Efficient Transitive Closure Computation

Esko Nuutila
Eljas Soisalon-Soininen

Laboratory of Information Processing Science
Helsinki University of Technology
Otakaari 1, FIN-02150 Espoo, Finland

enu@cs.hut.fi, ess@cs.hut.fi

TKO-B113/93
ISBN 951-22-1906-9
ISNN 0785-6601

TKK OFFSET 1993

Abstract

We present two new transitive closure algorithms that are based on strong component detection. The algorithms scan the input graph only once without generating partial successor sets for each node. The new algorithms eliminate the redundancy caused by strong components more efficiently than previous transitive closure algorithms. We present statistically sound simulation experiments showing that the new algorithms are more efficient than previous ones. We also describe a compact representation for the successor sets of the transitive closure that can be used together with the new algorithms. The new representation saves much space and reduces the number of operations needed in successor set construction. We show that the representation often eliminates redundant computations caused by multiple paths between pairs of nodes.

1 Introduction

Finding the transitive closure of a directed graph is an important subproblem in many computational tasks. It is required, for instance, in the reachability analysis of transition networks representing distributed and parallel systems and in the construction of parsing automata in compiler construction. Recently, efficient transitive closure computation has been recognized as a significant subproblem in evaluating recursive database queries, since almost all practical recursive queries are transitive.

Despite the increased efficiency of computers the need for faster transitive closure algorithms remains. This is for two reasons. First, the size of the inputs seems to grow in proportion to the growth of the memory capacity. Since the CPU speed has grown at the same rate as the memory capacity, only linear algoritms retain their execution times on typical inputs. Traditional transitive closure algorithms, such as [4, 5, 12, 14, 15, 17, 18], are not linear. Second, typical inputs (and outputs) in the area of databases do not fit into the main memory. Traditional transitive closure algorithms are mostly designed for main memory operation.

In computing the transitive closure, a major problem is the avoidance of redundant computations. A directed graph may contain pairs of nodes that are connected via multiple paths. Several nodes may have the same set of successor nodes. The graph in Figure 1 illustrates these redundancies. Two paths lead from node a to node f and all its successors. Nodes d, e, f, g, h, and i have the same successor set.

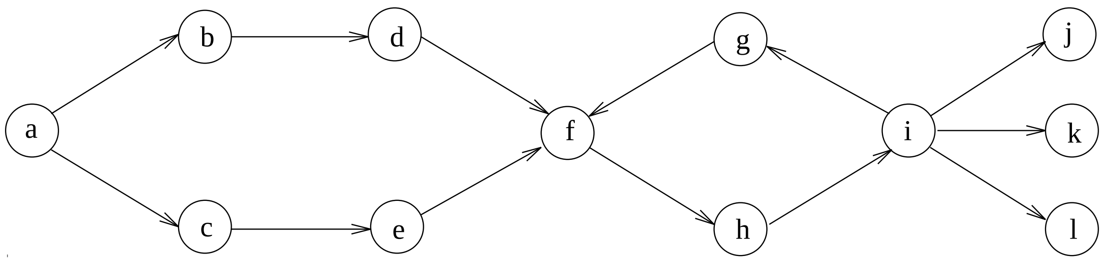

Figure 1: A directed graph.

One of the most important sources of redundant computations are the strong components of the input graph [6], since all nodes in a strong component have the same successor set. For example, in Figure 1 nodes f, g, h, and i are in the same strong component. Some transitive closure algorithms presented in the literature are based on the detection of the strong components [4, 5, 7, 8, 12, 14, 15], but these algorithms do not efficiently avoid all redundancies caused by strong components. Some algorithms generate a partial successor set for each node of a component. Other scan the input graph more than once. In computing the successor sets, the algorithms perform unnecessary set operations.

In this paper, we present two new transitive closure algorithms that are based on strong component detection. Compared to the previous algorithms, the main difference is that the new algorithms scan the input only once without generating a partial successor set for each node.

We present results of simulation experiments showing that the new algorithms are superior to the previous algorithms that are based on strong component detection.

Another important issue in efficient transitive closure computation is the choice of the representation for the transitive closure. The representation should be compact to reduce disk I/O. Usually, common set representations, such as bit matrices, bit vectors, and ordered or inordered lists, have been used for representing the transitive closure. The problem is that these representations do not take advantage of the properties of directed graphs. For instance, a bit vector representation may take quadratic space even when the size of a closure is linear to the number of nodes.

We present a new compact successor set representation that can be used in algorithms that compute the transitive closure during a single depth-first traversal. The representation uses numeric intervals of consecutively numbered strong components to compactly store the successor sets. The representation is based on the transitive closure compression technique for acyclic digraphs by Agrawal et al. [1].

We present results of simulation experiments showing that in addition to saving much space, the interval representation speeds up transitive closure computation. We show that the interval representation largely eliminates the inefficiencies caused by multiple paths between two nodes. Jakobson [9] presents a method that is specially designed for avoiding the redundancy caused by multiple paths between nodes. In examples that Jakobson presented to demonstrate the efficiency of his method the interval representation handles multiple paths even more economically.

We use the following terminology. We consider a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Without loss of generality, we assume that G has no self loops (v, v). The *transitive closure* of G is a graph $G^+ = (V, E^+)$ such that for all $v, w \in V$ there is an edge $(v, w) \in E^+$ if and only if there is a path from v to w in G that has at least one edge. The *successor set* of a node v in G is the set $Succ[v] = \{w \mid (v, w) \in E^+\}$. Two nodes v and w in G are *path equivalent* if there is a path from v to w and a path from w to v. Path equivalence partitions V into maximal disjoint sets of path equivalent nodes. These sets are called the *strong components* of the graph. A strong component that has only one node is called a *trivial component*.

2 Transitive closure computation using strong components

All nodes in a strong component have the same successor set. Several transitive closure algorithms presented in the literature are based on detecting the strong components of the input graph, see [4, 5, 7, 8, 12, 14, 15]. Most of these algorithms are based on Tarjan's strong component algorithm [16]. The problem with the previous algorithms is that they either generate one partial successor set for each node of the component or scan the input graph more than once. In this section we present two new transitive closure algorithms that solve these problems.

2.1 Simple algorithm

As a starting point we use the simple transitive closure algorithm SIMPLE_TC presented in Figure 2. The algorithm is Tarjan's algorithm, with lines 5, 9, and 17 added for the transitive closure computation.

```
(1)  procedure SIMPLE_TC(v);
(2)  begin
(3)      root[v] := v; C[v] := Nil;
(4)      PUSH(v, stack);
(5)      Succ[v] :={w | (v, w) ∈ E};
(6)      for each node w such that (v, w) ∈ E do begin
(7)          if w is not already visited then SIMPLE_TC(w);
(8)          if C[w] = Nil then root[v] := MIN(root[v], root[w])
(9)          Succ[v] :=Succ[v] ∪ Succ[w];
(10)     end;
(11)     if root[v] = v then begin
(12)         create a new component C;
(13)         repeat
(14)             w := POP(stack);
(15)             C[w] := C;
(16)             insert w into component C;
(17)             Succ[w] :=Succ[v];    /* Pointer assignment, not a copy */
(18)         until w = v
(19)     end
(20) end;
(21) begin/* Main program */
(22)     stack := ∅;
(23)     for each node v ∈ V do
(24)         if v is not already visited then SIMPLE_TC(v)
(25) end.
```

Figure 2: Tarjan's algorithm adapted to transitive closure computation.

