Transitive closure algorithm DISK_TC and its performance analysis

Vesa Hirvisalo
Esko Nuutila
Eljas Soisalon-Soininen

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

Telephone: +358 9 451 3339
Telefax: +358 9 451 3293

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

Abstract

We present a new algorithm for computing the full transitive closure of a binary relation. We analyze its average-case performance experimentally in an environment where secondary memory (disk) must be used to store the data. We compare our new algorithm, called DISK_TC, with the algorithm BTC designed by Ionnadis *et al.* [26] and reported by them to be the most efficient algorithm in disk memory environment. We present both quantitative and qualitative results showing that DISK_TC is faster than BTC. In our analysis, we simulate the disk I/O in the main memory. Simulation takes less time than measuring a real system that does actual disk I/O; thus we can study a large number of inputs. We present also our analysis method and tools, which are applicable to other algorithm performance studies.

TKO-B135
ISBN 951-22-3386-X
ISSN 1239-6893

TKK OFFSET 1996

1 Introduction

This paper presents a new transitive closure algorithm and an experimental study of its performance for large inputs, for which secondary memory (disk) must be used to store the data. We compare our new algorithm DISK_TC, derived from a main-memory algorithm given in [34], with the algorithm BTC, which was ranked the best algorithm for large inputs by Ioannidis *et al.* [26].

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 [8] and in the construction of parsing automata in compiler construction [6]. 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 [10]. Because of the wide applicability, quite a many transitive closure algorithms have been proposed in the literature, which we review in Section 3.

Processing large amounts of data is characteristic to transitive closure algorithms. If the data do not fit in the main memory, then disk memory is used as a secondary memory. Typically, the data transfer between the main memory and the secondary memory is the bottleneck of the computation. Algorithm DISK_TC tries to reduce disk I/O 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 components of the input and handles them in a topological order to avoid redundant computations.

Several performance evaluations of transitive closure algorithms have been presented in the literature. We discuss them in Section 3. Some of these studies use worst-case analysis, which often is overly pessimistic or even misleading. The experimental average-case studies do not assess the accuracy of the results. Further, the studies typically have no proper qualitative analysis, i.e., they do not demonstrate how the costs distribute over the various parts of the computation. Measuring the quantitative performance is not sufficient: the costly parts of the algorithm and its execution should be recognized.

We built an environment, called DBE, for analyzing the performance of memory I/O intensive algorithms. We ran transitive closure algorithms in fast random access memory and simulated the disk I/O. Simulation takes less time than measuring a real system that does actual disk I/O; thus we can study a large number of inputs. DBE has a special purpose compiler, which instruments the algorithm code to produce *memory reference trace* instead of performing true disk I/O. The trace is given to a simulator that simulates disk I/O according to a memory management scheme. The trace is compacted and the simulator is run in parallel with the transitive closure algorithm.

We performed statistically confirmed experiments to measure the amount of disk I/O required to construct the full transitive closure of random graphs when disk is used as secondary memory. We simulated situations where up to 100 megabytes of memory is required to hold the data. In our experiments, DISK_TC outperformed BTC. Further, the quantitative results indicate that disk I/O caused by DISK_TC grows linearly to the number of vertices, but disk I/O caused by BTC seems to grow overlinearly.

The qualitative results show that in both algorithms, a significant proportion of disk I/O is caused by reading the input. The main reason for the superiority of DISK_TC is the compact closure representation that is 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.

The structure of this paper is the following. In Section 2, we discusses the transitive closure problem. Section 3 presents an overview of the pertinent literature. In Section 4, we present our new algorithm. In Section 5, we describe our simulation environment. Section 6 presents the experiments and Section 7 summarizes the results and gives the conclusions.

2 Transitive closure computation

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. If $(v, w) \in E$, vertex v is *adjacent to* vertex w and w is *adjacent from* 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 non-null path from v to w in G. The *successor set* of a vertex v is the set $Succ(v) = \{w \mid (v, w) \in E^+\}$. 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). The *condensation graph induced by the strong components of graph* G is a graph $\overline{G} = (\overline{V}, \overline{E})$ such that $\overline{V}$ is the set of the strong components of G and $\overline{E}$ contains an edge (X, Y) if and only if E contains an edge (u, v) such that u and v are in components X and Y, respectively. A *topological order* of the vertex set V of a graph $G = (V, E)$ is any total order $\leq$ of V such that $v \leq w$ if edge $(v, w) \in E$. If $v \leq w$, we say that v is *topologically smaller* than w and w is *topologically greater* than v. The reverse relation of a topological order $\leq$ is a *reverse topological order*.

