An efficient transitive closure algorithm for cyclic digraphs

Esko Nuutila

Laboratory of Information Processing Science

Helsinki University of Technology

Otakaari 1, FIN-02150 Espoo, Finland

enu@cs.hut.fi

Abstract

We present a new transitive closure algorithm that is based on strong component detection. The new algorithm is more efficient than the previous transitive closure algorithms that are based on strong components detection, since it does not generate unnecessary partial successor sets and scans the input graph only once.

Keywords: Design of algorithms; transitive closure; strong components; Tarjan's algorithm; random graphs; simulation

1 Introduction

We consider a directed graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Without loss of generality, we assume that G has no self loops (v, v). The graph is represented as a set of adjacency lists $Adj(v) = \{w \mid (v, w) \in E\}$. 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 non-null path from v to w in G. The *successor set* of a vertex v in G is the set $Succ(v) = \{w \mid (v, w) \in E^+\}$. Two vertices 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 vertices. These sets are called the *strong components* of the graph.

All vertices in a strong component have the same successor sets. Several transitive closure algorithms presented in literature use graph search and strong component detection to avoid redundant computations [2, 3, 4, 8, 9]. These algorithms use Tarjan's algorithm [10] or some related depth-first search based algorithm for strong component detection. The algorithms can be classified with respect to the time point at which the successor sets are constructed during the computation [4]. Eve's and Kurki-Suonio's algorithm [3], Ebert's algorithm [2] and the algorithm GDFTC by Ioannidis et al. [4] construct the successor sets during the strong component detection. Purdom's algorithm [8] and the algorithm BTC by Ioannidis et al. [4] first detect all strong components and after that construct the successor sets. Schmitz's algorithm [9] constructs the successor set for a strong component immediately after detecting the component.

The problem with the algorithms that construct the successor sets during the strong component detection is that they generate partial successor sets for each vertex of a component and combine them, instead of constructing just one successor set per each component. This may cause many unnecessary set operations. The problem with the algorithms that construct the successor set of a component after detecting the component or after detecting all components is that they scan the input graph at least twice.

We present a new transitive closure algorithm that generates just one successor set per each component without scanning the input graph twice. The algorithm uses Tarjan's algorithm for strong component detection. During the detection of a strong component, the algorithm collects the adjacent components that are later needed in constructing the successor set of the component. An auxiliary stack, resembling the vertex stack of Tarjan's algorithm, is used for this purpose. When a component C is detected the components that were stored onto the stack during the detection of C are removed from the stack and the successor set of the component is computed by combining these components and their successor sets. A variant of our algorithm sorts the components in the stack in a topological order before computing the successor set to minimize the number of union operations needed.

Of the previous algorithms, the new algorithm resembles most Schmitz's algorithm [9], since both algorithms construct the successor set of a component immediately after detecting the component. We present experimental results showing that the new algorithm outperforms Schmitz's algorithm. The new algorithm also outperforms a previous transitive closure algorithm that we presented in [6].

2 The algorithm

The new algorithm is presented in Figure 1. It consists of a recursive procedure COMP_TC that traverses the vertices in a depth-first order and a main program that applies the procedure to all vertices that are not already visited. The procedure does two tasks. First, it detects the strong components and second, after detecting a strong component it constructs a successor set for the component.

We first review the strong component detection. For each strong component C, the first vertex of C that is entered is the *root* of component C. To detect the roots, we define a variable *root*(v) for each vertex v. When COMP_TC is processing vertex v, *root*(v) contains a candidate vertex for the root of the component containing v. Initially (at line 3), vertex v itself is the root candidate. When COMP_TC processes the edges leaving vertex v (at lines 6–11), new root candidates are obtained from the children vertices that belong to the same component as v. The MIN operation (at line 8) compares the vertices with respect to the order in which COMP_TC has entered them, i.e., $\text{MIN}(x, y) = x$ if COMP_TC entered vertex x before it entered vertex y, otherwise $\text{MIN}(x, y) = y$. When COMP_TC has processed all edges leaving v, *root*$(v) = v$ if and only if v is a component root (line 12). To distinguish between vertices belonging to the same component as vertex v and vertices belonging to other components, a variable $C(w)$ is defined for each vertex w. Its initial value is *Nil*. When a component C is fully detected COMP_TC sets $C(w) := C$ for each vertex w that belongs to C (line 22). For this purpose the algorithm uses a stack called *nstack*. Each vertex is stored on *nstack* in the beginning of COMP_TC. When the component is fully detected the vertices belonging to it are on top of the stack.

