Mixed-Approach Algorithms for Transitive Closure

Extended Abstract

Håkan Jakobsson*
Computer Science Department
Stanford University
Stanford, CA 94305, USA.
hakan@cs.stanford.edu

Abstract

We study two different approaches for computing the transitive closure of a directed graph and show that, in some sense, they are "dual" on edge-reversed graphs but, nevertheless, can differ asymptotically in cost on the same family of graphs. We show how the two approaches can be mixed into a new algorithm using reachability trees. We show that the new algorithm is $O(\Sigma_{(x,y)\in V\times V} CONN(x,y))$ where $CONN(x,y)$ is the pairwise connectivity of x and y, and give a more exact connectivity-based upper bound that is better than the lower bound for a wide class of other algorithms on every family of graphs.

1 Introduction

The transitive closure of a directed graph is a binary relation, TC, such that $(i,j) \in TC$ iff there is a path of edges from node i to node j. In cyclic graphs, all the nodes in a strongly connected component have the same set of reachable nodes and this observation can be used reduce the cyclic case to the acyclic. See, for instance, [Pur70], [Dzi75], [Ebe81], [Sch83], and [IR88].

The asymptotically fastest known algorithms are based on the fact that transitive closure reduces to matrix multiplication (see [Mun71] or [AHU74]). Hence, algorithms such as the one by Coppersmith and Winograd [CW87], which is $O(n^{2.376})$, can be used. Unfortunately, the asymptotically fastest algorithms suffer from high constant factors; more practical matrix-based algorithms, such as Warshall's [War62] and Warren's [War75], exist but do not take advantage of structural properties that make some graphs easy. However, many algorithms exist where the structure of the graph affects execution time. As will be described in Section 2, two basic approaches seem to underlie several of the published algorithms of this latter type. We will show how it is possible to combine these approaches into a single algorithm that is better at taking advantage of the structure of the graph than algorithms based on a single approach.

We will use the following notation and cost model: We seek to compute the transitive closure relation TC for a directed graph given its edge relation E and set of nodes V. We will use $TC(i)$ as shorthand for $\pi_2(\sigma_{\$1=i}(TC))$ and $REACH(i)$ for $\{i\} \cup TC(i)$, i.e., the set of nodes that are reachable from i. Also, we will use $EREACH(i)$ for $\{(j,k) \in E \mid j,k \in REACH(i)\}$, i.e., the set of edges that are reachable from i and $SUCC(i)$ to denote $\pi_2(\sigma_{\$1=i}(E))$, the immediate successors of i. Let n be the number of vertices in the graph, e the number of edges, and e_{red} the number of edges in the transitive reduction. In our cost model we assume that we can look up, insert, delete, or modify any tuple in a relation in constant time. We also assume that we can perform a selection on any attribute in time that is proportional to the number of tuples retrieved.

*Work supported by NSF grant IRI-87-22886, Air Force grant AFOSR-90-0066, and a grant of IBM Corporation.

2 Two Approaches

We now describe two simple approaches to computing tuples in TC that, in various forms, are used in several published algorithms.

2.1 Dynamic Programming

In the first approach, we note that, for each node i,

$$TC(i) = \bigcup_{j \in SUCC(i)} REACH(j).$$

This observation may suggest a dynamic-programming algorithm where for any node i, the sets of reachable nodes are first constructed for each of its immediate successors and then used to infer what nodes are reachable from i. If we require that the reachability sets for the successors be completely computed before they are used, we can only deal with acyclic graphs. (The algorithm called DAG_DFTC in [IR88] is an example of this approach.) We can handle cyclic graphs as well if we compute the reachability sets incrementally. If we evaluate the right-linear logic programming rules for transitive closure

$tc(X,Y)$:- $e(X,Y)$.
$tc(X,Y)$:- $e(X,Z)$, $tc(Z,Y)$.

with the semi-naive evaluation method ([Ban86]), we get an algorithm like Algorithm 1, which infers the reachability sets for each node from those of its successors and can handle cyclic graphs.

$TC := E$;
$\Delta TC := E$;
repeat
  $New\_\Delta TC := \pi_{x,y}(E(x,z) \bowtie \Delta TC(z,y)) - TC$;
  $TC := TC \cup New\_\Delta TC$;
  $\Delta TC := New\_\Delta TC$;
until no change in TC;

Algorithm 1

Examples of published algorithms that use variations of the dynamic programming approach can be found in [GK79], [Sch83], and [Meh84].