We first review the strong component detection. Procedure SIMPLE_TC enters the nodes of the graph in depth-first order. For each strong component C, the first node of C that is entered is the *root* of component C. The main goal is to find the component roots. For this purpose, we define a variable $root[v]$ for each node v. When procedure SIMPLE_TC is processing node v, $root[v]$ contains a candidate node for the root of the component containing v. Initially (at line 3), node v itself is the root candidate. When procedure SIMPLE_TC processes the edges leaving node v (at lines 7 and 8), new root candidates are obtained from the children nodes that belong to the same component as v. The *MIN* operation (at line 8) compares the nodes with respect to the order in which procedure SIMPLE_TC has entered them, i.e., $MIN(x, y) = x$ if procedure SIMPLE_TC entered node x before it entered node y, otherwise $MIN(x, y) = y$. When procedure SIMPLE_TC has processed all edges leaving v, $root[v] = v$ if and only if v is a component root (line 11). To distinguish between nodes belonging to the same component as node v and nodes belonging to other components, a variable $C[w]$ is defined for each node w. Its initial value is *Nil*. When a component C is fully detected procedure SIMPLE_TC sets $C[w] = C$ for each node w that belongs to C (line 15). A stack is used for this purpose. Each node is stored on the stack in the beginning of procedure SIMPLE_TC. When the component is fully detected the nodes belonging to it are on top of the stack. Procedure SIMPLE_TC removes them from the stack, sets their $C[w]$ variables, and inserts them into component C.

To compute the transitive closure, a variable $Succ[v]$ is defined for each node v. It contains the (partially computed) successor set of node v. Initially (at line 5) $Succ[v]$ contains only the children nodes of v. At line 9, the successors of a child node w computed so far are inserted into $Succ[v]$. When a strong component is fully detected the successor set of the root node is correctly computed. Other nodes of the component may have incomplete successor sets. At line 17 the successor set of the component root is distributed to the other members of the component.

The benefit of this simple algorithm is that the input graph is processed only once. Unfortunately, the algorithm has some inefficiences. Consider the graph in Figure 3. When the algorithm processes this graph starting at node one it first generates an incomplete successor set to each node. The total memory requirement for these sets is $\Theta(n^2)$. Thus, the algorithm does not avoid well the redundant computations caused by strong components.

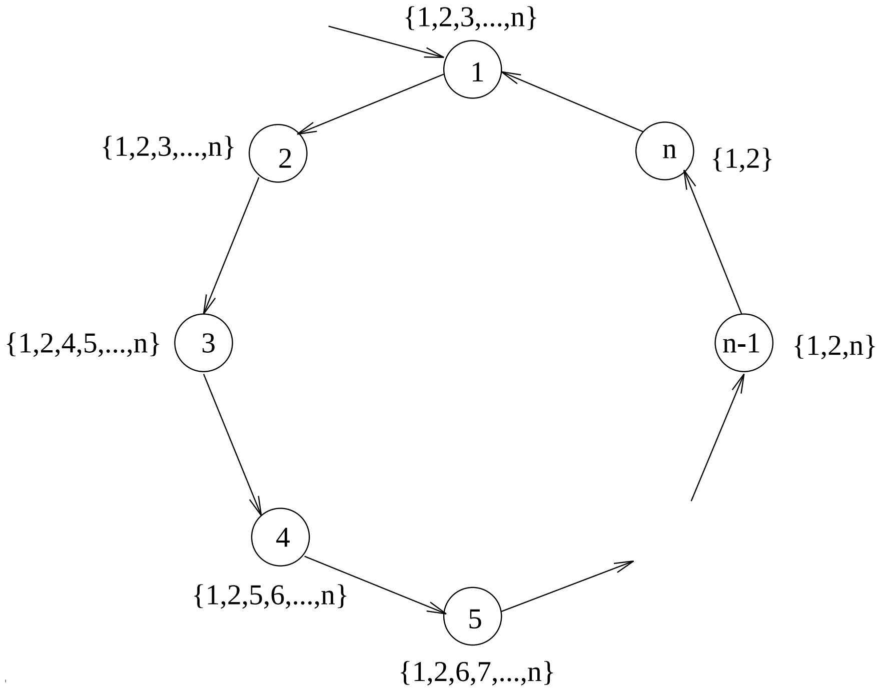

Figure 3: A cycle with partially computed successor sets.

2.2 "Current candidate root" transitive closure algorithm

We now present a new transitive closure algorithm that scans the input graph only once, but handles strong components more efficiently than the simple algorithm above. The algorithm is based on three observations.

First, instead of inserting the successors of a node x into $Succ[x]$, we can insert them into $Succ[root[x]]$, the successor set of the current candidate root node of x. This is a simple heuristic, but it reduces the number and size of partial successor sets that are computed for non-root nodes. It also reduces the number of union operations needed in successor set computation.

Second, all edges are not equally important in transitive closure computation. Edges inside a component can be ignored, since the nodes in component C are in the successor set of C if and only if C is non-trivial, i.e., C contains more than one node (we allow no self loops in the graphs, but they could easily be handled). Forward edges (x, y), where x and y are in different components, are not needed in successor set computation. By definition, if edge (x, y) is a forward edge then y is a descendant of x in the spanning forest induced by the depth-first traversal, but y is not a direct child of x in the spanning forest. There must be a path leading from x to y that consists solely of edges of the spanning tree. Node y will be included into the successor set of x through this path.

Third, instead of inserting nodes into the successor sets we can insert components. This makes the successor sets smaller.

```
(1)   procedure CC_TC(v);
(2)   begin
(3)       root[v] := v; C[v] := Nil;
(4)       PUSH(v, stack);
(5)       for each node w such that (v, w) ∈ E do begin
(6)           if w is not already visited then CC_TC(w);
(7)           if C[w] = Nil then root[v] := MIN(root[v], root[w])
(8)           else if (v, w) is not a forward edge and C[w] ∉ Succ[root[v]] then
(9)               Succ[root[v]] :=Succ[root[v]] ∪ { C[w]} ∪ Succ[C[w]];
(10)      end;
(11)      if root[v] = v then begin
(12)          create a new component C;
(13)          if TOP(stack) = v then Succ[C] :=Succ[v]
(14)          else Succ[C] :=Succ[v] ∪ {C};
(15)          repeat
(16)              w := POP(stack);
(17)              C[w] := C;
(18)              insert w into component C;
(19)              if w ≠ v and Succ[w] ≠ ∅ then Succ[C] :=Succ[C] ∪ Succ[w];
(20)          until w = v
(21)      end
(22)  end;
(23)  begin/* Main program */
(24)      stack := ∅;
(25)      for each node v ∈ V do
(26)          if v is not already visited then CC_TC(v)
(27)  end.
```

Figure 4: The "current candidate root" transitive closure algorithm.

The resulting algorithm is presented in Figure 4. Since the successors are always inserted into the successor set of the current candidate root node, the algorithm is called "current candidate root" transitive closure algorithm, CC_TC for short. At line 9, procedure CC_TC inserts a component $C[w]$ adjacent from node v and the successor set of $C[w]$ into the (partial) successor set of the current candidate root of node v. When a component is fully detected the non-empty partial successor sets of its nodes are combined (at line 19). If the component is nontrivial, i.e., it contains more than one node, the component is included into its own successor set (at line 14). After the execution of the algorithm, $Succ[x]$ can be found in $Succ[C[x]]$.