COMP_TC removes them from the stack, sets their $C(w)$ variables, and inserts them into component C.

In constructing the transitive closure of a graph G we use the *condensation graph* G_c induced by the strong components of G. The condensation graph is an acyclic graph that has the strong components of G as its vertices and an edge (X, Y) if X and Y are two different strong components and X contains a vertex u and Y contains a vertex v that are connected by an edge (u, v) in G. Since G_c is acyclic, its transitive closure is easier to construct than the closure of G. Purdom [8] noticed that the transitive closure of an acyclic graph can be computed by first sorting its vertices in a topological order and then constructing the successor sets in the reverse topological order. The successor set of a vertex x is constructed by combining the vertices adjacent from x and their successor sets. The topological order ensures that the successor sets of the adjacent vertices are already constructed.

In our algorithm, we benefit from the property of Tarjan's algorithm that the components are detected in a reverse topological order, i.e., if there is a path from a component C to a different component C' then C' is detected first. Thus, no separate topological sorting of the components is needed. The successor sets constructed in this way contain strong components, represented as their reverse topological numbers. To save space and time, we do not expand these successor sets to contain vertices of the original graph G. Since our algorithm stores the members of each component, the output of our algorithm can be used to answer queries about the successors of some vertex x as well.

Thus, to compute $Succ(C)$ we have to collect all components adjacent from C. COMP_TC does this during the depth-first traversal of the vertices of a component C.

For this purpose the algorithm uses another stack, called *cstack*. At line 8 COMP_TC checks if edge (v, w) leads to another component. Vertex w is in another component if and only if $C(w) \neq Nil$. If w is in another component and (v, w) is not a *forward edge* COMP_TC inserts $C(w)$ into *cstack*. By definition, if (v, w) is a forward edge then COMP_TC has already visited w during COMP_TC(v) via some path p before (v, w) is checked. Thus, the component C' containing w and $Succ(C')$ are inserted into $Succ(C)$ via path p and edge (v, w) can be ignored.

When entering a vertex v, COMP_TC stores the current height of *cstack* to a local variable $hsaved(v)$. When a component C with root vertex r is detected, the components that are needed in constructing $Succ(C)$ are between the top of *cstack* and $hsaved(r)$. At lines 16–19 COMP_TC removes these components from the stack, and for each component X checks if X already is in $Succ(C)$. If it is, then obviously the contents of $Succ(X)$ must also be in $Succ(C)$. If it is not, then COMP_TC adds X and $Succ(X)$ into $Succ(C)$. Note that the contents of *cstack* is the same after exiting root vertex r as it was before entering r. In this respect *cstack* behaves similarly to *nstack*.

The final point in constructing the successor set of component C is the inclusion or exclusion of C itself into $Succ(C)$. Obviously, component C is in its own successor set if and only if C contains more than one vertex[1]. This is realized at lines 14–15 in COMP_TC.

The benefits from collecting the adjacent components onto *cstack* are twofold. First, we avoid scanning twice all edges leaving the vertices of a component C.

[1]If we allowed self loops C would be in $Succ(C)$ also if $C = \{v\}$ and (v, v) is an edge of the input graph.

Second, the components are stored consecutively onto the stack. If the input does not fully fit into the main memory, the traversal of the stack requires much less paging operations than accessing the adjacency lists of the vertices of the components.

We have designed a variant of COMP_TC that uses *cstack* to minimize the number of union operations needed in successor set construction. Consider a component C and two components X and Y adjacent from C. If X is topologically greater than Y then Y may be in $Succ(X)$, but not vice versa. Processing X before Y at line 18 in COMP_TC may eliminate the need to insert Y and $Succ(Y)$ into $Succ(C)$. We can achieve this if we sort the components on *cstack* between the top and $hsaved(r)$ in topological order before line 16. The sorting is done in the following way. Let r be the root of component C. Scan the components on *cstack* between the top and $hsaved(r)$ and use a bit vector to record the components that are present, removing duplicates. If the number of unique components remaining on *cstack* above $hsaved(r)$ is small then sort them into the topological order using some common sorting algorithm. Otherwise scan the bit vector to obtain the components directly in the topological order.