Several variants of the transitive closure problem exist. We consider here the full transitive closure problem. In the full transitive closure problem, we should find all pairs of vertices in the input graph that are connected via non-null paths. We assume that the input graph may be too large to fit into the main memory and must reside in the secondary memory.

A major difficulty in transitive closure computation is the avoidance of redundant operations. A directed graph may contain pairs of vertices that are connected via multiple paths. Several vertices may have the same set of successor vertices. The graph in Figure 1 illustrates these redundancies. Two paths lead from vertex a to vertex f and all its successors. Vertices d, e, f, g, h, and i have the same successor set.

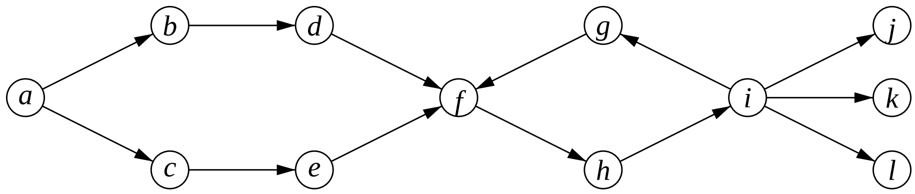

Figure 1: A directed graph with vertices having the same successor sets and with multiple paths between vertices.

The need for secondary memory, such as disk memory, makes the computation of the transitive closure even more complex. In a typical transitive closure algorithm, the data accesses are scattered around the memory. Regardless of the memory management scheme used, this often results in intensive disk I/O where each block can be read and written several times. Therefore, it is important to make the memory references as local as possible.

3 Related work

Warshall's algorithm [46] is probably the best-known transitive closure algorithm. The algorithm is simple, but its worst-case and best-case execution times are $O(n^3)$ and $O(n^2)$, respectively, which make it inferior to some newer algorithms.

Munro [33] and Fischer and Meyer [17] presented transitive closure algorithms that use matrix multiplication and that have the same worst-case execution time as a single matrix multiplication. These are the asymptotically fastest transitive closure algorithms when the input graph is dense, i.e., contains near n^2 edges. The fastest known matrix multiplication algorithm [11] has the worst-case bound $n^{2.376}$, but is not practical.

A class of practical and efficient transitive closure algorithms is based on strong component detection. Purdom [39] presented the first of these algorithms. His algorithm detects the strong components, builds the condensation graph, computes the transitive closure of the condensation graph, and finally computes the transitive closure of the input graph from the transitive closure of the condensation graph. Later, Eve and Kurki-Suonio [16], Ebert [15], Schmitz [41], Ioannidis *et al.* [26], and Nuutila and Soisalon-Soininen [34, 35, 36, 37] presented transitive closure algorithms that use the strong components without generating the condensation graph.

In [35], we showed that algorithm COMP_TC [34] has the smallest worst-case execution time of these algorithms, namely $O(n\overline{e}_r + \min(ns, e' \log n))$, where $\overline{e}_r$ is the number of edges in the transitive reduction of the condensation graph. The term $\min(ns, e' \log n)$ is usually negligible to $n\overline{e}_r$. The algorithm by Goralcikova and Koubek [19] has a worst-case bound $O(n\overline{e}_r)$, but it does several passes over the input graph making the algorithm impractical. Algorithm COMP_TC uses only a single traversal. We also presented a large collection of experiments for comparing the average-case performance of the transitive closure algorithms that are based on strong component detection. Algorithm COMP_TC was more efficient than the other algorithms in all these experiments.

Recently, several transitive closure algorithms have been presented for databases, see [12, 35] for surveys of these algorithms. Some work concentrates on transitive closure computation in parallel or distributed environments, see, for instance, [4, 9, 21, 22].

Several performance analyses and comparisons of transitive closure algorithms have been presented in the literature. The comparisons between the algorithms are mostly based on empirical performance measurements or simulations. We describe some of the studies below.

Ioannidis compared the logarithmic algorithm [23] and the semi-naive algorithm [7] when computing the transitive closure of small lists and trees. The logarithmic algorithm was more efficient than the semi-naive algorithm in most inputs, both for the disk I/O and the CPU-time. Valduriez and Boral [45] also reported results indicating that logarithmic algorithms are more efficient than the semi-naive algorithm.

