A Single-Pass Algorithm for Transitive Closure

Esko Nuutila
Eljas Soisalon-Soininen

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

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

Abstract

One of the most attractive methods for computing the transitive closure of a binary relation is to apply depth-first search to the corresponding directed graph, recognize the strongly connected components, and, at the same time, compute the successor lists for all nodes. Recognizing the strongly connected components is important because the straightforward successor list computation computes correctly only the successors of the component roots, and these successor lists have to be distributed to all members of the strong components. This scheme has some obvious inefficiences: e.g., if the graph consists of a single cycle of length n, then during the search n sets of size $O(n)$ will be computed, and, finally, the set containing all nodes will be distributed to all nodes in the cycle. A much better way would be to compute directly only the successor list of the root (the node where the search for the cycle starts). For other nodes in the cycle, the successor list is obtained directly via the root node value stored in every node. In the present paper we generalize this idea for all directed graphs: whenever the search of a node exits and we know that the node is not a component root, we will directly add the successor set of the node to the *current candidate* of the component root.

TKO-B 95/93
ISBN 951-22-1412-1
ISNN 0785-6601

TKK OFFSET 1993

1 Introduction

We consider $O(ne)$ algorithms for computing the transitive closure of a binary relation. Here n is the number of nodes and e is the number of edges of the directed graph that represents the binary relation in question. Thus we assume that the graph is stored as adjacency lists of nodes. As a result the algorithms return the successor lists of the nodes, representing the transitive closure.

The algorithms can be classified into two categories: they are either single-pass or two-pass meaning that the graph is traversed only once or, respectively, twice. Essential here is how often the edges of the graph are traversed, nodes may be looked at twice even in a single-pass algorithm.

When the input graph does not fit in the main memory the second traversal of edges in a two-pass algorithm may cause much disk i/o that is avoided in single-pass algorithms. However, two-pass algorithms can often use knowledge about the structure of the input graph more efficiently than single-pass algorithms. In single-pass algorithms it is difficult to avoid extraneous set operations in the computation of the successor sets.

All algorithms considered here are based on the strong components of the input graph. The algorithm of Tarjan [8] for recognizing the strong components in a directed graph may easily be augmented for transitive closure computation. Let v be the current node that Tarjan's algorithm is processing. First, initialize the successor set of v to contain the children of v. Second, after processing an edge (v, w) add all the successors of w computed so far to the successors of v. However, in this way we obtain correct successor sets only for the component roots. Thus, whenever we recognize the root of a component, we have to distribute the successor set for all members of the component. This can be done easily because at such a point of the search all members of the component are directly obtained from the auxiliary stack used in the strong component recognition.

The algorithm sketched above is single-pass and $O(ne)$. Unfortunately, it has some obvious inefficiences. For example, consider the graph composed of a single cycle given in Figure 1. The nodes are named by their depth-first numbers. In the case of a cycle of n nodes, node i will have a successor set of $n - i + 1$ nodes. Thus, n sets of $O(n)$ nodes have to be created. It would be better to compute only one set of n nodes, namely the successor set of the root node 1.

Eve and Kurki-Suonio [2] suggested the following improvement upon the straightforward solution. When we are processing an edge (v, w) we first insert w into the successor set of v. After w has been processed we add the successor set of w to the successor set of v if and only if v and w are in different strong components. When the root of the component is recognized, then all members of the component are looked at (they are found in the auxiliary stack) and the union of their successor sets are computed. This union is then the correct successor set of all component members.