When algorithm CC_TC is applied to the graph of Figure 3 it generates only one successor set and this successor set contains only one component. However, there are inputs that cause CC_TC to compute some partial successor sets that are not needed after the computation.

2.3 "Component" transitive closure algorithm

Next, we present an algorithm that generates exactly one successor set per each strong component. The algorithm, called COMP_TC, is presented in Figure 5. Like algorithm CC_TC, COMP_TC ignores edges that are not needed in transitive closure computation and collects components instead of nodes into the successor sets. The major difference is that COMP_TC does not compute any successor sets during the depth-first search, but only after a component is fully detected.

```
procedure COMP_TC(v);
begin
    root[v] := v; C[v] := Nil; AdjComp[v] :=∅;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then COMP_TC(w);
        if C[w] = Nil then root[v] := MIN(root[v], root[w])
        else if (v, w) is not a forward edge then
            AdjComp[v] :=AdjComp[v] ∪ {C[w]};
    end;
    if root[v] = v then begin
        create a new component C;
        if TOP(stack) = v then Succ[C] :=∅
        else Succ[C] :={C};
        repeat
            w := POP(stack);
            C[w] := C;
            insert w into component C;
            for X ∈ AdjComp[w] do    /* In topological order */
                if X ∉ Succ[C] then Succ[C] :=Succ[C] ∪ {X} ∪ Succ[X];
        until w = v
    end
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then COMP_TC(v)
end.
```

Figure 5: The "component" transitive closure algorithm.

To avoid scanning again the edges that leave a node v, COMP_TC collects components adjacent from node v into an auxiliary set $AdjComp[v]$. This is done at line 9 of procedure COMP_TC. Tarjan's algorithm detects the strong components in reverse topological order. Procedure COMP_TC uses this to keep the auxiliary sets topologically ordered. When a component is fully detected procedure COMP_TC removes the nodes of the component from the stack. For each node w procedure COMP_TC scans $AdjComp[w]$ in topological order. For a component X in $AdjComp[w]$ procedure COMP_TC adds X and its successor set into $Succ[C]$, the successor set of the new component, if X is not already there. If X already is in $Succ[C]$ then we must have $Succ[X] \subseteq Succ[C]$.

Note that if a component X comes before a component Y in the topological order then Y may be in $Succ[X]$, but not vice versa. Thus, processing $AdjComp[w]$ in topological order is locally optimal, since it yields the minimum number of union operations when processing $AdjComp[w]$. We could combine the $AdjComp$ sets of the nodes of a component before building its successor set. This would yield a globally optimal algorithm with respect to the successor set computation, but it seems that combining the $AdjComp$ sets costs too much.

2.4 Comparison with previous algorithms

An important difference among transitive closure algorithms that use strong component detection is the time at which the successor sets are constructed [8]. Some algorithms construct the successor set of a strong component during the depth-first traversal of component nodes. Other defer the construction until some or all components have been detected.

Ebert's algorithm [4] constructs the successor set of a component during the depth-first traversal of the component nodes. The algorithm suffers from the same weaknesses as SIMPLE_TC. If Ebert's algorithm is applied to the graph in Figure 3 it generates a partial successor set for each node and uses $\Theta(n^2)$ memory.

Algorithm GDFTC by Ioannidis et al. [8] is another early insertion algorithm. Instead of a stack of nodes, GDFTC uses a complicated stack that contains successors sets. By a careful situation analysis GDFTC avoids the weaknesses of SIMPLE_TC. However, according to Ioannidis et al. [8] GDFTC is less efficient than Schmitz's algorithm and algorithm BTC (both discussed below). The problem seems to be the complicated stack mechanism.

Eve and Kurki-Suonio [5] has presented an algorithm that constructs a partial successor set for each node during the depth-first traversal. When a component is detected these partial successor sets are combined to get the full successor set. The algorithm does not propagate successor sets via edges inside a component, but all children of a node are added to its successor set even when they belong to the same component. Further, the algorithm does not avoid forward edges.

Schmitz's algorithm [15] constructs only one successor set for each component. The successor sets contain only component root nodes. This is similar to CC_TC and COMP_TC where the successor sets contain components. Schmitz's algorithm constructs no partial successor sets during the traversal of component nodes. When a component root is detected all children of the component nodes are scanned again and their successor sets are added to the successor set of the root node. Schmitz's algorithm ignores forward edges and edges inside a component when combining successor sets. However, it may insert the component root to its own successor set many times. Schmitz presents a variant of his algorithm that construct an *edge basis*, i.e., a minimal set of edges yielding the same closure. The computation of the edge basis is probably too expensive compared to the savings.

The only practical algorithm that constructs the successor sets when all components are detected seems to be algorithm BTC by Ioannidis et al. [8]. During the component detection BTC does several optimizations that speed up the successor set construction, especially in a disk environment.

3 Representing the successor sets

The most time consuming part of transitive closure computation is the successor set construction. It can be shown that the algorithms presented above require $O(ne)$ time in the worst case whereas the strong component detection takes only $O(n+e)$ time. Our goal is to design a representation that permits fast successor set operations and that can be used efficiently with our transitive closure algorithms. The latter means that the representation can be computed during a single depth-first traversal of the input graph.

Most transitive closure algorithms presented in the literature use common set data structures such as bit matrices, bit vectors, ordered or unordered lists, or relation tables to represent the successor sets. An obvious property of all these data structures is that the representation of a set A requires $\Omega(|A|)$ memory space. The union of two sets requires at least as much memory as the larger of its operands.

3.1 Interval representation

We present a new compact successor set representation that can be used with our transitive closure algorithms. Contrary to the set data structures discussed above, this representation permits a set A to be stored in memory space smaller than $\Omega(|A|)$.

The representation is based on intervals of integers. An interval $[i, j]$ represents numbers $i, i+1, \ldots, j$. A set of integers is represented as a collection of distinct intervals. When new intervals are added to a set, overlapping intervals are merged together. Also, if there are two intervals $[i, j]$ and $[j+1, k]$ they are combined to an interval $[i, k]$. Note that since we merge overlapping intervals, the union of two sets may require less memory space than either of its operands.

The integers that the intervals contain are strong component numbers. The strong components are numbered increasingly in the order they are detected, i.e., in a reverse topological order. This numbering yields a compact representation. For instance, a subtree in the condensation digraph induced by the components is represented as a single interval.

It is easy to see that the maximum numbers of intervals that we need to represent a set of integers $S \subseteq \{1, 2, \ldots, n\}$ is $\lceil n/2 \rceil$. This upper bound is reached when $S = \{1, 3, 5, \ldots, 2\lfloor (n-1)/2 \rfloor + 1\}$. The expected number of intervals needed is harder to define and it depends on the distribution of the integers in S. If we assume that each integer $i \in \{1, 2, \ldots, n\}$ has the same probability of being a member of S

and $|S| = k$ the expected number of intervals needed is $k - k(k-1)/n$.

If the elements of $\{1, 2, \ldots, n\}$ do not have the same probability of being a member of S the expected number of intervals is different. Our experiments suggest that the reverse topological numbering we are using yields a much lower expected number of intervals. We present these experiments in the next section.

A collection of intervals can be implemented in several ways. Since the intervals are distinct, they can be ordered and stored in an ordered binary tree. The binary tree of intervals could be balanced, but usually this is not necessary, since the trees are rather small. To keep the required memory space low, it is a good idea to store only incomplete successor sets as binary trees of intervals. Fully completed successor sets can be stored as ordered arrays of intervals. This reduces disk I/O.