In a more recent study, Kabler *et al.* [29] compared the semi-naive algorithm, the logarithmic algorithm and the Blocked Warren algorithm, which is one of the Warshall-derived algorithms [3], when computing the transitive closure of randomly generated trees and graphs. The Blocked Warren algorithm was usually superior to the other two algorithms. Contrary to the results of Ioannidis [23] and Valduriez and Boral [45], the semi-naive algorithm was usually superior to the logarithmic algorithm.

Lu [31] compared his improved semi-naive algorithm, the original semi-naive algorithm, and the logarithmic algorithm when computing the transitive closure of small lists and binary trees. The method of study was simulation, not direct measurement as in the other comparisons described above. The semi-naive algorithm augmented with the improvements proposed by Lu was more efficient than the other two algorithms.

Agrawal and Jagadish [3] compared three Warshall-derived algorithms and a logarithmic algorithm when computing the transitive closure of randomly generated graphs. The Blocked Warren algorithm was the most efficient of the Warshall-derived algorithms and it was also superior to the logarithmic algorithm.

Agrawal *et al.* [2] compared two Warshall-derived algorithms and the semi-naive algorithm when computing the transitive closure, the bill of materials, and the shortest paths of randomly generated graphs. The Warshall-derived algorithms were roughly equally efficient, but the semi-naive algorithm was much slower.

Agrawal and Jagadish [5] compared their hybrid algorithm with a Warshall-derived algorithm, a graph-based algorithm presented in [24] and the Grid algorithm by Ullman and Yannakakis [44] in computing the bill of materials of randomly generated acyclic graphs. The hybrid algorithm was in all inputs more efficient than the other algorithms.

Ioannidis *et al.* [25, 26] compared their graph-based algorithms BTC and GDFTC with Schmitz's graph-based algorithm [41] when computing the transitive closure of randomly generated graphs. Algorithm BTC was in most inputs more efficient than the other two algorithms.

Dar and Jagadish [13] compared their spanning tree transitive closure algorithm with the hybrid algorithm by Agrawal and Jagadish [5] when computing the transitive closure of randomly generated acyclic graphs. The spanning tree algorithm was more efficient than the hybrid algorithm in all inputs.

Dar and Ramakrishnan [12, 14] compared several algorithms in computing full and partial transitive closures of randomly generated acyclic graphs. The algorithms were BTC by Ioannidis *et al.* [25, 26], the hybrid algorithm by Agrawal and Jagadish [5], Jiang's algorithm [28], the spanning tree algorithm by Dar and Jagadish [13], and two algorithms by Jakobson [27]. The hybrid algorithm and the spanning tree algorithm were less efficient than the other algorithms. The relative efficiency of the algorithms varied in different inputs and no algorithm was generally superior to the other algorithms.

The performance studies discussed above do not give a clear picture of the efficiency of the algorithms. The most severe problems in these studies were in the selection of the inputs and in the analysis of the measurement data. In some studies, only trees and lists were used as inputs. In all studies, the inputs were small; the studies did not answer the question of what happens if realistic amounts of data are used as inputs. Further, the studies gave no information on the accuracy of the results.

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. BTC seemed to be the best algorithm for disk environment; thus we selected it for our comparison.

4 Algorithm DISK_TC

The algorithm DISK_TC is derived from the algorithm COMP_TC [35], which was designed for main memory operation. The closure representation and the principle of computing the closure are the same in both algorithms, but DISK_TC has a memory management and uses different auxiliary data structures. To reduce and localize memory references it stores partially handled vertices into the control stack. Further, it has a single work stack instead of using separate stacks for adjacent components and vertices as in COMP_TC.

Disk 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 main memory. DISK_TC uses the LRU (Least Recently Used) algorithm when caching disk pages except the pages of the input graph. The following page management algorithm is used for the input graph: the pages with vertices that appear near the top of the control stack are preferred to be kept in the main memory.

Algorithm DISK_TC (Figure 2) is based on Tarjan's strong component detection algorithm [43]. It consists of a recursive procedure VISIT traversing the vertices in a depth-first order and of a main program that applies the procedure to all vertices that have not already been visited. The procedure does two tasks. The first task is to detect the strong components and the second task, after detecting a strong component, is to construct the successor set of the strong component.