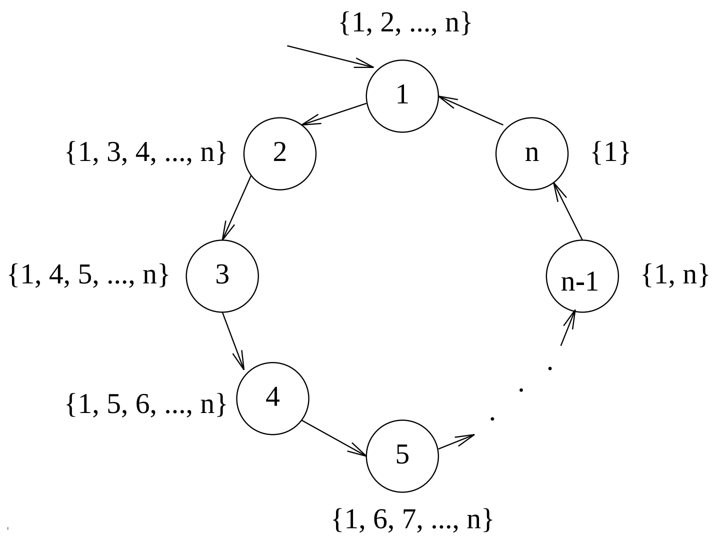

Figure 1: A cycle with partially computed successor sets.

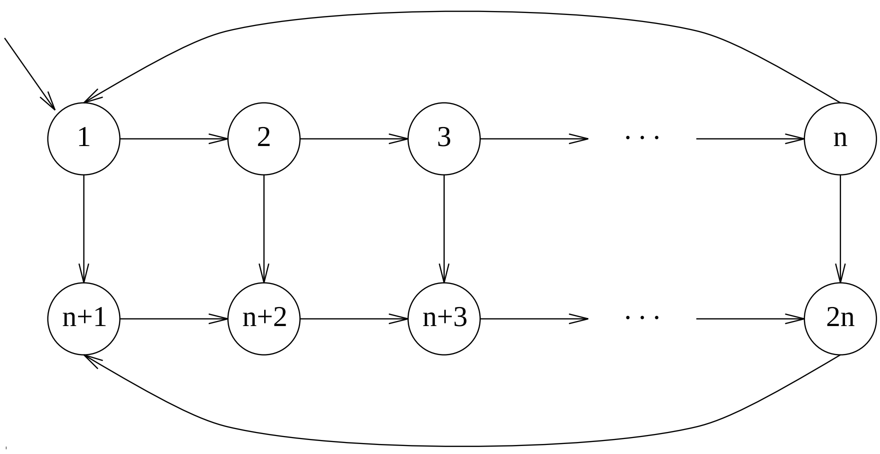

Figure 2: Two cycles with intercomponent edges.

However, even this strategy has some drawbacks. As an example, consider the graph given in Figure 2. There are two strong components $C_1 = \{1, 2, \ldots, n\}$ and $C_2 = \{n+1, n+2, \ldots, 2n\}$. The successor set of nodes $1, 2, \ldots, n$ is $C_1 \cup C_2$ and the successor set of nodes $n+1, n+2, \ldots, 2n$ is C_2. When processing the component C_1, the Eve and Kurki-Suonio algorithm will create n different successor sets of $n+1$ nodes. When the root of component C_1 is recognized, then all these n sets of $n+1$ elements will be unioned. Also in this example it would be better to directly compute the successor set of the root of C_1. Note that if the n successor sets cannot all be stored in the main memory, then it may be costly to collect all necessary sets to be unioned for computing the final successor set of the root.

In this paper we present a new single-pass algorithm that avoids the inefficiencies described above and many other inefficiencies typical to other single-pass algorithms (without introducing the second traversal of the edges). The algorithm is based on the following novel idea: when the search of a node exits and we know that the node is not a component root, we directly add the successor set of the node to the *current candidate* of the component root. According our experiments, this current candidate root node is very often the real component root.

