Discrete Applied Mathematics 110 (2001) 77–84

DISCRETE APPLIED MATHEMATICS

# Transitive closure algorithm MEMTC and its performance analysis

Vesa Hirvisalo*, Esko Nuutila, Eljas Soisalon-Soininen

*Helsinki University of Technology, Laboratory of Information Processing Science, P.O. Box 1100, FIN-02015 HUT, Finland*

**Abstract**

In this paper, we present a new algorithm for computing the full transitive closure designed for operation in layered memories. The algorithm is based on strongly connected component detection and on a very compact representation of data. We analyze the average-case performance of the algorithm experimentally in an environment where two layers of memory of different speed are used. In our analysis, we use trace-based simulation of memory operations. © 2001 Elsevier Science B.V. All rights reserved.

## 1. Introduction

This paper presents a new transitive closure algorithm and an experimental study of its performance for large inputs, for which the memory behavior of the algorithm is essential. We compare our new algorithm MEMTC with the algorithm BTC, which was ranked the best algorithm for large inputs by Ioannidis et al. [8].

Transitive closure computation is a basic computational task. 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.

Processing large amounts of data is characteristic to transitive-closure algorithms. If the memory used is not homogeneous, but consists of layers of memory of different speed, then the data transfer between the memory layers is typically the bottleneck of the computation. Algorithm MEMTC tries to reduce memory operations by reading the input in a single pass, using a very compact representation for the resulting closure, and preferring local memory references. Further, it recognizes the strongly connected

* Correspondence address: Department of Computer Science, Helsinki University of Technology, P.O. Box 5400, 02150 Helsinki, Finland.
*E-mail address:* vesa.hirvisalo@cs.hut.fi (V. Hirvisalo).

components of the input and handles them in a topological order to avoid redundant computations.

MEMTC belongs to a class of practical and efficient transitive closure algorithms, which are based on strong component detection. Purdom [16], presented the first of these algorithms. Later better variants have been suggested by Eve and Kurki–Suonio [6], Ebert [5], Schmitz [17], Ioannidis et al. [8], and Nuutila and Soisalon-Soininen [12–15]. In [13], Nuutila showed that algorithm COMPTC [12] has the smallest worst-case execution time of these algorithms.

Several performance evaluations of transitive closure algorithms have been presented in the literature [2–4,8–10]. In most of the studies, the iterative algorithms were less efficient than the matrix-based algorithms, and the matrix-based algorithms were less efficient than the graph-based and the hybrid algorithms. Algorithm BTC seemed to be the best competitor; thus we selected it for our comparison.

BTC [8], suggested by Ioannidis et al., is a graph-based algorithm. It works in two phases: first, it detects the strong components of the input graph by using a modification of Tarjan's algorithm [18], and second, it computes the successor sets representing the transitive closure. The computation of the successor sets is localized in a manner, which makes the algorithm suitable for computations in a paging environment.

To analyze the performance of the algorithms, we ran them to produce memory access traces. Based on the traces, we simulated the memory operations. We performed statistically confirmed experiments to measure the number of memory operations required to construct the full transitive closure of random graphs. We simulated situations where up to 100 Mbytes of memory is required to hold the data. In our experiments, MEMTC outperformed BTC.

The qualitative results show that in both algorithms, a significant proportion of memory overhead, i.e., data transfer between the memory layers, is caused by reading the input. The main reason for the superiority of MEMTC is the compact closure representation that it uses. The algorithm BTC uses its transitive closure representation in a local manner, but the representation is often sparse and occupies a large memory space. Further, BTC uses intensively its large auxiliary data structures.

## 2. Algorithm MEMTC

We consider a directed graph $G=(V,A)$, where $V$ is the set of vertices and $A \subseteq V \times V$ is the set of arcs. The *transitive closure* of $G$ is a graph $G^+ = (V, A^+)$ such that for all $v,w \in V$ there is an arc $(v,w) \in A^+$ if and only if there is a non-null path from $v$ to $w$ in $G$. The *successor set* of a vertex $v$ is the set $Succ(v) = \{w \mid (v,w) \in A^+\}$. A *strong component* of $G$ is a maximal subset $C \subseteq V$ such that for each $v,w \in C$ there is a path from $v$ to $w$ (and vice versa). A *topological order* of the vertex set $V$ of a graph $G = (V,A)$ is any total order $\leqslant$ of $V$ such that $v \leqslant w$ if arc $(v,w) \in A$.

