains valid after visit(i) returns. Finally, if (a2) holds but $i = root[i] \neq n + 1$, then statement (21) is executed, and establish (a1) for i after visit(i) returns. In all cases, (a) holds for i. □

THEOREM A.4. *Algorithm GDFTC terminates and correctly computes the transitive closure of G.*

PROOF. Clearly, the algorithm terminates since every edge in G is considered only once (statement (5)). By Lemma A.3, any node i that satisfied (a1) after the call visit(i) returned, has its descendents correctly computed and stored in S_i. Consider a node i that satisfied (a2) after the call visit(i) returned. (These are the nodes that are members of nontrivial strong components but not a root.) This means that, when visit(i) returned, $root[i]$ was equal to a node r for which visit had not returned yet. Since there is a path in the spanning forest from r to i, and a path ending in a back arc whose head is r, $r \in B_r^{E_r}$, and therefore, by Lemma A.3 for r, $visited[root[r]] \leq visited[r]$ after all calls to r's children return. If $visited[root[r]] < visited[r]$, due to the finiteness of the nodes, there must be a node r', such that $root[r'] = r'$, that is an ancestor of i in the spanning forest of calls to visit. Before visit(r') returns, all members of $nodes[frame[i]]$ have their descendents set equal to $S_{r'}$. Since $i \in T_{r'}^{E_{r'}}$ and $frame[i] \neq n + 1$ (by Lemma A.3), $i \in nodes[r']$, and i's descendents are appropriately updated. The correctness of the update is straightforward, since there is a cycle that involves both r' and i, and therefore, the two nodes have the same descendents. □

REFERENCES

1. AGRAWAL, R., DAR, S., AND JAGADISH, H. V. Direct transitive closure algorithms: Design and performance evaluation. *ACM Trans. Database Syst. 15*, 3 (Sept. 1990), 427–458.
2. AGRAWAL, R., AND JAGADISH, H. V. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International VLDB Conference* (Brighton, England, Sept. 1987), 255–266.
3. AGRAWAL, R., AND JAGADISH, H. V. Hybrid transitive closure algorithms. In *Proceedings of the 16th International VLDB Conference* (Brisbane, Australia, Aug. 1990). VLDB Endowment, 326–334.
4. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

5. BANCILHON, F. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems—Integrating Database and AI Systems*, M. Brodie and J. Mylopoulos, Eds., Springer-Verlag, New York, 1985.
6. CARRE, B. *Graphs and Networks*. Clarendon Press, Oxford, England, 1979.
7. CRUZ, I., AND NORVELL, T. S. Aggregative closure: An extension of transitive closure. In *Proceedings of the 5th IEEE Data Engineering Conference* (Los Angeles, Feb. 1989), 384–391.
8. DAR, S. Ph.D. thesis, Univ. of Wisconsin-Madison, Aug. 1993.
9. DIJKSTRA, E. W. A note on two problems in connection with graphs. *Numer. Math. 1* (1959), 269–271.
10. EBERT, J. A sensitive transitive closure algorithm. *Inf. Process. Lett. 12*, 5 (1981).
11. EVE, J., AND KURKI-SUONIO, R. On computing the transitive closure of a relation. *Acta Inf.* (1977), 303–314.
12. IOANNIDIS, Y. E. On the computation of the transitive closure of relational operators. In *Proceedings of the 12th International VLDB Conference* (Kyoto, Aug. 1986), 403–411.
13. IOANNIDIS, Y. E., AND RAMAKRISHNAN, R. Efficient transitive closure algorithms. In *Proceedings of the 14th International VLDB Conference* (Long Beach, Calif., Aug. 1988), 382–394.
14. JIANG, B. A suitable algorithm for computing partial transitive closures in databases. In *Proceedings of the 6th IEEE Data Engineering Conference* (Los Angeles, Feb. 1990), 264–271.
15. JIANG, B. I/O-efficiency of shortest path algorithms: An analysis. In *Proceedings of the 8th IEEE Data Engineering Conference* (Tempe, Ariz., Feb. 1989). IEEE, New York, 12–19.
16. KABLER, R., IOANNIDIS, Y. E., AND CAREY, M. Performance evaluation of algorithms for transitive closure. *Inf. Syst. 17*, 5 (Sept. 1992), 415–441.
17. KUNG, R., HANSON, E., IOANNIDIS, Y. E., SHAPIRO, L., SELLIS, T., AND STONEBRAKER, M. Heuristic search in database systems. In *Expert Database Systems, Proceedings of the 1st International Workshop*, L. Kerschberg, Ed., Benjamin-Cummings, Menlo Park, Calif., 1986, 537–548.
18. LU, H. New strategies for computing the transitive closure of a database relation. In *Proceedings of the 13th International VLDB Conference* (Brighton, England, Sept. 1987), 267–274.
19. LU, H., MIKKILINENI, K., AND RICHARDSON, J. P. Design and evaluation of algorithms to compute the transitive closure of a database relation. In *Proceedings of the 3rd International Data Engineering Conference* (Los Angeles, Feb. 1987), 112–119.
20. PURDOM, P. A transitive closure algorithm. *BIT, 10* (1970), 76–94.
21. NAUGHTON, J. F. One-sided recursions. In *Proceedings of the 6th ACM-PODS Conference* (San Diego, Calif., Mar. 1987), 340–348.
22. QADAH, G. Z., HENSCHEN, L. J., AND KIM, J. J. Efficient algorithms for the instantiated transitive closure queries. *IEEE Trans. Softw. Eng. 17*, 3 (Mar. 1991), 296–309.
23. RICH, E. *Artificial Intelligence*. McGraw-Hill, New York, 1983.
24. ROSENTHAL, A., HEILER, S., DAYAL, U., AND MANOLA, F. Traversal recursion: A practical approach to supporting recursive applications. In *Proceedings of the 1986 ACM-SIGMOD Conference* (Washington, D. C., May 1986), 166–176.
25. SCHMITZ, L. An improved transitive closure algorithm. *Computing 30* (1983), 359–371.
26. TARJAN, R. E. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (1972), 146–160.
27. VALDURIEZ, P., AND BORAL, H. Evaluation of recursive queries using join indices. In *Proceedings of the 1st International Expert Database Systems Conference* (Charleston, S. C., April 1986), 197–208.
28. WARREN, H. S. A modification of Warshall's algorithm for the transitive closure of binary relations. *Commun. ACM 18*, 4 (April 1975), 218–220.
29. WARSHALL, S. A theorem on Boolean matrices. *JACM, 9*, 1 (Jan. 1962), 11–12.

Received August 1989; revised September 1991; accepted July 1992