2 Terminology and definitions

Let $G = (V, E)$ be a directed graph corresponding a binary relation R. V is the set of nodes and E is the set of edges in G. We assume that the graph is stored in the adjacency list form, i.e., for every node v there is a set of *children nodes* $E_v = \{w | (v, w) \in E\}$. A *path* from node v_0 to node v_k is an alternating sequence $(v_0, (v_0, v_1), v_1, (v_1, v_2), \ldots, v_{k-1}, (v_{k-1}, v_k), v_k)$ of nodes and edges. It is *non-null*, if it contains at least one edge. The *transitive closure* of G is a graph $G^+ = (V, E^+)$ such that for any $v, w \in V$ 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 children of node v in G^+ are called the *successors* of v in G. Two nodes v and w are *path equivalent* if there is a path from v to w and a path from w to v. Path equivalence partitions the nodes set V into maximal disjoint sets of path equivalent nodes. These sets are called the *strong components* of the graph. A strong component that consists of a single node is called a *trivial component*. Note that all nodes in a strong component have the same successor sets. A *condensation graph* $G^C = (V^C, E^C)$ induced by the strong components of G is a graph whose nodes are the strong components of G. Let $x, y \in V$ and let C_x and C_y be the strong components containing x and y, respectively. Then $(C_x, C_y) \in E^C$ if and only if $(x, y) \in E$.

The algorithms described in this paper are based on depth-first traversal. We use the following terminology. When the traversal visits a node the node is given a unique *depth-first number*. Depth-first numbers form an increasing sequence with the first node visited having the lowest number and the last visited number the highest. Depth-first traversal induces a *depth-first spanning forest* $F = (V, E^s)$, $E^s \subseteq E$, on the graph G based on the order in which the nodes are visited. The edges of graph

G are divided into four categories based on the order of visiting. An edge (v, w) is a *tree edge* if the traversal directly enters w during the processing of node v. Only the tree edges belong to the spanning forest. If an edge (v, w) is in graph G but not in spanning forest F we have the following possibilities: First, if w is a descendant of v in the spanning forest then (v, w) is a *forward edge*. Second, if v is a descendant of w in the spanning forest then (v, w) is a *back edge*. Third, if neither v is a descendant of w nor w is a descendant of v in the spanning forest then (v, w) is a *cross edge*. When depth-first traversal is used for detecting the strong components, the node of the component that is first visited is called the *component root* node.

3 Transitive closure computation based on strong components

In Figure 3 we present SIMPLE_TC, a straightforward adaptation of Tarjan's algorithm [8] to transitive closure computation. This is a single-pass algorithm. Lines `(6)`, `(14)`, and `(20)` are added to Tarjan's algorithm. A variable *tc* is defined for each node. tc[v] contains (the partially computed) successor set of node v. At line `(6)` the children of the current node v are added into the successor set of v. At line `(14)` the successor set of child w computed so far is added into the successor set of 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 `(20)` the successor set of the component root is distributed into the other members of the component.

This algorithm has some obvious inefficiencies. It does many unnecessary set insertions and unions as was explained in Section 1. The following algorithms presented in the literature try to avoid these unnecessary operations[1].

In the algorithm of Eve and Kurki-Suonio [2] the successor set of a child node w of node v is added to the successor set of v only if w belongs to a different strong component. When popping the nodes of a component off the stack, the algorithm computes the union of the successor sets of these nodes and adds it to the successor set of the root node. In general, this algorithm has several sources of inefficiency: First, when processing an edge (v, w) the algorithm always inserts w to the successor set of v. This may cause many unnecessary insertions. Second, the algorithm propagates successor sets along forward edges which is unnecessary since they have already been propagated along tree edges. Third, the algorithm computes the successor set of a non-root node and later adds it to the successor set of the component root. It would be better to compute directly only the successor set of the root. Fourth, all the successor sets of the nodes of a component have to be stored until the component is fully detected. If the sets do not fit in the main memory this may cause many unnecessary disk operations.

[1] Originally most of these algorithms used vectors to represent the result of the computation. Here we describe them in terms of successor sets.

```
procedure simple_tc(node v);
begin
  mark v visited and give it a depth-first number;
  root[v] := v;
  push v onto the stack;
  tc[v] := {w | (v,w) in E};   /* The children of v */
  for each edge (v,w) in E do begin
    if w is not visited then begin
      simple_tc(w);
      root[v] := min{root[v],root[w]}
    end else if v > w then
      if w is not an element of a found component then
        root[v] := min{root[v],root[w]};
    tc[v] := tc[v] U tc[w];
  end;
  if root[v] = v then
    repeat
      w := pop stack;
      mark w as an element of a found component;
      tc[w]:=tc[v]
    until w = v
  end
end;
begin /* Main Program */
  unmark all nodes in V;
  initialize stack to empty;
  for each node v in V do
    if v is not visited then
      simple_tc(v)
end.
```

Figure 3: A simple one-pass transitive closure algorithm based on Tarjan's algorithm [8].