2.2 Search

A second approach for computing the transitive closure is to search the graph n times, each time starting from a different node, thereby determining what can be reached from that node. In this approach, we completely disregard any parts of the TC-relation that have already been computed for other starting nodes. Instead, we search the entire part of the graph that is reachable from the starting node by following every edge we can get to. We assume that the cost of the search is proportional to the number of edges that are reachable from the starting node, i.e., to $|EREACH(i)|$ for starting node i. While many different search strategies (depth-first, breadth-first, etc.) are possible, we will use a logic program to describe a search-type algorithm. If we evaluate the left-linear logic program for transitive closure

$tc(X,Y)$:- $e(X,Y)$.
$tc(X,Y)$:- $tc(X,Z)$, $e(Z,Y)$.

with the semi-naive method, we get an algorithm like Algorithm 2 (referred to as "Semi-naive" in [IR88] and several other papers). The algorithm can be viewed as a simultaneous breadth-first search from all nodes in the graph and its cost is $O(\Sigma_{i \in V} |EREACH(i)|)$.

$TC := E$;
$\Delta TC := E$;
repeat
  $New\_\Delta TC := \pi_{x,y}(\Delta TC(x,z) \bowtie E(z,y)) - TC$;
  $TC := TC \cup New\_\Delta TC$;
  $\Delta TC := New\_\Delta TC$;
until no change in TC;

Algorithm 2

Other variations of the search approach is used in several papers: In both [BFM76] and [Sch78], a search approach is used together with some additional features to halt searches early. In both cases, these features rely on nodes having large reachability sets — with cardinalities n and $> n/2$, respectively — in order to be useful.

2.3 Comparison

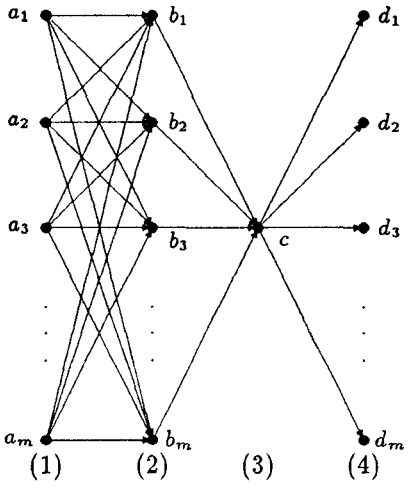
The two approaches we have presented may in some sense seem very different in that the second appears to lack the dynamic-programming flavor of the first and instead works on each node independently without exploiting reachability sets computed for other nodes. However, there is a sense in which the two approaches are "dual" that is suggested by the following theorem:

Theorem 2.1 For any graph, G, with n nodes, let G^R be the graph formed by reversing all the edges of G. If the cost of evaluating Algorithm 1 on G is $f(n)$, then the cost of Algorithm 2 is $\Theta(f(n))$ on G^R. □

As for the worst-case cost, it is easy to see that a search from a source node can involve at most e edges. Hence, a search-based algorithm like Algorithm 2 is $O(ne) = O(n^3)$, and, by Theorem 2.1, so is Algorithm 1. We will later give a connectivity-based lower bound for both algorithms.

We now give an example to show that, in spite of their duality on edge-reversed graphs, the algorithms can differ by a factor of n in cost on the same graph.

EXAMPLE 2.1 Consider the following graph:



There are four "columns" of nodes. Each of the m nodes in (1) has an edge to each of the m nodes in (2). Each of the m nodes in (2) has an edge to the single node in column (3), and this node has edges to each of the m nodes in (4). Let the total number of nodes (i.e., $3m+1$) be n. Then m is $\Theta(n)$ and the total number of edges $\Theta(n^2)$. Note that the graph is transitively reduced.

When we execute Algorithm 1 on the graph, we derive the TC-tuples of the form (a_i, d_j) using E-tuples of the form (a_i, b_k) together with TC-tuples of the form (b_k, d_j). For each node a_i, there are m immediate successors b_k, each of which with m nodes d_j in $TC(b_k)$, resulting in a cost that is $\Omega(n^2)$. Since there are m such a-nodes, Algorithm 1 is $\Omega(n^3)$.

On the other hand, in Algorithm 2, we compute TC-tuples of the form (a_i, d_j) by searching the graph starting at each different a_i. There are $O(m)$ edges that are reachable from each starting node making the total cost over all starting nodes $O(n^2)$.