The algorithm MEMTC is derived from the algorithm COMPTC [12], which was designed for operation in a homogeneous memory. The closure representation and the principle

```
(1)  procedure VISIT(vertex $v$)
(2)    push $v$ onto the work stack;
(3)    $Root(v) := v$;
(4)    fetch all arcs leaving $v$ to buffer (push them onto the control stack);
(5)    for each arc $(v, w)$ on the control stack do begin
(6)      if $w$ is not visited then VISIT($w$);
(7)      if $w$ is not already a member of a component then
(8)        if $Root(w) < Root(v)$ in the depth-first order then
(9)          $Root(v) := Root(w)$
(10)     else if arc $(v, w)$ is not a forward arc then
(11)       push the component containing $w$ onto the work stack;
(12)   end;
(13)   if $Root(v) = v$ then begin
(14)     create a new component $C$;
(15)     for all items on the work stack between the top and vertex $v$ do
(16)       if item is a vertex then move it into component $C$;
(17)     $Succ(C) :=$ if component $C$ is cyclic then $\{C\}$ else $\{\}$;
(18)     sort the components that were above vertex $v$ in the work stack
(19)       into a topological order and remove duplicates;
(20)     for each remaining component $X$ in topological order do begin
(21)       pop $X$ from the work stack;
(22)       if $X \notin Succ(C)$ then $Succ(C) := Succ(C) \cup Succ(X) \cup \{X\}$
(23)     end
(24)   end
(25) end;
(26) for each vertex $v$ of the graph do /* main program */
(27)   if $v$ is not visited then VISIT($v$)
```

Fig. 1. Algorithm MEMTC.

of computing the closure are the same in both algorithms. MEMTC differs from COMPTC in two ways, which make MEMTC more suitable for a heterogeneous memory.

The first difference is that MEMTC stores partially handled vertices into the control stack and uses a single work stack for adjacent components and vertices. COMPTC has separate sets for storing partially handled vertices and uses separate stacks for adjacent components and vertices. This difference does not affect the complexity of the algorithm, but it reduces and localizes the memory references.

The second difference is that MEMTC has a memory management. COMPTC has no memory management; it assumes that all memory references have an equal cost. In MEMTC, slow memory is used to store those parts of the input graph, the transitive closure, and the auxiliary data structures, that do not fit in the fast memory. MEMTC uses the least recently used (LRU) algorithm when caching memory except the memory for the input graph. The following block management algorithm is used for the input graph: the blocks with vertices that appear near the top of the control stack are preferred to be kept in the fast memory.

The algorithm MEMTC is given is Fig. 1. In the following description, we refer to the line numbering given in the figure.

The memory overhead is reduced by storing arcs into the control stack, i.e., the stack, where the depth-first-search path is stored. When we enter a vertex $v$, we fetch all arcs leaving $v$ from the slow memory and store them on top of the control stack (line 4). If such buffering were not used, then the recursion inside the for loop (lines 5–12) could cause several fetches of the same arcs.

MEMTC uses Tarjan's algorithm [18] to detect the strong components of a graph. To construct the successor set of a component $C$, MEMTC uses the components adjacent from $C$. These components are collected during the depth-first traversal and stored onto the work stack (lines 10–11). Thus, the work stack contains both vertices and strong components. This localizes further the memory references and reduces overhead.

When a new strong component $C$ is detected, and the vertices of $C$ are removed from the work stack, the components adjacent from $C$ are on top of the work stack. MEMTC sorts them into a topological order and removes duplicates (lines 18–19). Then MEMTC scans the topologically ordered components, and for a component $X$ checks whether $X$ already is in $Succ(C)$. If it is not, then $X$ and $Succ(X)$ are added into $Succ(C)$. Thus, the use of the topological order further reduces the overhead.

MEMTC uses an *interval representation* for the successor sets [13]. The representation was developed from a method by Agrawal et al. [1] for compressing the transitive closure of an acyclic graph. The interval representation consists of two parts: a method for storing sets of integers compactly and a method for mapping the strong components into integers.