The disk I/O is reduced by storing edges into the control stack, i.e., the stack, where the depth-first-search path is stored. When we enter a vertex v, we read all edges leaving v from the disk and store them on top of the control stack (line 4). If the edges leaving v are stored successively on the disk and such input buffering were not used, then the recursion inside the for loop (lines 5–12) could cause reading the edges several times.

The strong components are detected in the following way. For each strong component C, the first entered vertex of C is called the *root* of component C. To

```
(1)  procedure VISIT(vertex v)
(2)      push v onto the work stack;
(3)      Root(v) := v;
(4)      read all edges leaving v and push them onto the control stack;
(5)      for each edge (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 edge (v, w) is not a forward edge 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 ∉ Succ(C) then Succ(C) := Succ(C) ∪ Succ(X) ∪ {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)
```

Figure 2: Algorithm DISK_TC

detect the roots, we define a variable $Root(v)$ for each vertex v. When DISK_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 DISK_TC processes the edges leaving vertex v (at lines 5–12), new root candidates are obtained from the vertices adjacent from v that belong to the same component as v (lines 8–9).

When DISK_TC has processed all edges leaving v, $Root(v) = v$ if and only if v is a component root (line 13). At line 14, DISK_TC creates a new component C. To obtain the vertices of component C, DISK_TC uses a work stack. Each vertex is stored on the work stack in the beginning of DISK_TC. At lines 15–16, DISK_TC removes the vertices of component C from the work stack and inserts them into component C.

To construct the successor set of a component C, DISK_TC 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 disk I/O.

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. DISK_TC sorts them into a topological order and removes duplicates (lines 18–19). Then DISK_TC 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 disk I/O.

DISK_TC uses an *interval representation* for the successor sets [35]. 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 $S \subseteq \{1, 2, \ldots, n\}$. If S contains sequences of consecutive integers $i, i+1, \ldots, j$, 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 DISK_TC detects the strong components in this order. In [35], we showed that the interval representation of the transitive closure is very compact.

5 Analysis environment

We considered using mathematical analysis or measurements of a real implementation as the evaluation method, but simulation turned out to be the best choice. Mathematical average-case analysis using realistic input models is beyond the limits of the current analytical methods [20, 32] for algorithms like DISK_TC. On the other hand, simulation takes orders of magnitude less time than measuring a real system that does actual disk I/O; thus we can study a large number of inputs.

We have implemented an environment, called DBE, for the performance analysis of memory I/O intensive algorithms. DBE executes algorithms using a large and

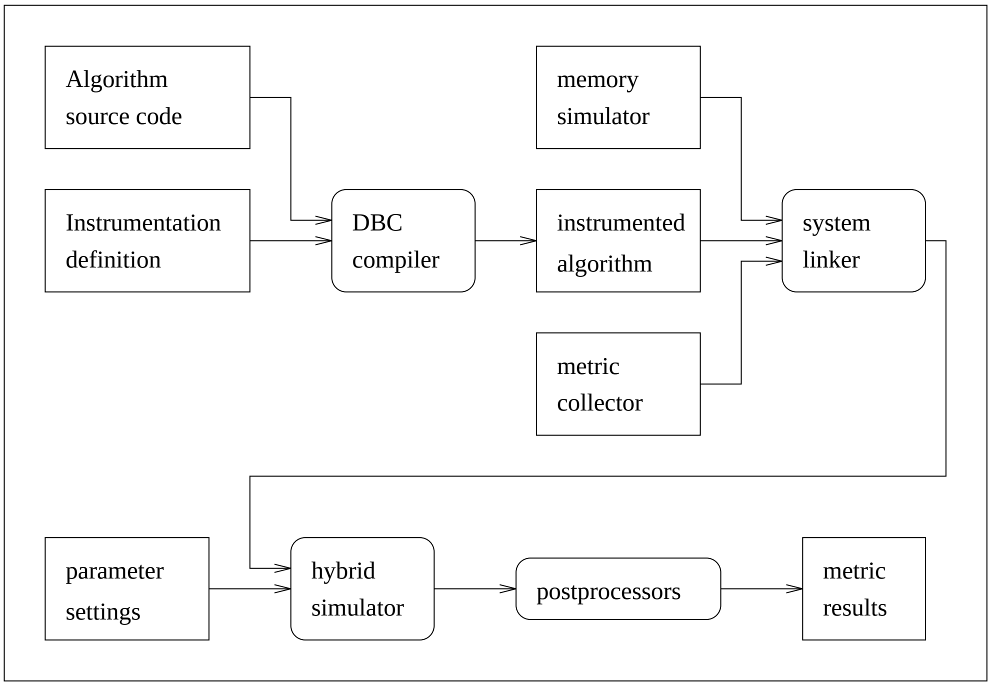

Figure 3: The analysis environment DBE.

fast random access memory. During the execution, it simulates the disk I/O that would be needed using a smaller main memory and a swap disk.