3 Comparisons

The worst case run time of the new algorithm is the same as in the previous algorithms [2, 3, 4, 8, 9] that are based on strong component detection, i.e., $O(ne+n+e)$, where n is the number of vertices and e is the number of edges in the input graph. The worst case size of *cstack* is e.

In the variant that sorts *cstack* to minimize the number of union operations, the worst case time required for sorting the adjacent components of a component C (with root r) is $O(\min(s \log s, s+n))$, where s is the number of components on *cstack* between the top and *hsaved*(r). It is easy to see that the total time required for sorting in computing all successor sets is $O(e+n^2)$.

We used simulations to estimate the expected behavior of the new algorithm. To get reliable estimates we used a *sequential simulation procedure* [7] and constructed a 95% confidence interval with 5% relative error for each estimated parameter, i.e., the error in the expected values that we present is maximally 5% with probability 95%.

As inputs to simulations we used directed random graphs. The standard model of random graphs is $G(n,p)$, where a graph drawn from $G(n,p)$ has n vertices and any two vertices are joined by an edge with probability p independently of the existence of any other edge [1]. The problem with this model is that a graph drawn from $G(n,p)$ has almost all of its vertices in one large strong component and in trivial single vertex components [5]. Thus, $G(n,p)$ gives us no knowledge on graphs with many nontrivial components. To overcome this problem we used a slightly different model of random graphs $G(n,d,l)$. Parameter d is the expected out-degree and corresponds np in model $G(n,p)$. Parameter l, which is called the *locality*, limits the set of possible edges. For a vertex 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 d/l independently of the existence of any other edge in the graph. Figure 2 presents the estimated proportion of vertices that are outside the large component and the single vertex components. The inputs are drawn from $G(n,d,l=5)$ using different values for n and d. When d is between two and four, most vertices are in non-trivial components other than the large component. Results similar to those of Figure 2 were obtained with other small values of l. When l grows, the graphs drawn from $G(n, d, l)$ start to resemble graphs drawn from $G(n, d/p)$. Since $d \leq 2l$, the model $G(n, d, l)$ suits best for studying sparse graphs with many nontrivial components.

In the first experiment we studied the expected number of components stored onto *cstack* and the expected maximum height of *cstack* during the execution of COMP_TC. The graphs were drawn from $G(n, d, l = 5)$ where d varied between 0 and 10, and n varied between 1000 and 50000. Figure 3 presents the expected number of component stored onto *cstack* during the execution of COMP_TC divided by the number of vertices. As we see, at all values of n, the maximum number of components were stored onto the stack when the expected out-degree d was about 1.5. The expected number of components stored onto *cstack* was always smaller than n. Figure 4 presents the expected maximum height of *cstack* during the execution of COMP_TC divided by the number of vertices. The maximum height of *cstack* grew slowly when n grew. This parameter had its maximum value between 3 and 4. Thus, although the maximum height of *cstack* is e in the worst case, usually a much smaller amount of memory suffices for *cstack*.

In the second experiment we studied the expected run time of COMP_TC. Of the previous algorithms that are based on strong component detection our algorithm resembles most Schmitz's algorithm [9]. According to [4], Schmitz's algorithm seems to be the fastest strong component based transitive closure algorithm with respect to CPU-time. For these reasons we compared the expected run time of our new algorithm with Schmitz's algorithm.

In this experiment we represented the input graph as a table of vertices that contains pointers to a table containing the adjacency lists. The output was a table of components containing the vertices of the components and pointers to the successor sets. A successor set was represented as a linked list of component numbers. To speed up membership lookups in successor set construction we used a bit vector. *nstack* and *cstack* were implemented as arrays. Recursion was removed from the algorithms and they were implemented in C++ and run on a Sun Sparcstation 10.

Figure 5 presents the estimates of the expected CPU-times in milliseconds required by Schmitz's algorithm and COMP_TC when the inputs are drawn from $G(n, d, l)$ where $l = 5$, and d varies between 0 and 10. In Figure 5(a) $n = 1000$ and in Figure 5(b) $n = 10000$. As we see, the CPU-times of both algorithms are rather similar when d is between 0 and 1. When d increases, COMP_TC becomes considerably faster than Schmitz's algorithm. At $d = 10$, COMP_TC is about three times faster than Schmitz's algorithm.