For the edge-reversed graph, Algorithm 2 is $\Omega(n^3)$ and Algorithm 1 $O(n^2)$ as indicated by Theorem 2.1. □

3 A Mixed-Approach Algorithm

As was shown in Example 2.1, there are families of graphs where Algorithms 1 and 2 have asymptotically different costs. This fact suggests the simple strategy of running both algorithms in parallel until one of them halts, thereby staying within a factor of two in cost of the algorithm that happens to be fastest on a particular graph. Unfortunately, one graph could contain sections where one algorithm is far better than the other but other sections where the other algorithm is best, in which case neither algorithm will do very well on the graph as a whole. Consider, for instance, a graph consisting of the graph in Example 2.1 and its reverse as disconnected subgraphs. Both Algorithms 1 and 2 would be $\Omega(n^3)$ on the graph as a whole. As a matter of fact, all the algorithms referenced in Section 2 would be $\Omega(n^3)$ on such a graph. We now present an algorithm that combines elements of both the search approach and dynamic programming approach and that will never do worse than the best of either approach on any part of a graph.

3.1 Reachability Trees

In the dynamic-programming approach we have described, the $TC(i)$ set for a node i is computed from the union of the reachability sets of its immediate successors. However, a reachability set lacks some useful information of the structure of the graph. In our mixed approach, we will use *reachability trees* which have a structure that can be used to avoid some duplicate derivation of tuples. A reachability tree for a node i is simply a tree rooted in i that contains all nodes that are reachable from i, and where all tree edges are edges in the graph. The basic idea of our mixed approach is to find the nodes that are reachable from a node by forming reachability trees for its immediate successors and search those trees.

Forming the union of reachability sets has a cost that, for each set involved, is proportional to the number of nodes in the set. But the time it takes to traverse a tree is also proportional to the number of nodes so using trees instead of sets can never cost more (except, possibly, by some constant factor).

Moreover, our traversal of reachability trees in a sense amounts to a search, and in our mixed approach, we will maintain the nice property of the search approach never having to traverse the same edge in the graph twice from the same source node. Hence, the cost of our approach will never be more than proportional to the number of edges reachable from each source node and we will never do worse than the regular search approach (except, possibly, by some constant factor).

We now consider the following scenario to illustrate why using trees is superior to merely using sets: Assume that we are computing what nodes are reachable from a node i by searching the reachability tree for some successor j of i. If we get to a node k that we have already found to be reachable from i through some other successor j' of i, we need not search through the descendants of k in the reachability tree for j since we know that all those descendants must also be in the tree for j'. It is thus possible to avoid considering some nodes in the reachability trees and thereby do better than with

mere reachability sets.

It should be noted that some care must be taken when selecting what graph edges should be used as tree edges and when conducting the search of the trees. Otherwise, we may end up with an incorrect algorithm that misses some nodes. Still, there are many ways in which these issues can be handled giving rise to different algorithms. For example, on acyclic graphs, one could recursively compute the reachability tree for a node as a depth-first search tree of the graph formed by those edges that are in the reachability trees of the immediate successors of the node. For cyclic graphs, we would not, in general, be able to completely compute the reachability trees of all successors of a node before computing the tree for the node itself. We would need an algorithm that is incremental, much like Algorithms 1 and 2, and we will present one such algorithm in the next section.

3.2 A New Algorithm

Our new algorithm iteratively forms longer and longer paths. A path of length m is discovered by joining two paths of length $m-1$ that were discovered on the previous iteration. To represent reachability trees, we use a predicate $S(i,j,k,l)$ which should be interpreted as follows: In the reachability tree for node i, there is a path to node l, and on that path, j is the immediate successor of i, and k the immediate predecessor of l. On each iteration of the algorithm, we form tuples (i,j,k,l) from the join $S(i,j,\_,k) \bowtie S(j,\_,k,l)$ meaning that we find new nodes l reachable from i by going down one level in the reachability trees for successors j of i. For every (i,j) pair, we only add one (i,j,k,l) tuple to S — if we allowed more than one path from i to l in S, it would not represent a tree. We have the following algorithm:

(1) $TC(i,j)$:- $E(i,j)$;
(2) $S(i,j,i,j)$:- $E(i,j)$, $i \neq j$;
(3) **repeat**
(4) $New\_S := \emptyset$;
(5) **for each** i s.t. $S(i,\_,\_,\_)$ exists **do**
(6) **for each** j s.t. $S(i,j,\_,\_)$ exists **do**
(7) **for each** (i,j,k,l) in $S(i,j,\_,k) \bowtie S(j,\_,k,l)$ **do**
(8) **if** $(i,l) \notin TC$ **then**
(9) **begin**
(10) add (i,l) to TC;
(11) **if** $i \neq l$ **then**
(12) add (i,j,k,l) to New_S;
(13) **end**;
(14) $S := New\_S$;
(15) **until** no change in TC;