Ebert's algorithm [1] is a slight modification of SIMPLE_TC. When processing an edge (v, w) Ebert's algorithm adds the successor set of w to the successor set of v only if edge (v, w) is either a tree edge or an intercomponent cross edge. The processing of a fully detected strong component is similar to SIMPLE_TC. The problems are also similar: the algorithm propagates the successor sets in a step-by-step fashion along the tree edges until the component root is reached, thus making many unnecessary set operations.

Many of the unnecessary set operations may be avoided if successor sets are computed only after the strong components have been detected. In this way only the successor sets of the component roots have to be computed. This results in a two-pass algorithm. Two different strategies have been used. Schmitz [7] presents algorithm V1 in which the passes are interleaved. The first pass scans the edge list of node v as in SIMPLE_TC (and in the original Tarjan's algorithm), but no successor set computation is done. The second pass takes place when a strong component is fully detected. The edge lists of the nodes of the component are then scanned again and the successor set of the component root is computed. The original algorithm computes only the transitive closure of the condensation graph induced by the strong components of the input graph, but it can easily be adapted to compute the transitive closure of the whole input graph. Schmitz [7] presents also another algorithm V2 that tries to avoid some unnecessary set operations that V1 does. V2 is based on the *edge basis* of the condensation graph of the input graph. This algorithm benefits more from the use of an adjacency matrix to represent the result and it may be difficult to adapt it to use node sets.

Ioannidis, Ramakrishnan and Winger [3, 4] present another two-pass algorithm called BTC. Here the first pass is a slightly modified version of Tarjan's algorithm. In addition to computing the strong components, it computes a reverse topological order (ignoring back edges) of the nodes of the graph. In the second pass the nodes are processed in this order and the successor sets of the component roots are computed. When all nodes of a component have been processed this way the resulting successor set is distributed to all non-root component members.

Ioannidis, Ramakrishnan, and Winger present also a single-pass algorithm called GDFTC. This algorithm tries to generate the successor set of only one node in a strong component. To achieve this, a complicated stack mechanism is used. Instead of nodes, the stack consists of stack frames that contain the partially computed strong components and successor sets. The complicated stack mechanism makes this algorithm run slower than Schmitz' algorithm and BTC [4].

The benefits of two-pass algorithms may be outweighted by the costs of scanning edge lists twice. If the whole graph does not fit into the main memory, which is usually the case when we are processing a database, the whole graph has to be paged into the main memory twice.

4 The New Algorithm

In this section we present a new single-pass algorithm that avoids the problems typical to other single-pass algorithms without introducing the costs of scanning the edge lists twice. The algorithm, called NEWTC, is presented in Figure 4.

The detection of the strong components is done in the same way as in the original Tarjan's algorithm and in SIMPLE_TC. The computation of the successor sets in NEWTC is based on the following ideas:

1. Assume v is the current node and we are processing edge (v, w). If (v, w) is an intercomponent tree edge or an intercomponent cross edge we add root$[w]$ into an auxiliary set adjacent_component_roots$[v]$ (lines `(12)` and `(17)`). Edges inside a component and intercomponent forward edges are ignored.

2. When all edges leaving v have been processed we add every node r in adjacent_component_roots$[v]$, together with tc$[r]$, into tc[root$[v]$], the successor set of the final candidate root node of v (lines `(19)-(21)`). Note that root$[v]$ may be a false root node, but, as we shall explain later, it usually *is* the real component root.

3. Before we exit a non-root node v we add v and tc$[v]$ into tc[root$[v]$] (line `(36)`). In this case tc$[v]$ is non-empty only if v is the (false) final candidate root node of some another node.