The idea for this representation came from [1] where Agrawal et al. use a similar representation for compressing the computed transitive closure of a directed acyclic graph. The stored transitive closure can be used to directly answer queries about successors of a node without the need for computing the successors on-the-fly. Another goal they have, is to incrementally propagate changes to the stored closure when the base relation changes. Since their goals are different from ours, their strategy is also different.

To get a compact representation Agrawal et al. [1] use an algorithm to compute an optimal spanning tree. The spanning tree is then covered with intervals of postorder numbers. Finally, the nodes of the input graph are processed in the reverse topological order and the sets of intervals are propagated via non-tree edges from the target of the edge to the source of the edge. As we see, their strategy traverses the input graph several times, whereas our strategy needs only one traversal.

When propagating the intervals Agrawal et al. [1] do not merge two adjacent intervals $[i, j]$ and $[j + 1, k]$. In their experiments, the additional compression obtained by merging usually was less than 5%. In our strategy merging is essential, since the single traversal strategy that we use does not necessarily yield an optimal spanning tree.

Another difference is that their inputs are restricted to acyclic digraphs whereas our inputs may have cycles. Our strategy overcomes this problem by detecting the strong components during the traversal.

3.2 Avoiding multiple paths

In Section 1 we pointed out two sources of redundancy that should be avoided when computing the transitive closure. So far we have only discussed the redundancy caused by strong components. We consider now the redundancy caused by multiple paths between two nodes and show how the interval representation efficiently copes with this kind of redundancy.

Assume that a transitive closure algorithm that is based on Tarjan's algorithm is applied to the graph in Figure 1. There are two paths from a to f, namely a, b, d, f and a, c, e, f. The depth-first traversal uses both paths, but node f and the subtree rooted at f are entered only once. Thus, it seems that the algorithm avoids well the redundancy caused by multiple paths. Unfortunately, the successor set of f is propagated from f to a via both paths and is twice added to the successor set of a.

Jakobson [9] presents a transitive closure algorithm that avoids this kind of redundancy. The idea is to use successor trees instead of successor sets. A successor tree contains the same nodes as a successor set, but in addition it contains the paths that were used to obtain these nodes. Multiple paths are detected and the same nodes are not added again to the successor tree. The cost that has to be paid is the space required by the successor trees and the time needed to construct them. Another problem is that the method does not detect strong components.

When common set data structures, such as ordered lists or relation tables, are used to represent the successor sets, taking the union of two sets is the most time consuming operation. Adding the same successor set twice to another set should be avoided. But when we use the interval representation, a union operation usually means processing only a small number of intervals. Adding a few intervals twice to a successor set is not expensive. Interval merging often reduces the costs further.

The following example is taken from Jakobson's paper [9]. The graph in Figure 6 has $n = m^2$ nodes that are ordered as a square of m columns of m nodes. For every node in column k, $1 \leq k \leq m-1$, there is an edge to every node in column $k+1$. As we see, each node in column k has the nodes in columns $k+1, k+2, \ldots, m$ as its successors. The size of the successor set is $\Theta(m^2)$ for most nodes.

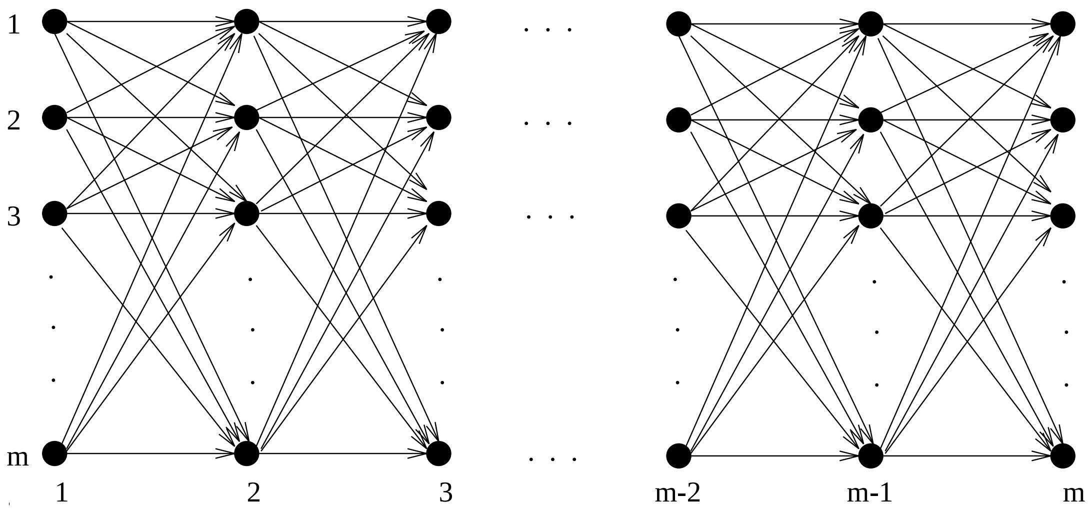

Figure 6: An example from Jakobson's paper[9].

Assume that we apply CC_TC to this graph. Since the graph has no cycles, each node has an own component. To compute the successor set of a component x in column k we must insert the components in column $k+1$ into $Succ[x]$. Further, we must add the successor sets of the components in column $k+1$ into $Succ[x]$. Although these sets are equal we must combine m sets of $\Theta(m^2)$ components. This takes $\Omega(m^3)$ time when common set data structures are used. Since we have m^2 components, the computation takes $\Omega(m^5) = \Omega(n^2\sqrt{n})$ time.

Jakobson's algorithm requires only $O(n^2)$ time to compute this transitive closure, since it uses only one path from a node x in column k to a node y in column $k+2$.

Assume that CC_TC is used together with the interval representation. When we traverse the graph in depth-first order the components in column m get the reverse topological numbers $1, 2, \ldots, m$. Similarly, the components in column k get the numbers $m(m-k)+1, m(m-k)+2, \ldots, m(m-k+1)$.

The successor set of a component in column k is the single interval $[1, m(m-k)]$. To construct the successor set of a component C in column $k-1$, we add the reverse topological numbers and the successor sets of the components in column k into $Succ[C]$. Thus, we insert m numbers $m(m-k)+1, m(m-k)+2, \ldots, m(m-k+1)$ into $Succ[C]$ and we insert m times the interval $[1, m(m-k)]$ into $Succ[C]$.

If the intervals are stored in ordered balanced binary trees these insertions require $O(m \log m)$ time. Since the graph has m^2 nodes, the total time required is $O(m^3 \log m)$ which is $O(n\sqrt{n} \log n)$. Thus, the interval representation is here more efficient than Jakobson's successor tree method.

4 Simulation studies

In this section we describe simulation experiments we did to compare the efficiency of our new transitive closure algorithms with some previous algorithms. We also compared the storage requirements of different representations. As inputs we used randomly generated graphs.

4.1 Simulation method

The results of simulation studies often have little credibility, since they are presented without regard to their random nature and the need for proper statistical analysis of simulation output data [13]. If the simulation model is controlled by random numbers drawn from some distributions also the outputs are usually random variables having some distribution. Often the authors of simulation studies make a single simulation run of somewhat arbitrary length and treat the resulting simulation estimate as being the "true" measure of performance. Since this estimate is a realization of a random variable, which may have a large variance, it may differ greatly from the corresponding true measure [11].