The method for storing sets of integers is the following. Consider a set $S \subseteq \{1, 2, \dots, n\}$. If $S$ contains sequences of consecutive integers $i, i+1, \dots, j$, we represent $S$ by storing only the pair of endpoints $[i, j]$ of each sequence. We call such a pair an *interval.* When new intervals are added to a set, overlapping intervals are merged together. Also, two intervals $[i, j]$ and $[j+1, k]$ are combined into an interval $[i, k]$.

The integers that we store are the reverse topological numbers of the strong components. This numbering is easy to compute, since MEMTC detects the strong components in this order. In [13], we showed that the interval representation of the transitive closure is very compact.

## 3. Experiments

In our experiments we compared MEMTC and BTC. The BTC implementation that we used was based on [8]. Because BTC was designed for environments having main memory (fast RAM) and disk memory, we compared the algorithms in that environment. The BTC implementation included the LRU-based memory management suggested in [8]. We performed simulations to measure the number of memory operations required to construct the full transitive closure. In some selected input points, we saved the simulated memory access traces in a file and analyzed them qualitatively.

To measure the performance of the algorithms, we used trace-based analysis tool DBE [7]. DBE executes algorithms using a large and fast random access memory.

During the execution of an algorithm, DBE produces a trace of the memory accesses done by the algorithm. The trace is given to a memory simulator, which records the memory transfer operations caused by the memory accesses, e.g. page faults. The size of memory traces is typically the major problem in such performance studies. DBE uses a trace compaction method to reduce the size of the traces.

Inputs were random graphs drawn from $G(n, p, l)$ with several values of $n$, $p$, and $l$. Graphs in $G(n, p, l)$ have $n$ vertices. An arc between any two vertices $v$ and $w$ is possible only if $(ord_v - ord_w) \bmod n \leqslant l$, where $ord_i$ is a unique integer in the range $\{0, \ldots, n-1\}$ randomly selected for each vertex $i$. Each of the possible arcs exists with probability $p$. Thus, the expected number of arcs is $(2l+1)np$.

We used the $G(n, p, l)$ model, because it generates graphs, which have strongly connected components of various sizes. The more commonly used $G(n, p)$ model tends to generate graphs consisting of one giant component and a number of very small components [11].

We represented the input graph as a relation table $R$ containing the arcs of the input graph (8 bytes of memory per arc). The relation table was clustered, i.e., all arcs departing from a vertex were successive. In addition, there was an index table $T$ consisting of pointers (4 bytes of memory per pointer). Each pointer $T[i]$ pointed to the first arc $(i, j)$ of the cluster of arcs adjacent from $i$.

In the simulation, both the fast and the slow memory consisted of blocks of 512 bytes. The fast memory acted as a buffer to hold the blocks of the input graph, the resulting successor sets, and all the auxiliary data structures needed by the algorithms. Initially, the input graph resided in the slow memory. The code of the algorithms was not modeled, because the code of both MEMTC and BTC are only a few blocks of memory (i.e., can be assumed to reside in the fast memory).

## 3.1. *Quantitative results*

We performed several experiments using inputs drawn from $G(n, p, l)$ with several values of $n$, $p$, $l$, and several buffer sizes. The count of vertices $n$ was in the range 1000–32 000, locality $l$ in the range 5–30, the expected outdegree $(2l+1)p$ in the range 1–10, and buffer size in the range 50–2000 blocks.

In all experiments the number of memory operations caused by MEMTC was less than the number of operations caused by BTC.

We present here the simulation results for $n \leqslant 32\,000$, $p = 0.14$ and $l = 10$ (Fig. 2). The size of the fast memory buffer was 500 blocks for BTC and 300 (LRU)+200 (stack based management) blocks for MEMTC. We computed a 90% confidence interval of the number of operations for each value of $n$. The relative error was at most 10% or the absolute error at most 100 memory operations. The combined condition was used, since the sizes varied much in different parts of the input space.

The results obtained with other outdegrees, localities, and buffer sizes were similar. MEMTC caused much less memory overhead than BTC.