DBE uses a special purpose compiler that instruments the algorithm code. The compiled code does not perform physical disk I/O operations. It only generates a trace of memory accesses, which is given to a memory simulator. The memory simulator is a part of the runtime system of the compiler. The compiled algorithm is linked with the runtime system to form a hybrid simulation program. The program is a hybrid simulator in the sense that only the disk I/O is simulated. The algorithm itself is not simulated; it is properly executed.

The use of a compiler instead of hand-coded instrumentations makes its possible to reliably instrument large programs. Further, experimenting effects of modifications to an algorithm is easy, because the instrumentations are automatically kept up to date.

Besides the memory simulator, the runtime system includes a set of input generators and a metric data collector, which collects the data produced by the simulator and the compiled code. The collector performs a statistical analysis to get values for the metrics, and outputs all metric values into a metric file when the simulation terminates.

The simulations are run by a simulation driver, which uses a *sequential procedure* [38] to yield statistically confirmed measurements. A driver control file defines a set of areas in the input space and the required accuracy of the results. For each defined area of the input space, the simulation driver repetitively generates random inputs and runs the simulation program until the required accuracy is reached. The approach is similar to the one presented in [18].

5.1 DBC compiler and simulation

We must define *a model of computation* to be able to measure the performance of an algorithm by simulation. The performance is expressed in terms of the model. In DBE, a model of computation is described as an abstract machine on which the algorithm runs. The abstract machine used in our experiments consists of a CPU, a main memory, a memory manager and a mass memory. The main memory is a byte addressed random access memory. Each memory cell is accessible in the same constant time. The mass memory consists of blocks. The size of the mass memory is infinite. All the computation is done in a virtual memory. The abstract machine is implemented by the compiler and the memory simulator in the runtime system.

Programs running on DBE are written in a simple language called DBC, which is a subset of ANSI C language. DBC excludes some unorthogonal features of C, e.g., the switch statement and the increment operator (`++`).

The DBC compiler compiles a source program in a single pass into a sequence of three address statements and inserts instrumentation statements according to the definitions in a definition file. The DBC compiler uses C++ as an intermediate code and the Gnu `g++` compiler as its back end.

The memory access trace resulting from a single run is too large to be stored on disk and processed later by the simulator. Therefore, the memory access trace is compacted, and the simulator is run in parallel with the transitive closure algorithm. The trace compaction is done both during the compile time and during the run time. The trace compaction is based on a method that is similar to optimization used in compilers [6], replacing long sequences with shorter ones having the same effect [40, 42].

5.2 Postprocessing

The simulation of an algorithm typically yields a large amount of metric data. For example, the algorithm may be run on several thousands of inputs. Such an amount of data cannot be analyzed without automated postprocessing tools.

We have used three postprocessors in our experiments: COMBINER, PICTURE and SUMMARY. COMBINER reads the metric data files of several experiments and collects the values of a selected metric. Output of COMBINER can given to common tools, for example, to produce graphical presentation (see Figure 4 in the next section).

PICTURE reads the metric data from a single experiment and produces a two dimensional graphical map, where each position represents a memory access. The y-coordinate of a map is the ordinal number of a memory access; the x-coordinate is the memory address accessed. Thus, a memory event map shows all points (x, y) such that the yth memory access refers to the memory location x. Because of scaling, each pixel of a map usually represents several memory accesses. The maps in the next section (Figures 5 and 6) are created with PICTURE.

SUMMARY reads the memory event data produced by a single experiment. It finds out how many memory events occurred in a memory area during a time interval. For example, it can calculate how many page faults occurred in a specific memory area during a specific phase of the execution of an algorithm.