4. Before we exit a root node v of a non-trivial component we add v into its own successor set tc$[v]$ (line `24)`) and set the root value of the other component members to v (line `(30)`). If the successor set tc$[w]$ of a non-root component member w is not empty it is added into tc$[v]$ (lines `(28)` and `(29)`). Note that tc$[w]$ is usually empty when the component is fully detected because tc$[w]$ has been cleared at line `(37)`. tc$[w]$ may be non-empty only if all the following conditions hold: (i) w is a non-root node, (ii) there is another non-root node x and an edge (x, w), (iii) the search has exited from w when edge (x, w) is processed, and (iv) root$[w]$ is a false candidate root.

5. The successor set of the root node is not distributed to the other component members. After the transitive closure computation the successor set of any node x is found in tc[root$[x]$].

We assume here that adjacent_component_roots$[v]$ is stored into a local data structure and that these structures usually fit into the main memory during the detection of a component. Thus, the scanning of adjacent_component_roots$[v]$ does not violate the requirement of not traversing the graph more than once.

```
procedure newtc(node v);
begin
  mark v visited and give it a depth-first number;
  root[v] := v;
  push v onto the stack;
  tc[v] := {}; adjacent_component_roots[v]:= {};
  for each edge (v,w) in E do begin
    if w is not visited then begin
      newtc(w);
      root[v] := min{root[v],root[w]};
      if w is an element of a found component then
        adjacent_component_roots[v]:=adjacent_component_roots[v] U {root[w]}
    end else if v > w then
      if w is not an element of a found component then
        root[v] := min{root[v],root[w]}
      else
        adjacent_component_roots[v]:=adjacent_component_roots[v] U {root[w]}
  end;
  for each node r in adjacent_component_roots[v] do
    if r not in tc[root[v]] then
      tc[root[v]] := tc[root[v]] U {r} U tc[r];
  if root[v] = v then begin
    if top(stack) > v then begin
      tc[v] := tc[v] U {v};  /* Insert root to tc[root] */
      repeat
        w := pop stack;
        mark w as an element of a found component;
        if tc[w] not empty then  /* Usually tc[w] = {} */
          tc[v]:=tc[v] U tc[w];
        root[w]:=v;
      until top(stack) = v
    end;
    pop stack;
    mark v as an element of a found component;
  end else begin
    tc[root[v]] := tc[root[v]] U {v} U tc[v];
    tc[v] := {}
  end
end;
begin /* Main Program */
  unmark all nodes in V;
  initialize stack to empty;
  for each node v in V do
    if v is not visited then
      newtc(v)
end.
```

Figure 4: A single-pass transitive closure algorithm that inserts the successor sets to the candidate root node.

4.1 Examples

Consider the graph in Figure 1. First NEWTC enters node 1, then recursively node 2, then recursively node 3, and so on. No nodes are inserted into adjacent_component_roots. Finally NEWTC recursively enters node n. The only edge leaving n leads back to node 1 which is already visited: a loop is detected and root[n] is set to 1 (at line (15)). Because n is not a root node NEWTC adds n and its successor set (which is empty) to the successor set of node 1 (at line (36)). The traversal returns to node $n-1$ and root[$n-1$] is set to 1 (at line (10)). Similarly, $n-1$ is added to the successor set of node 1. This continues until all nodes $n, n-1, \ldots, 2$ have been added to the successor set of node 1. When the traversal returns to node 1 we detect (at line (22)) that 1 is the component root. The component is not trivial. Thus, the component root node 1 is added to its own successor set (at line (24)). The nodes of the component are popped off the stack and marked as elements of a found component. The successor sets of the non-root nodes are empty. Thus, the union (at line (29)) is omitted. As we see, NEWTC does exactly n set insertions and no unions when processing this graph and the run time is $O(n)$. The adjacent_component_roots sets are not used. SIMPLE_TC and Ebert's algorithm would take $O(n^2)$ time when processing this graph.