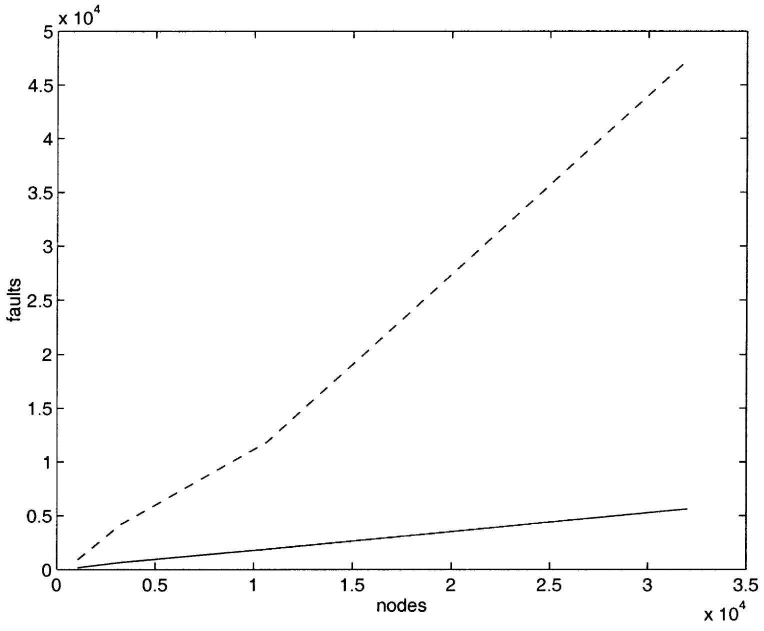

Fig. 2. Number of memory operations caused by MEMTC (solid line) and BTC (dashed line).

## 3.2. *Qualitative results*

We studied the memory behavior of the algorithms using memory access maps. Fig. 3 represents a typical behavior of BTC. The input was a graph with 10 000 vertices and 30132 arcs. The relation table and the index are in memory area 0–281 kbytes. At memory addresses 287–607 kbytes are the auxiliary structures of BTC. The topological order of vertices and the stacks are at addresses 607–728 kbytes. The rest of the memory is used for representing the transitive closure (addresses 769–4337 kbytes).

The two phases of BTC can be clearly seen in Fig. 3. In the first phase, immediate successors of the vertices are moved into the closure representation. The input is scanned heavily and it causes many memory operations. The transitive closure representation is used in a local manner, but it is sparse and occupies a large amount of memory.

In the second phase, the input is no longer used. The closure representation becomes more dense and accesses are more scattered. This causes overhead. The auxiliary data structures are heavily used in both phases. They cause a significant number of memory operations.

In Fig. 4, MEMTC is given the same input as BTC in Fig. 3. Note that the pictures are presented on the same scale. BTC uses much memory and does many memory accesses compared to MEMTC. For larger inputs, this becomes even more apparent.

The input table and index occupy the same memory area as for BTC. The auxiliary data structures are in memory area 462–542 kbytes. At 543–703 kbytes is the work

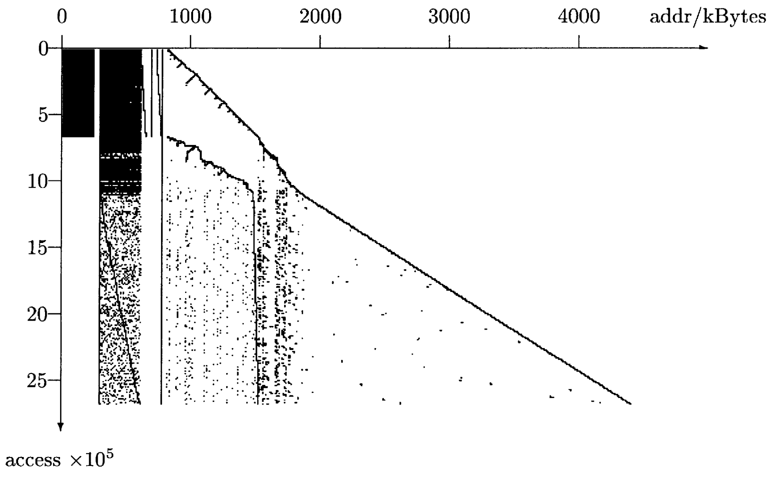

Fig. 3. Memory access map for BTC, input = $G(10\,000, 0.14, 10)$. A pixel at position $(x, y)$ shows that $y$th memory access has referred to the memory location $x$.