A standard statistical technique to assess the accuracy of an estimate is to construct a confidence interval for the estimated parameter. A $100(1-\alpha)\%$ confidence interval for a parameter θ is an interval $[L_1, L_2]$ that contains the actual value of θ with probability $(1-\alpha)$. The confidence interval is constructed using the collected samples. See, e.g., [3] for the statistical details.

To construct the confidence interval of the estimated parameter we used a *sequential procedure* [11, 13]. In a sequential procedure, the length of the simulation is increased sequentially until the required confidence interval is obtained. This is practical, since it is often impossible to know in advance the simulation run length needed to meet the required accuracy.

To estimate the expected value of a parameter p, we proceeded in the following way. First, generate a simulation input I. Apply an algorithm to compute the value of parameter p in input I. Next, update the sample mean that is used as the estimate. If the estimate has reached the required confidence interval, stop the simulation. Otherwise continue with a new input. To get reliable results at least 30 inputs has to be generated.

All simulation results that we present were produced with this procedure. We used a 90% confidence interval with 10% relative width. More accurate results can easily be produced, but it takes more simulation time.

4.2 Inputs

As simulation inputs we used randomly generated graphs.

A standard model of random graphs is $G(n, p)$ where a graph drawn from $G(n, p)$ has n nodes and any two nodes are joined by an edge with probability p independently of the existence of any other edge in the graph [2].

The graphs drawn from $G(n, p)$ are usually very similar in some respects. For instance, the distribution of the sizes of the strong components in two directed random graphs is likely to be very similar. Karp [10] has shown that, when n is large and np (the expected outdegree) is equal to a constant c greater than one, it is very likely that the graph contains one large component and the other components are very small. Figure 7 (a) presents the expected percentage of nodes that belong either to the large component or to the trivial components of one node. The inputs were random graphs drawn from $G(n, p)$ with several values of n and np. As we see, nearly all nodes are in the large component or in trivial components. If we use graphs drawn from $G(n, p)$ as simulation inputs we get no information about graphs with many nontrivial components.

To overcome this problem we used another model of directed random graphs $G(n, p, l)$. The graphs in $G(n, p, l)$ have n nodes. Parameter l, which is called the *locality*, limits the set of possible edges. For a node i, the set of possible edges is $\{(i, j) \mid (i - l + n) \bmod n \leq j \leq (i + l) \bmod n \wedge i \neq j\}$. Each edge of this set exists with probability np/l independently of the existence of any other edge in the graph. Figure 7 (b) presents the expected percentage of nodes that belong either to the large component or to the trivial components of one node when the graphs were drawn from $G(n, p, l = 5)$. When the expected outdegree is between two and four, most nodes are in non-trivial components other than the large component. Note that the behavior only depends on the outdegree, not on the number of nodes. Ioannidis et al. [8] use a similar model in their simulations.

Results similar to those of Figure 7 (b) were obtained with other small values of l. When l grows, the model produces similar graphs as $G(n, p)$. Thus, $G(n, p, l)$ suits best for studying sparse graphs. Most graphs in practical applications are sparse.

4.3 The size of the transitive closure

We did a series of experiments to estimate the memory space required to represent the transitive closure. As inputs we used random graphs from $G(n, p, l = 10)$. We used algorithm COMP_TC with the interval representation to compute the closures. From the closures we computed the values for the estimated parameters.

The first parameter is the memory space required when strong components are not handled. There is one successor set for each node and the successor set contains nodes. The successor sets are represented as lists. One list element requires one memory word. The second parameter is like the first, but now strong components are detected, only one successor set is constructed for each component, and the successor sets contain nodes. The third parameter is like the second, except that the successor sets contain components. The fourth parameter is the memory space

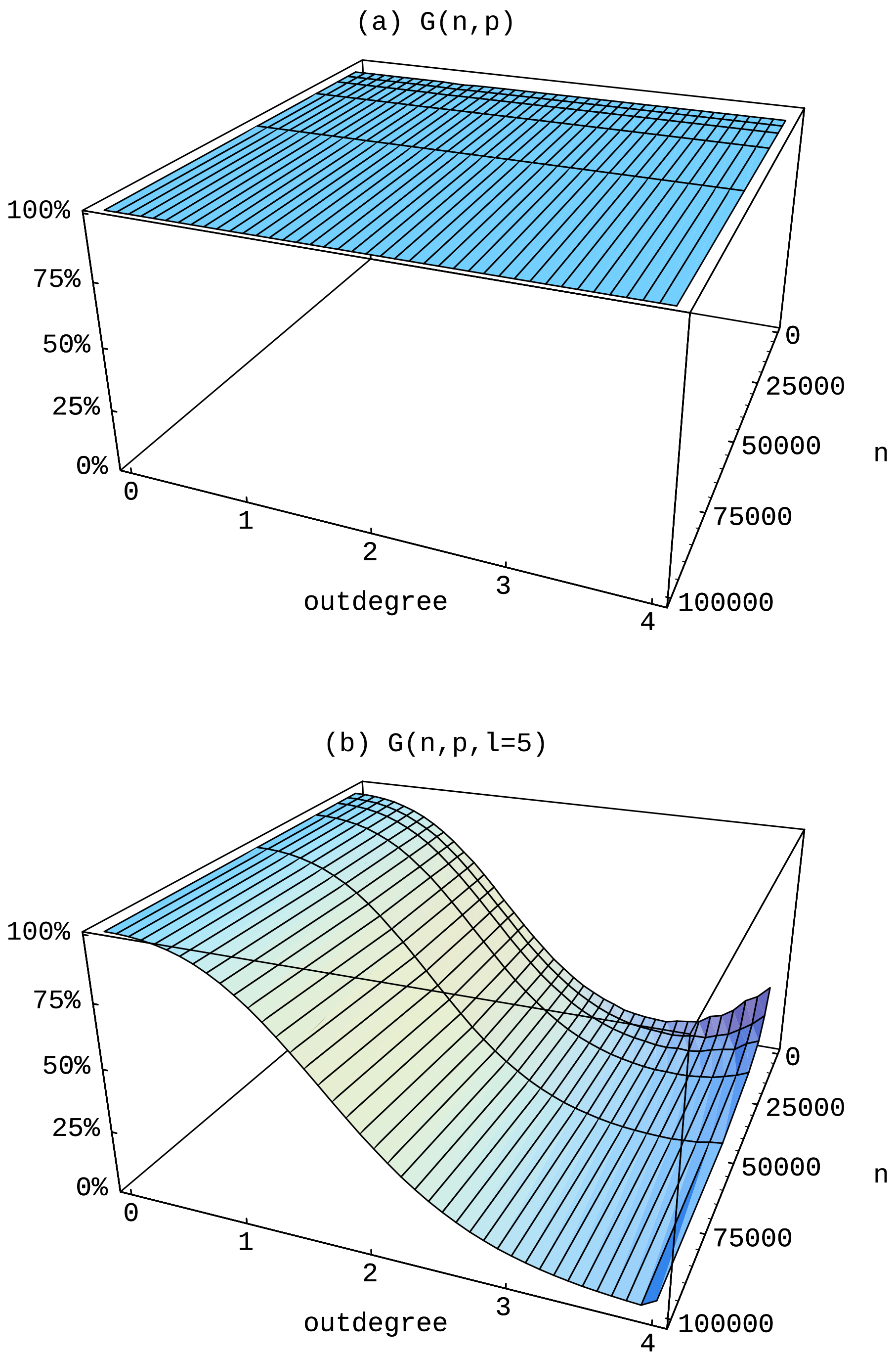

Figure 7: Percentage of nodes in the large component and in the trivial components.

required when one successor set is constructed for a component, and the successor sets are represented as a lists of intervals. One interval requires two memory words.