Consider now the graph in Figure 2. The detection of the component $C_2 = \{n+1, n+2, \ldots, 2n\}$ is similar to the previous example. Here NEWTC detects C_2 when processing the edges leaving some of the nodes $1, 2, \ldots, n$. When exactly this happens depends on the order in which the edges are processed. In any case, NEWTC adds the root of component C_2 into all the sets adjacent_component_roots[i], $1 \leq i \leq n$. When processing set adjacent_component_roots[n], NEWTC inserts the nodes of component C_2 into tc[1]. When NEWTC processes the adjacent_component_roots sets of the other nodes of C_1, tc[1] already contains the root of C_2 and no set operations need to be done. NEWTC does n set insertions when detecting component C_2 and no unions. When detecting component C_1 NEWTC does n set insertions and one union between the set of n elements and an empty set. Thus, the run time is $O(n)$. The Eve and Kurki-Suonio algorithm does $3n$ set insertions and $n-1$ unions between sets of $O(n)$ elements. Thus, the run time is $O(n^2)$. In addition, the Eve and Kurki-Suonio algorithm requires $O(n^2)$ memory space to store the successor sets during the detection of component C_1 whereas NEWTC requires only $O(n)$ space.

5 Variations and efficiency

We have designed some interesting variations of NEWTC. In the first variation we try to minimize the use of the pushdown stack. Tarjan's algorithm pushes nodes onto the stack so that when the whole component is identified the nodes of the component can be marked as members of that component. In NEWTC the nodes in the pushdown stack are also used when computing the successor set of the root node. Note that if NEWTC has not fully detected a component C then, for every node $x \in C$, tc[x] is non-empty only if x is a candidate root node. Note also that every node in adjacent_component_roots is a root node. Thus, NEWTC only uses the successor sets of candidate root nodes in the computation of the transitive closure. If the tests at lines (11) and (14) are changed to check if root[w] (instead of w itself) is a member of a found component then also the component detection can be performed using only candidate root nodes. It is therefore sufficient to push only all candidate root nodes onto the stack. This can be accomplished by removing line (5) from NEWTC and inserting after line (36) the following statement:

```
if root[v] not on the stack then push root[v] onto the stack;
```

Because all component nodes are not on the stack some nodes may have false root values after the detection of the component. Thus the successor set of the real root node has to be distributed to the other candidate roots of the component.

In our second variation we have removed the adjacent_component_roots sets. The omission of these sets makes NEWTC a truly single-pass algorithm even in the cases when all the adjacent_component_roots sets of the nodes in a component would not fit into the main memory during the detection of the component. The modified algorithm works in the following way: Let v be the current node. When processing an edge (v, w), we add w and tc[w] into tc[roots[v]] if (v, w) is an intercomponent tree edge or an intercomponent cross edge. Note that root[v] may change during the processing of the children, thus nodes may be added to the successor set of many different candidate root nodes and also to the successor set of v. If a node v is not a root then tc[v] is added to tc[root[v]] at line (36) of NEWTC. Therefore tc[v] is not needed later and the stack usage optimization described above may be used together with this variation.

So far we have not done any formal analysis or experiments on the behavior of the algorithm. However, we have examined the number of false candidate roots in directed random graphs. Figure 5 presents the maximum number of false candidate root nodes divided by the number of nodes in the input graph as a function of the number n of nodes and the expected number np of edges leaving a node. As we can see, there is usually only a few false candidate root nodes. Random graphs with $3 < np < 4$ seem to have most false candidate root nodes, but even then the number of false candidate root nodes is below 13% of all nodes (below 9% when $n \geq 1000$). When np grows the number of false candidate roots decreases rapidly towards zero. Thus, it seems that at least in directed random graphs, the successors of non-root nodes will usually be added to the successor set of the real component root. The step-by-step propagation typical to SIMPLE_TC and to Ebert's algorithm is mostly avoided. Thus, NEWTC outperforms Ebert's algorithm.

NEWTC is also more efficient than the algorithm by Eve and Kurki-Suonio, because NEWTC does at most as many set operations as their algorithm and often strictly less. The Eve and Kurki-Suonio also requires more successor sets to be stored during the detection of a component than NEWTC.