Algorithm 3

Theorem 3.1 Algorithm 3 correctly computes the transitive closure of a directed graph. □

3.3 Analysis and Examples

A transitive closure algorithm seeks to compute the existence of paths in the graph, and it seems reasonable to assume that for many algorithms, the number of paths — especially disjoint paths — between each pair of nodes in the graph may affect the cost of the computation. It therefore seems natural to use various measures of connectivity as tools for analyzing and comparing algorithms of this type. We will show that, unlike other algorithms, the new algorithm has a connectivity-based upper bound and that this upper bound is better than the lower bound for a wide class of algorithms on every family of graphs. We begin by defining some different connectivity measures.

Definition 3.1 A pair of nodes (i,j) has *pairwise connectivity* k, denoted $CONN(i,j) = k$, if k is the largest number such that there exists a set of k paths from i to j that are vertex disjoint (except for the end nodes i and j). □

Definition 3.2 A path P is *significant* if it is a single edge or every proper subpath of P is a shortest path in the graph. □

Definition 3.3 A pair of nodes (i,j) has *significant connectivity* k, denoted $SCONN(i,j) = k$, if k is the largest number such that there exists a set of k significant paths from i to j that are disjoint (except for the end nodes i and j). □

We now give special path and connectivity definitions for our new algorithm. In doing so, we will assume that there exists a set of n ordering relations for the nodes that we will denote $\prec$. For every node i, $\prec_i$ is an ordering on the immediate successors of i that the algorithm uses. Such an ordering does not need to be anything more than the order in which the nodes happen to appear on the adjacency list for i and we do not require that two nodes with the same successor sets have them ordered in the same way.

Note that the following two definitions are mutually recursive.

Definition 3.4 A path $i \rightarrow a_1 \rightarrow \cdots \rightarrow a_m \rightarrow j$ is a *good path* if it is of the form $i \rightarrow j$ (a single edge) or $i \rightarrow a_1 \rightarrow \cdots \rightarrow a_m$ and $a_1 \rightarrow \cdots \rightarrow a_m \rightarrow j$ are both best paths. □

Definition 3.5 Given a set of orderings $\prec$ on the nodes in the graph such that each ordering relation $\prec_i$ is a total ordering on $SUCC(i)$, a path $i \rightarrow a_1 \rightarrow \cdots \rightarrow a_m \rightarrow j$, $i \neq l$, is a *best path* if it is a shortest good path from i to j and there is no good path $i \rightarrow a'_1 \rightarrow \cdots \rightarrow a'_m \rightarrow j$ of the same length such that $a'_1 \prec_i a_1$. □

Definition 3.6 Given a set of orderings $\prec$ on the nodes in the graph such that each ordering relation $\prec_i$ is a total ordering on $SUCC(i)$, a pair of nodes (i, j) has *good-path connectivity* k, denoted $GCONN_{\prec}(i, j) = k$, if there are k good paths from i to j. □

The value of $GCONN_{\prec}(i, j)$ may be different with different orderings $\prec$. Even so, the following theorem holds regardless of what orderings are used.

Theorem 3.2 For any pair of nodes (i, j), $GCONN_{\prec}(i, j) \leq SCONN(i, j) \leq CONN(i, j)$. □

Theorem 3.3 Assume that $\prec$ is the set of orderings used by Algorithm 3 when enumerating the j's in the **for** loop in line (6). Then Algorithm 3 is $O(\Sigma_{(i,j)\in TC} GCONN_{\prec}(i, j))$. □

Corollary 3.1 On any acyclic graph, Algorithm 3 is $O(n\, e_{red})$, where e_{red} is the number of edges in the transitive reduction. □

It is interesting to note that while the $O(n\, e_{red})$ bound has been achieved by other algorithms [GK79], our new algorithm does not require a topological sorting of the nodes in order to achieve the bound.

We will now compare the new algorithm to a whole class of algorithms that are based on composing binary relations. First, we define a basic operation for composition and its cost.

Definition 3.7 Let $JP(A, B)$ be the *join-project* operation $\pi_{1,3}(A(X, Z) \bowtie B(Z, Y))$ for computing the composition of two binary relations, A and B for which the cost is proportional to $|A| + |A(X, Z) \bowtie B(Z, Y)|$. □

The cost assumption means that we assume that a "standard" algorithm is used for the composition (e.g. a nested loop-algorithm) rather than fast (better than $O(n^3)$) boolean matrix multiplication.