The results are presented in Figures 8–11. Note that we have divided the values of the estimated parameters by n, the number of nodes. Note also that every picture has a different scale on the z-axis.

Figure 8 shows the memory space, divided by the number of nodes, when strong components are not detected. When the expected outdegree is over four, the memory requirement is $\Theta(n^2)$. Storing the closure of a graph with 100,000 nodes and outdegree over five requires 10^{10} memory words.

Figure 9 shows the memory space, divided by the number of nodes, when there is only one successor set per component and the sets contain nodes. The representation is much more compact than in Figure 8. The memory space first increases when the number of edges in the closure grows, but starts to decrease when the number of components decreases. When outdegree is about seven, the graphs are usually strongly connected and $\Theta(n)$ memory is needed. Note that between outdegrees three and four the memory space is not linear to n. Storing the closure of a graph with 100,000 nodes and outdegree about four requires 80,000,000 words.

Figure 10 shows the memory space, divided by the number of nodes, when there is only one successor set per component and the sets contain components. The memory requirements is orders of magnitude smaller than in Figure 9. Also here, most space is required between outdegrees three and four, and the memory space is not linear to n. Storing the closure of a graph with 100,000 nodes and outdegree about four takes 1,000,000 words.

Figure 11 shows the memory space, divided by the number of nodes, when the interval representation is used. The memory space required is linear to n with every outdegree. The largest memory requirement is about $2.5n$ and is attained at outdegree 1.3. Storing the closure of a graph with 100,000 nodes and outdegree about 1.3 takes 250,000 words.

4.4 Successor set construction

In these experiments we compared the amount of work required to construct the successor sets by our new algorithms and by some previous algorithms that use strong components detection.

Based on the observations presented in Section 2.4 we chose Schmitz's algorithm [15] and algorithm BTC by Ioannidis et al. [8] for the comparison. In the implementation of these algorithms we represented the successor sets as lists. In BTC, the lists contained nodes and in Schmitz's algorithm they contained components. A bit vector was used to speed up membership queries when constructing a successor set. Since the representation used by BTC is so wasteful, we implemented another version of BTC, here called BTC_I, that uses the interval representation. With our new algorithms, we used the interval representation for successor sets.

As a measure of successor set construction work we used the number of basic elements accessed. For lists, this is the number of nodes accessed and for interval representation the number of intervals accessed. Obviously, the access time for a single element is not the same for intervals and for lists. Note also that these results do not contain the operations needed to scan the input graph (which BTC,

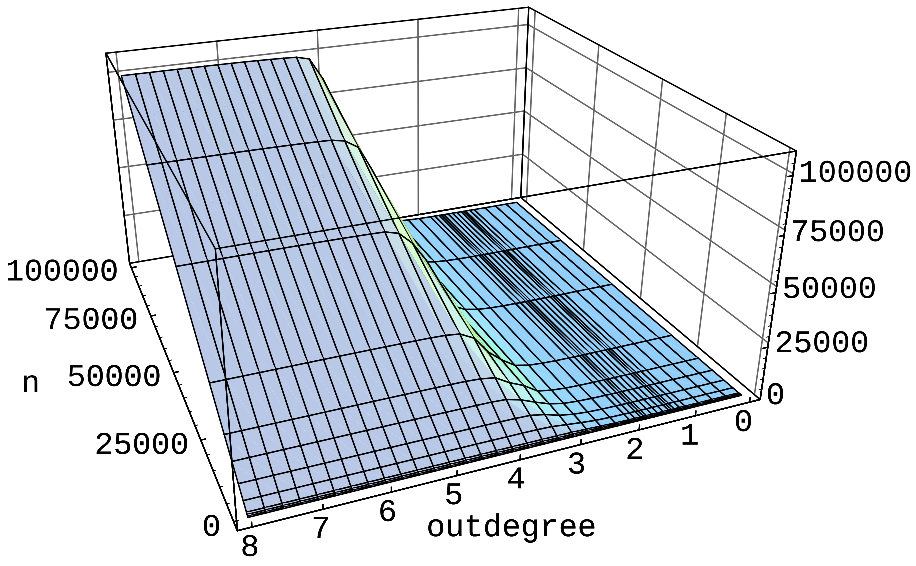

Figure 8: Memory space required divided by the number of nodes. Components not detected.

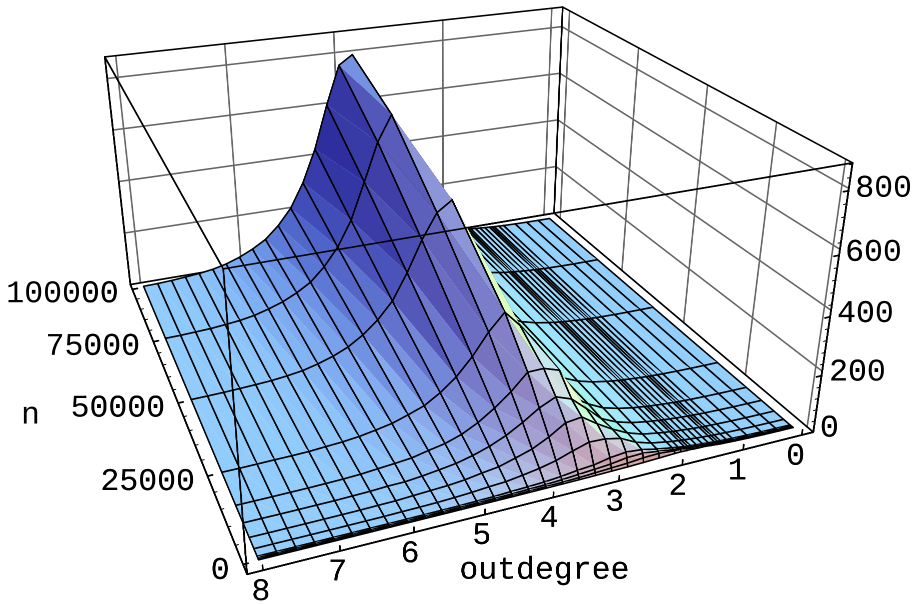

Figure 9: Memory space required divided by the number of nodes. One successor set of nodes per component.

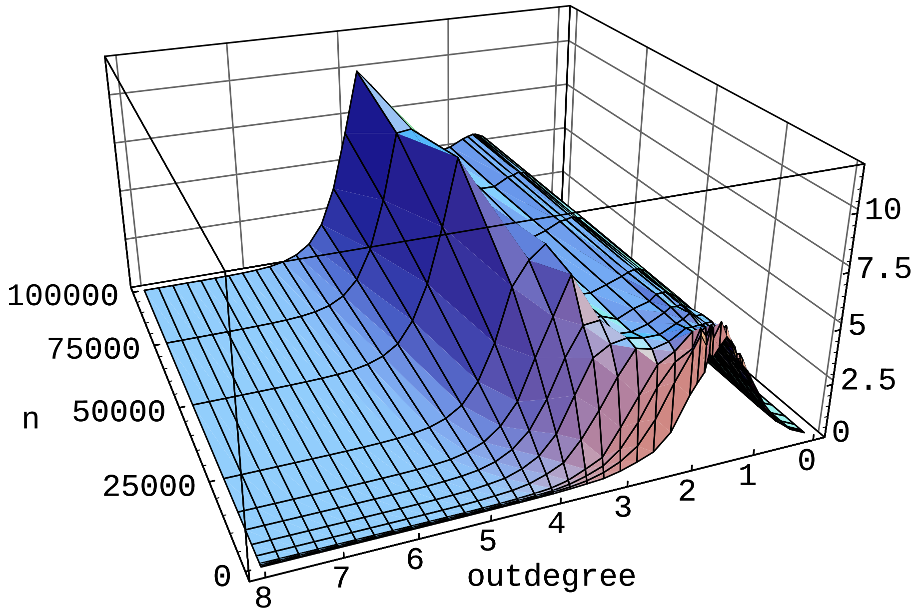

Figure 10: Memory space required divided by the number of nodes. One successor set of components per component.