To understand the difference in the CPU-times we refer to Figure 3. When d is between 0 and 1, the number of components stored onto *cstack* is near the number of edges in the graph, but it decreases rapidly as d grows. Schmitz's algorithm scans every edge twice, once in the depth-first traversal and once when constructing the successor set of a newly detected component. COMP_TC scans each edge once during the depth-first traversal and, in addition, scans the components on *cstack*. Thus, when d is between 0 and 1 the total number of "items" scanned in COMP_TC is about the same as in Schmitz's algorithm, but it decreases to the half of items scanned in Schmitz's algorithm when d grows. The time required for union operations seems to dominate when d is small, but its effect diminishes when d is larger. Further, Schmitz's algorithm does not handle separately the inclusion of a strong component to its own successor set, which leads to one unnecessary successor set insertion operation per each edge inside a component.

We obtained similar results with other values of n and l and also with the random graph model $G(n, p)$. When the input graphs were drawn from $G(n, p)$ when p was near 1 our algorithm outperformed Schmitz's algorithm by a factor 3.12. In other experiments we have found out that the new algorithm also always outperforms the transitive closure algorithm that we presented in [6].

References

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

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

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

[4] 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.

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

[6] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49:9–14, 1994.

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

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

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

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

```
procedure COMP_TC(v);
begin
    root(v) := v; C(v) := Nil;
    PUSH(v, nstack);
    hsaved(v) := HEIGHT(cstack);
    for each vertex 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
            PUSH(C(w), cstack);
    end;
    if root(v) = v then begin
        create a new component C;
        if TOP(nstack) = v then Succ(C) := ∅
        else Succ(C) := {C};
        while HEIGHT(cstack) ≠ hsaved(v) do begin
            X := POP(cstack);
            if X ∉ Succ(C) then Succ(C) := Succ(C) ∪ {X} ∪ Succ(X);
        end;
        repeat
            w := POP(nstack);
            C(w) := C;
            insert w into component C;
        until w = v
    end
end;
begin/* Main program */
    nstack := ∅; cstack := ∅;
    for each vertex v ∈ V do
        if v is not already visited then COMP_TC(v)
end.
```

FIGURE 1: Algorithm COMP_TC.

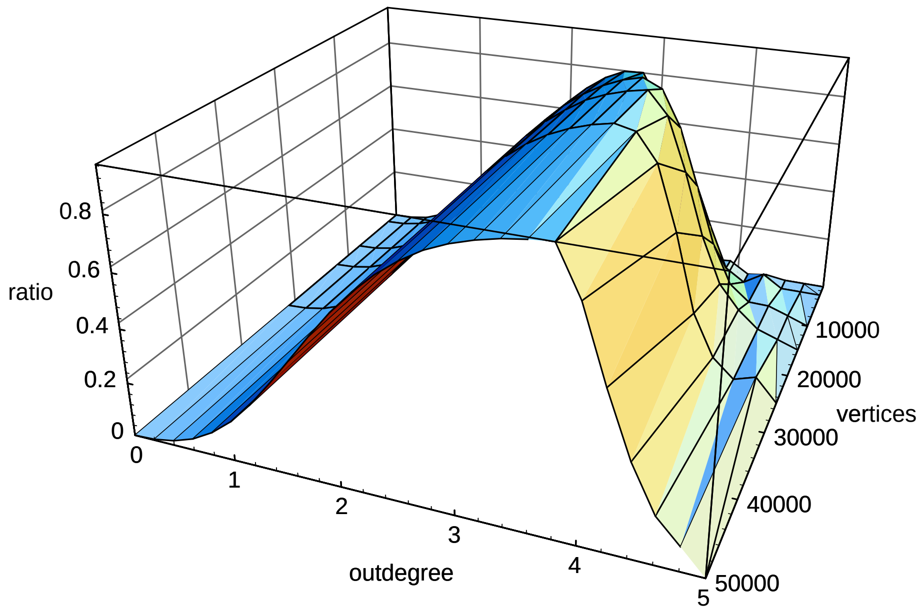


FIGURE 2: Proportion of vertices outside the large component and the trivial components.