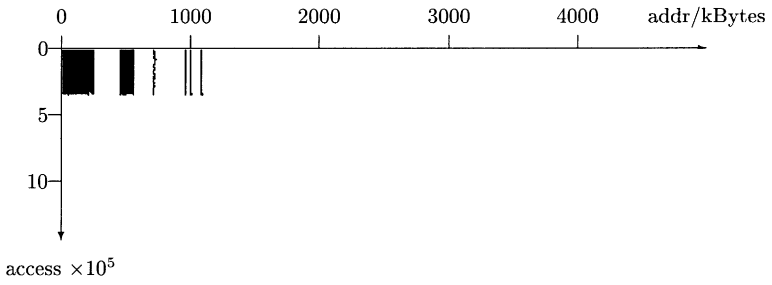

Fig. 4. Memory access map for MEMTC, input = $G(10\,000, 0.14, 10)$. A pixel at position $(x, y)$ shows that $y$th memory access has referred to the location $x$.

stack and at 704–944 kbytes is the control stack. The rest of the memory is used for transitive closure representation (addresses 944–1127 kbytes), i.e., the representation of strong components and the interval representation of successor sets. The almost vertical lines are accesses to these structures.

Most of the memory operations are caused by reading the input. Some operations are caused by accessing the auxiliary data structures. The transitive closure representation is very small and used in a very local manner. Accessing the transitive closure representation causes only a few operations.

## 4. Conclusions

We presented a new transitive closure algorithm, called MEMTC and an experimental study of its performance in an environment having main memory (fast RAM) and disk memory. We compared its performance with BTC, which was reported to be the best algorithm in such an environment [8]. We analyzed and compared the performance of the algorithms for large inputs and used statistically sound methods to state the accuracy of our results.

In our analysis, MEMTC was found superior to BTC. Besides the quantitative analysis, we conducted a qualitative analysis. The qualitative analysis revealed why BTC is slower than MEMTC. The main reason is the representation of the data. The closure representation of BTC is large and it has large auxiliary data structures that it uses very intensively. MEMTC uses a very compact representation for the closure and has only a few auxiliary data structures.

## References

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

[2] R. Agrawal, S. Dar, H.V. Jagadish, Direct transitive closure algorithms: design and performance evaluation, ACM Trans. Database Systems 15 (3) (1990) 427–458.

[3] R. Agrawal, H.V. Jagadish, Hybrid transitive closure algorithms, Proceedings of the 16th International VLDB Conference, Brisbane, Australia, 1990, pp. 326–334.

[4] S. Dar, R. Ramakrishnan, A performance study of transitive closure algorithms, Proceedings of the ACM-SIGMOD 1994 Conference on Management of Data, 1994.

[5] J. Ebert, A sensitive transitive closure algorithm, Inform. Process. Lett. 12 (1981) 255–258.

[6] J. Eve, R. Kurki-Suonio, On computing the transitive closure of a relation, Acta Inform. 8 (1977) 303–314.

[7] V. Hirvisalo, DBE — a tool for trace driven memory simulation, Tool Descriptions of Ninth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'97), St. Malo, France, June 1997.

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

[9] B. Jiang, A suitable algorithm for computing partial transitive closures in databases, Proceedings of the IEEE Sixth International Conference on Data Engineering, Los Angeles, CA, February 1990, pp. 264–271.

[10] R. Kabler, Y.E. Ioannidis, M. Carey, Performance evaluation of algorithms for transitive closure, Inform. Systems 17 (5) (1992) 415–441.

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

[12] E. Nuutila, An efficient transitive closure algorithm for cyclic digraphs, Inform. Process. Lett. 52 (1994) 207–213.

[13] E. Nuutila, Efficient transitive closure computation in large digraphs, Ph.D. Thesis, Helsinki University of Technology, Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, No. 74, 1995.

[14] E. Nuutila, E. Soisalon-Soininen, Efficient transitive closure computation, Technical Report TKO-B113, Helsinki University of Technology, Laboratory of Information Processing Science, 1993.

[15] E. Nuutila E. Soisalon-Soininen, On finding the strongly connected components in a directed graph, Inform. Process. Lett. 49 (1994) 9–14.

[16] P. Purdom, A transitive closure algorithm, BIT 10 (1970) 76–94.

[17] L. Schmitz, An improved transitive closure algorithm, Computing 30 (1983) 359–371.

[18] R.E. Tarjan, Depth first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.