Definition 3.8 Let CJP be the class of algorithms that computes the transitive closure through a sequence of relations $P^1, P^2, \ldots, P^m$, where m is the length of the longest shortest path between any pair of nodes in the graph, $TC = \bigcup_{1\leq i\leq m} P^i$, and P^i is defined as

$$P^i = \begin{cases} E, & \text{if } i = 1; \\ JP(P^k, P^{i-k}) - \bigcup_{j<i} P^j, & \text{otherwise.} \end{cases}$$

□

Intuitively, each P^i contains the pairs of nodes between which the shortest path is of length i. By varying the value of k used in $JP(P^k, P^{i-k})$, we get different algorithms in the class CJP. By using a fixed $k = 1$ for every i, we get Algorithm 1 whereas using $k = i - 1$, corresponds to Algorithm 2.

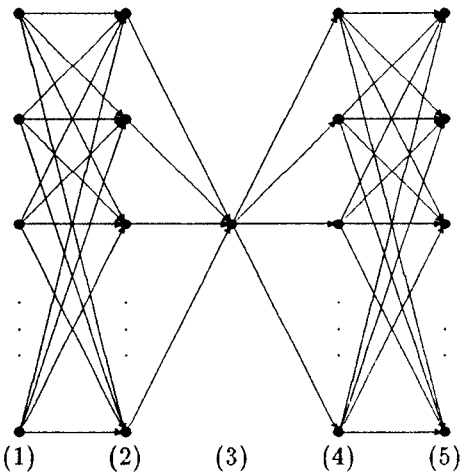
Theorem 3.4 Any algorithm in CJP is $\Omega(\Sigma_{(i,j)\in TC} SCONN(i, j))$ on every family of graphs. □

The theorem implies that if there are k disjoint significant paths from i to j, any algorithm in CJP will infer that $(i, j) \in TC$ *at least* k times. Note that this is just a lower bound and, as will be shown by Example 3.1, there are graphs on which any algorithm in CJP will do much worse.

Theorem 3.5 Assume that the cost formula for JP in Definition 3.7 holds for the join-project operations in Algorithms 1 and 2. Then, both algorithms are $\Omega(\Sigma_{(i,j)\in TC} CONN(i, j))$ on every family of graphs. □

Note that this lower bound does not depend on the breadth-first aspect of Algorithm 2. Were we to compute the transitive closure using a depth-first search from each node, the result would still hold.

EXAMPLE 3.1 Consider the following graph:

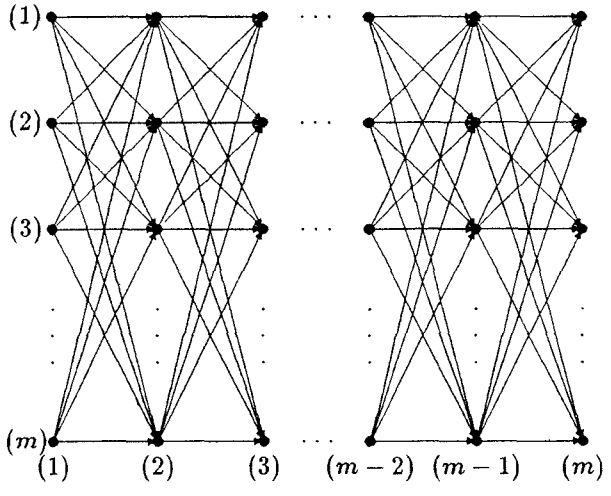


There are 5 columns, each of which has m nodes except column (3) which has a single node. Hence, $m = \Theta(n)$.

For every node in column (k), $1 \leq k \leq 4$, there is an edge to every node in column $(k+1)$. There are only two ways an algorithm in *CJP* can compute P^3: $JP(P^1, P^2) - \bigcup_{j<3} P^j$ and $JP(P^2, P^1) - \bigcup_{j<3} P^j$. Both expressions are $\Omega(n^3)$ on the graph, eventhough, for any pair of nodes (i,j) of distance 3, $SCONN(i,j) = CONN(i,j) = 1$ and $\Sigma_{(i,j)\in TC} SCONN(i,j) = O(n^2)$. Obviously, then, Algorithm 3 is $O(n^2)$ on the graph. □

The previous example showed that there are families of graphs on which any algorithm in *CJP* will do much worse than the connectivity-based lower bound we have given. Our next example will demonstrate that there are graphs for which the upper bound for Algorithm 3 is much better than the lower bound for any algorithm in *CJP*.