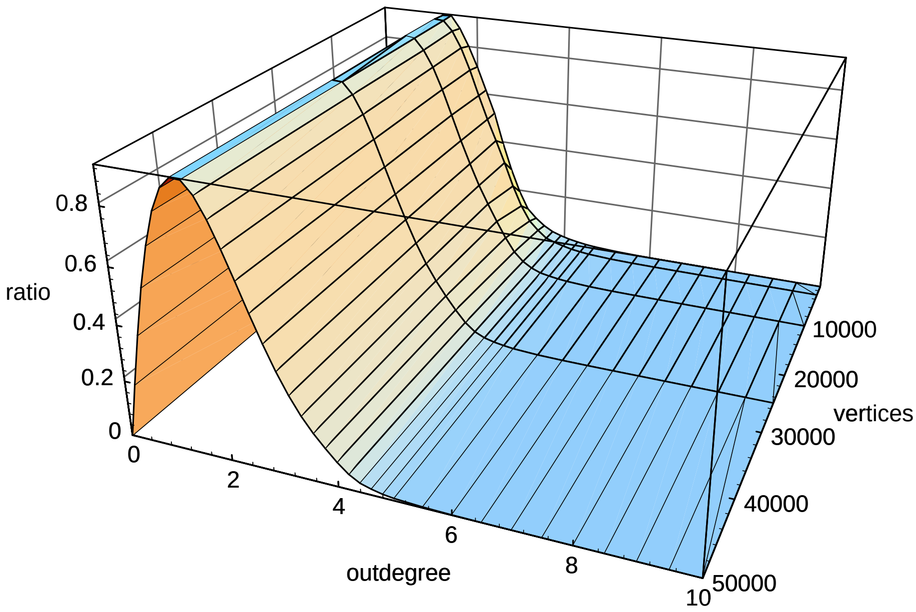


FIGURE 3: Expected number of components stored onto *cstack* divided by the number of vertices, $l = 5$.