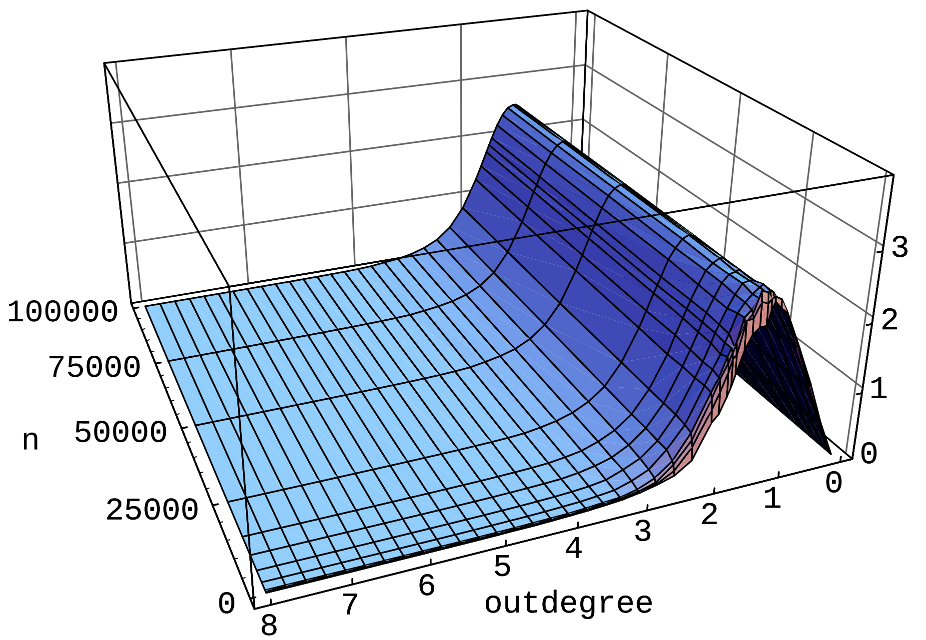

Figure 11: Memory space required divided by the number of nodes. Interval representation.

BTC_I, and Schmitz's algorithm do twice). Nor do they contain the operations for topologically ordering the children nodes (done by BTC and BTC_I) or the adjacent components (done by COMP_TC). Thus, the results cannot be directly translated into CPU-times or other real-world performance metrics.

As inputs we used random graphs drawn from $G(n, p, l)$ with several values of n, np (the expected outdegree), and l. We present here the simulation results for $G(n = 10000, p, l = 5)$. Similar results were obtained with other values of n and l.

The results are in Figure 12. COMP_TC was the most economical. The largest number of element accesses COMP_TC needed for graphs with outdegrees between one and two. When the outdegree increased the number of accesses rapidly decreased. CC_TC was almost as good as COMP_TC. It seems that sorting the adjacent components topologically does not reduce the number of accesses very much. The partial successor sets for a few non-root nodes do not cost much in CC_TC.

BTC and Schmitz required many more accesses. Both reached their highest values near outdegree three. BTC required so many accesses that its curve does not fit in the same plot. Note that both BTC and Schmitz did set accesses even when the input graph was strongly connected.

BTC_I was several times more efficient than BTC. Obviously, using the interval representation is a good idea even when the algorithm scans the input graph more than once. Still, BTC_I required more accesses than CC_TC or COMP_TC.

Finally, we studied how the number of basic element accesses changes in COMP_TC when the number of nodes changes. Figure 13 shows the expected number of accesses divided by the number of nodes. Note that the number of basic element accesses contains here also the accesses required for storing the *AdjComp* sets in topological order. As we see, the number of accesses that COMP_TC needs grows linearly to the size of the input when n varies between 100 and 100,000 nodes.

5 Conclusions

Transitive closure is an important problem in many areas. Efficient transitive closure computation requires the elimination of redundant computations caused by strong components and multiple paths between nodes.

We have presented two new transitive closure algorithms that handle strong components better than previous algorithms. The algorithms scan the input graph only once, but do not generate a partial successor set for each node like some previous algorithms. In simulation experiments the new algorithms outperformed Schmitz's algorithm [15] and algorithm BTC by Ioannidis et al. [8].

We have presented a representation for the successor sets that in addition of saving much space, reduces the work for constructing the successor sets. We have shown how the representation reduces problems caused by multiple paths between nodes.

The next subject to be studied is the behavior of the new algorithms in a disk environment. The low numbers of operations needed in the successor set construction suggests that the algorithms perform well also in this case. A careful analytical study of the algorithms and the representation should also be done.