EXAMPLE 3.2 Consider the following graph:



The graph can be viewed as a square with m rows and m columns of nodes. Hence, $n = m^2$. For every node in column (k), $1 \leq k \leq m-1$, there is an edge to every node in column $(k+1)$. Let (i,j) be a pair of nodes such that j is reachable from i. If the distance from i to j is 1 or ≥ 3, there is only a single good path from i to j, i.e., $GCONN_{\prec}(i,j) = 1$. In contrast, unless the distance form i to j is 1, $SCONN(i,j) = \sqrt{n}$. We get that $\Sigma_{(i,j)\in TC} SCONN(i,j) = \Omega(n^2\sqrt{n})$ wheras $\Sigma_{(i,j)\in TC} GCONN_{\prec}(i,j) = O(n^2)$. Hence, on this graph, our new algorithm beats any algorithm in *CJP* by a factor of $\sqrt{n}$. □

4 Further Work

The reachability tree approach can be generalized and used for other applications than transitive closure. Our current work includes:

- A generalization of the algorithm for the k-source nodes reachability problem and more general types of recursive queries.
- A join algorithm for certain types of relational algebra queries.
- An adaptation of the new algorithm to solve the all-pairs shortest path problem and certain other generalized transitive closure problems.
- The use of reachability trees to speed up the pivot operation in adjacency list implementations of Warshall's and Floyd's algorithms.
- Updates to recursively defined relations. It is interesting to note that the algorithms by Italiano [Ita86, Ita88] made limited use of reachability trees.

5 Acknowledgements

The author wishes to thank Jeff Ullman and Moshe Vardi for valuable comments on earlier versions of this paper, and Inderpal Mumick for interesting discussions.

References

[AHU74] Aho, A. V., Hopcroft, J. E., Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[Ban86] Bancilhon, F., "Naive evaluation of recursively defined relations," in *On Knowledge Base Management Systems*, (M. L. Brodie and J. Mylopoulos, eds.), Springer-Verlag, New York, 1986.

[BFM76] Bloniarz, P. A., Fischer, M. J., Meyer, A. R., "A note on the average time to compute transitive closures," *Automata, Languages and Programming*, Michaelson and Milner, eds., Edinburgh University Press, Edinburgh, 1976.

[CW87] Coppersmith, D., Winograd, S., "Matrix multiplication via arithmetic progression," *Proc. 19th ACM Symp. on Theory of Computing*, pp 1–6, 1987.

[Dzi75] Dzikiewicz, J. "An algorithm for finding the transitive closure of a digraph," *Computing* 15, pp. 75–79, 1975.

[Ebe81] Ebert, J. "A sensitive transitive closure algorithm," *Inf. Proc. Letters* 12, pp. 255–258, 1981.

[GK79] Goralcikova, A., Koubek, V., "A reduct and closure algorithm for graphs," *Mathematical Foundations of Computer Science*, Springer Lecture Notes in Computer Science 74, pp. 301–307, 1979.

[IR88] Ioannidis, Y. E., Ramakrishnan, R., "Efficient transitive closure algorithms," *Proc. of the 14th International VLDB Conference*, 1988.

[Ita86] Italiano, G. F., "Amortized efficiency of a path retrieval data structure," *Theoretical Computer Science* 48(2,3), pp. 273–281, 1986.

[Ita88] Italiano, G. F., "Finding paths and deleting edges in directed acyclic graphs," *Info. Proc. Letters* 28(1), pp. 5–11, 1988.

[Meh84] Mehlhorn, K., *Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin Heidelberg, 1984.

[Mun71] Munro, I., "Efficient determination of the transitive closure of a directed graph," *Inf. Proc. Letters* 1(2), pp. 56–58, 1971.

[Pur70] Purdom, P., "A transitive closure algorithm," *BIT* 10 pp. 76–94, 1970.

[Sch78] Schnorr, C. P., "An algorithm for transitive closure with linear expected time," *SIAM J. Comput.* 7, pp. 127–133, 1978.

[Sch83] Schmitz, L., "An improved transitive closure algorithm," *Computing* 30, pp. 359–371, 1983.

[War62] Warshall, S., "A theorem on Boolean matrices," *J. ACM* 9(1), pp. 11–12, 1962.

[War75] Warren, H. S., "A modification of Warshall's algorithm for the transitive closure of binary relations," *C. ACM* 18(4), pp. 218–220, 1975.