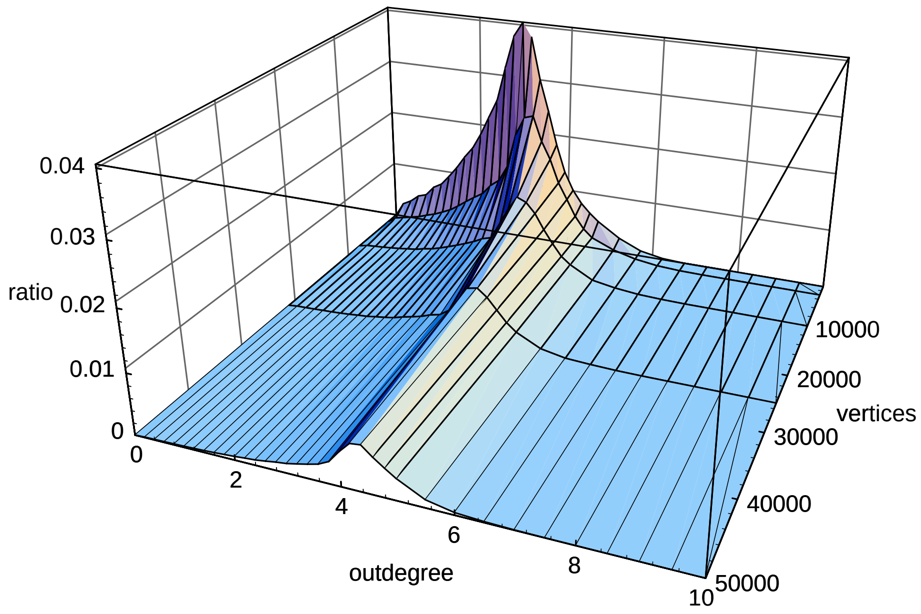
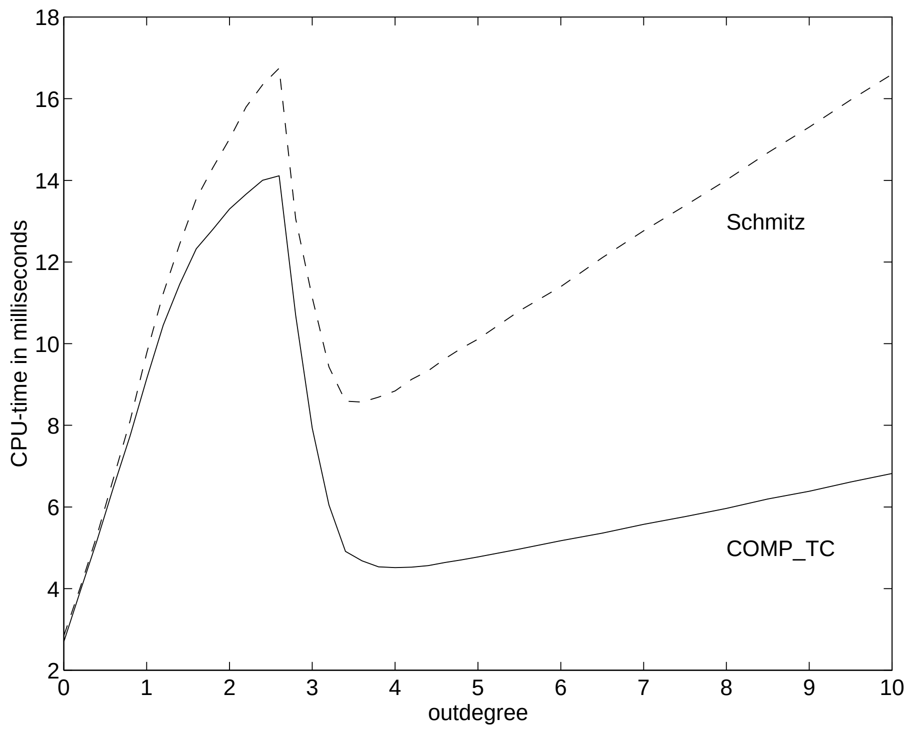
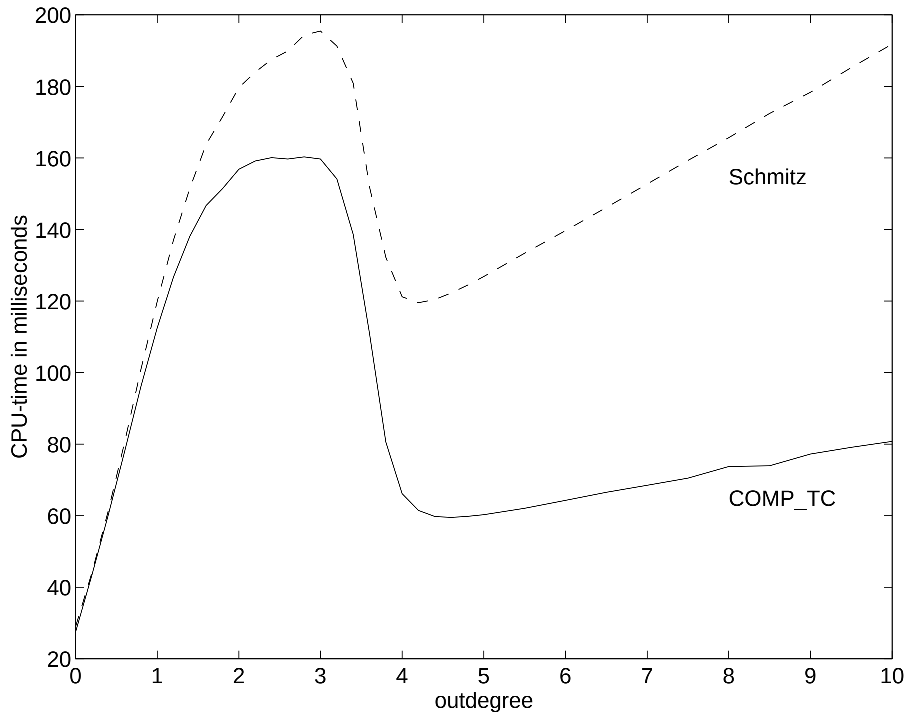


FIGURE 4: Expected maximum height of *cstack* divided by the number of vertices, $l = 5$.



(a) $n = 1000$, $l = 5$



(b) $n = 10000$, $l = 5$

FIGURE 5: CPU times in COMP_TC and in Schmitz's algorithm.