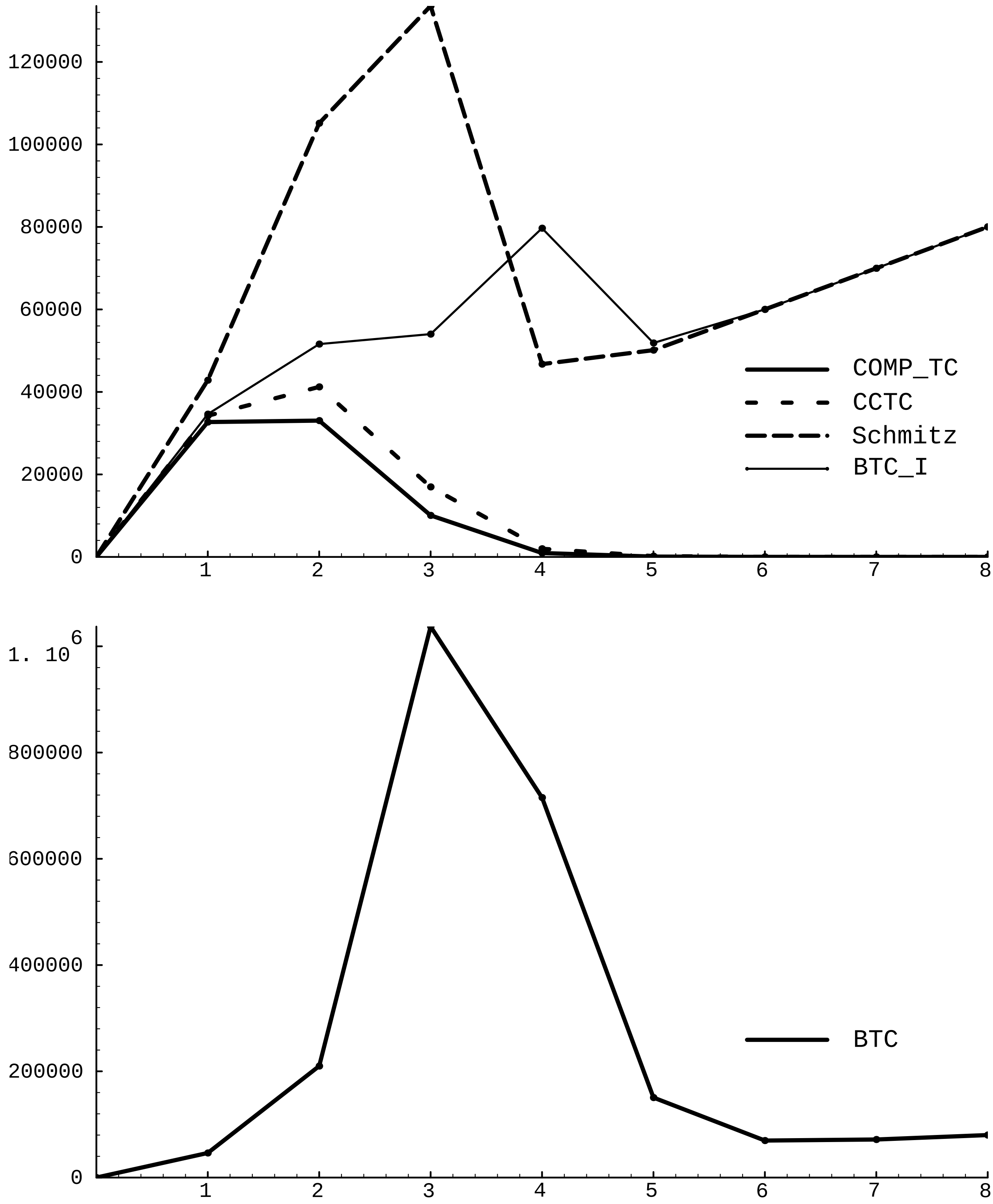

Figure 12: Basic element accesses in successor set construction.

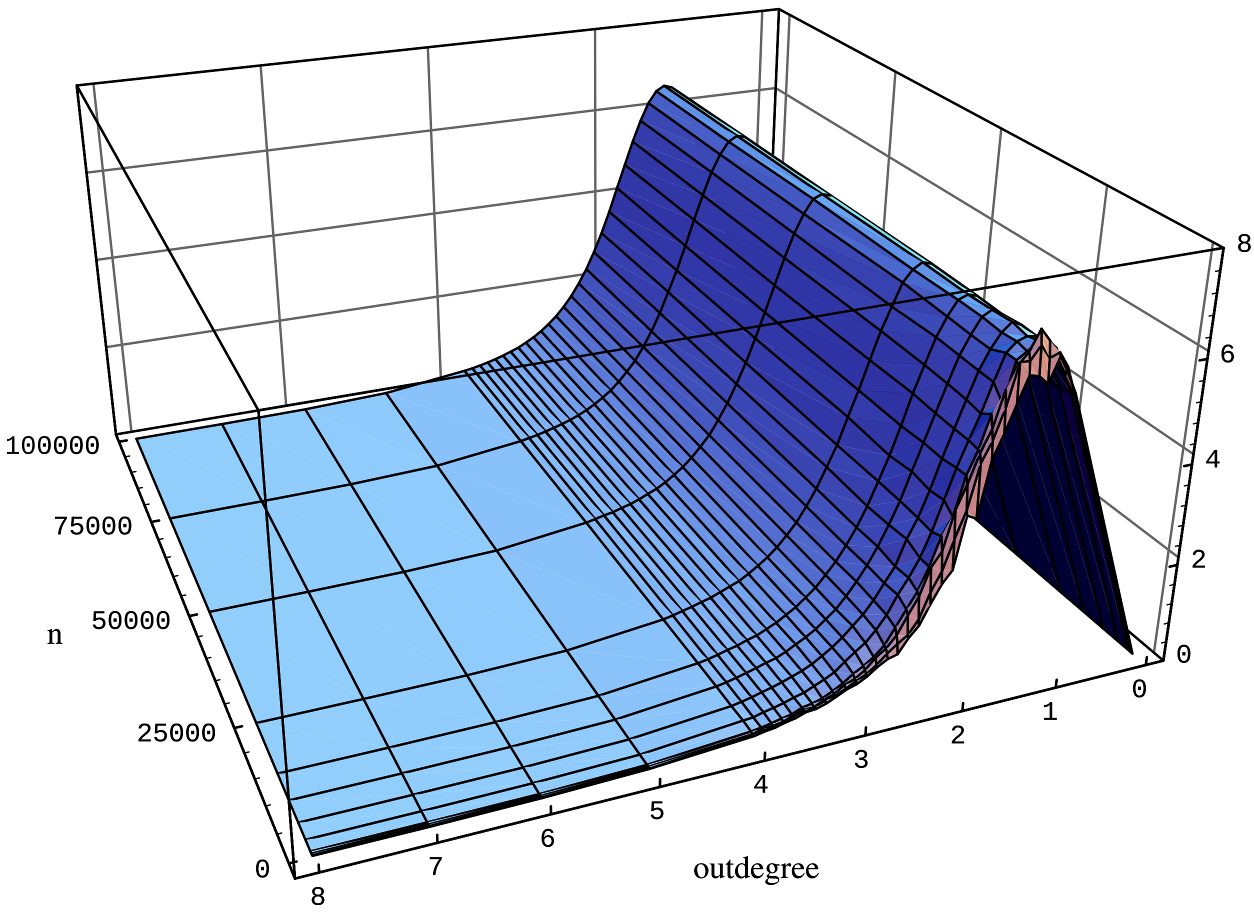

Figure 13: Basic element accesses in COMP_TC divided by the number of nodes.

Acknowledgements

We thank Ilkka Karanta and Pentti Laininen for valuable comments on the simulation experiments.

References

[1] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM-SIGMOD 1989 Conference on Management of Data*, pages 253–262, Portland, Oregon, May-June 1989.

[2] B. Bollóbas. *Random Graphs.* Academic Press, New York, 1985.

[3] E.J. Dudewicz and N.M. Satya. *Modern Mathematical Statistics.* John Wiley & Sons, Inc., New York, 1988.

[4] J. Ebert. A sensitive transitive closure algorithm. *Information Processing Letters*, 12:255–258, 1981.

[5] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.

[6] S. Ganguly, R. Krishnamurthy, and A. Silberschatz. An analysis technique for transitive closure algorithms: A statistical approach. In *Proceedings of the IEEE 7th International Conference on Data Engineering*, pages 728–735, Kobe, Japan, 1991.

[7] Y.E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proceedings of the 14th VLDB Conference*, pages 335–346, Los Angeles, California, 1988.

[8] Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, September 1993.

[9] H. Jakobson. Mixed-approach algorithms for transitive closure. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, pages 199–205, 1991.

[10] R. M. Karp. The transitive closure of a random digraph. *Random Structures and Algorithms*, 1(1):73–93, 1990.

[11] A.M. Law and W.D. Kelton. Confidence intervals for steady-state simulations, II: A survey of sequential procedures. *Management Science*, 28(5):550–562, May 1982.

[12] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[13] K. Pawlikowski. Steady-state simulation of queueing processes: A survey of problems and solutions. *ACM Computing Surveys*, 22(2):123–170, June 1990.

[14] P. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[15] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30:359–371, 1983.

[16] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[17] H.S. Warren. A modification of warshall's algorithm for the transitive closure of binary relations. *Communications of the ACM*, 18(4):218–220, 1975.

[18] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.