Obviously NEWTC outperforms Schmitz' algorithm too. There are two reasons for this: First, because NEWTC usually adds the successors of non-root nodes to the successor set of the real component root it does approximately the same amount of set operations as Schmitz' algorithm. Second, Schmitz' algorithm must scan the edges of the graph twice whereas NEWTC scans them only once. For the same reasons NEWTC is also more efficient than the algorithm BTC by Ioannidis, Ramakrishnan, and Winger.

Ioannidis, Ramakrishnan, and Winger seem to have had similar goals as we when they developed algorithm GDFTC. However, NEWTC achieves these goals without introducing the complicated data structures that make GDFTC less efficient than BTC and Schmitz' algorithm [4].

To get a more detailed understanding we are going to do extensive performance comparisons between the different versions of NEWTC and other efficient transitive closure algorithms. In addition to the single and two-pass algorithms described here, we will study other algorithms described in the literature, for example the reachability tree algorithm by Jakobson [5, 6].

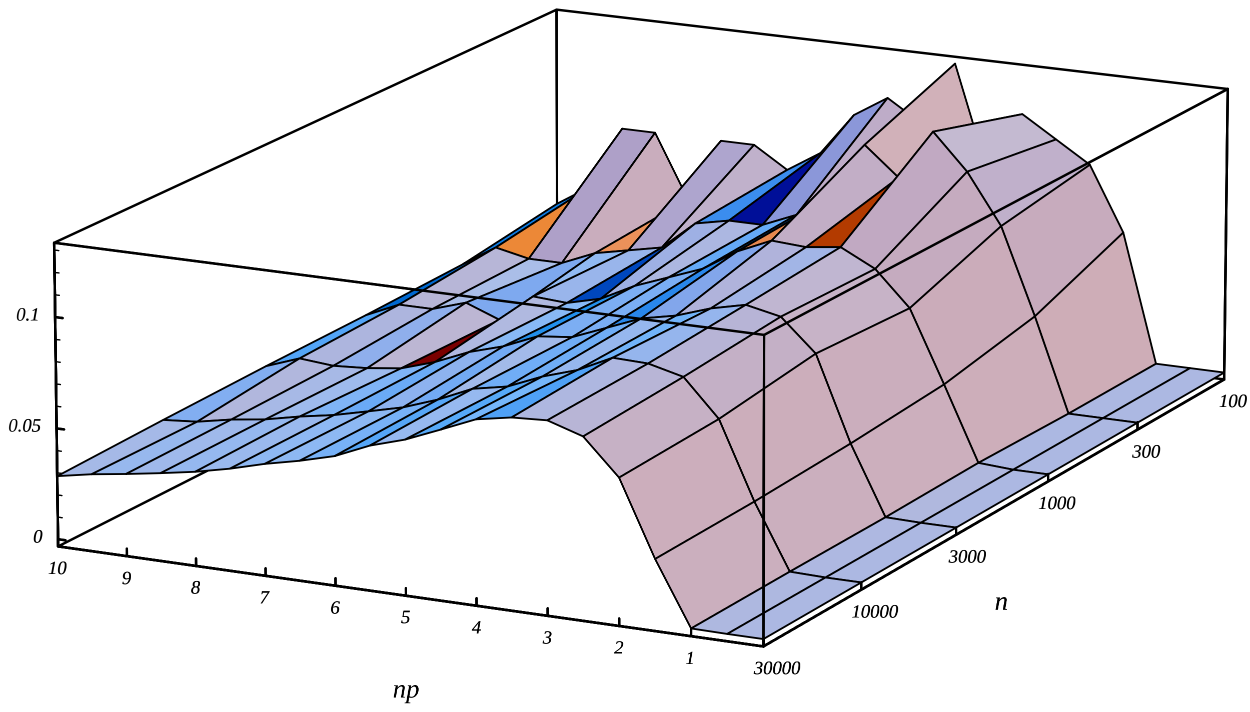

Figure 5: The maximum number of false candidate root nodes divided by the number of nodes.

References

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

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

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

[4] Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. Technical report, Computer Science Department, University of Wisconsin, Madison, WI, 1991.

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

[6] H. Jakobson. On tree-based techniques for query evaluation. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 380–392, San Diego, California, 1992.

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

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