6 Experiments

In our experiments we compared DISK_TC and BTC. We performed simulations to measure the amount of disk I/O required to construct the full transitive closure, when the main memory is not sufficient for the computation. In some selected input points, we saved the simulated memory access traces in a file and analyzed them qualitatively.

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 edge between any two vertices v and w is possible only if $(ord_v - ord_w) \bmod n \leq l$, where ord_i is a unique integer in the range $\{0 \ldots n-1\}$ randomly selected for each vertex i. Each of the possible edges exists with probability p. Thus, the expected number of edges 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 [30].

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

The simulation model had a paged main memory of finite size and a paged disk memory of infinite size. The size of a page was 512 bytes. The main memory acted as a buffer to hold pages of the input graph, the resulting successor sets, and all

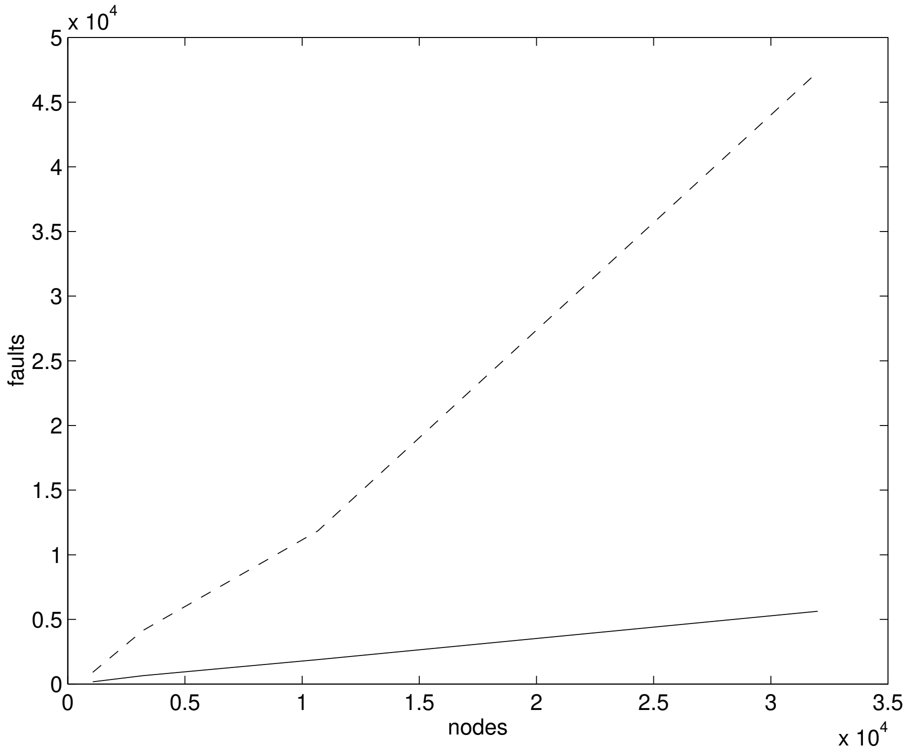

Figure 4: Page faults by DISK_TC (solid line) and BTC (dashed line).

the auxiliary data structures needed by the algorithms. The algorithm code itself was not modeled, because the code of both DISK_TC and BTC are only a few pages of memory.

6.1 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 range 1000–32000, locality l in range 5–30, the expected outdegree $(2l + 1)p$ in range 1–10, and buffer size in range 50–2000 pages.

In all experiments the disk I/O caused by DISK_TC was less than to the disk I/O caused by BTC.

We present here the simulation results for $n \leq 32000$, $p = 0.14$ and $l = 10$ (Figure 4). The size of the main memory buffer was 500 pages for BTC and 300 (LRU) + 200 (stack based management) pages for DISK_TC. We computed a 90% confidence interval of the page fault count for each value of n. The relative error was at most 10% or the absolute error at most 100 page faults. The combined condition was used, since the sizes varied much in different parts of the input space.

DISK_TC caused much less disk I/O than BTC. Further, as can be seen from the figure, disk I/O caused by DISK_TC tends to grow linearly (with respect to the number of the vertices). Disk I/O caused by BTC seems to grow overlinearly. The results obtained with other outdegrees, localities, and buffer sizes were similar.

6.2 Qualitative results

We studied the memory behavior of the algorithms using memory access maps (described in Section 5.2). Figure 5 represents a typical behavior of BTC. The input was a graph with 10000 vertices and 30132 edges. The relation table and the index are in memory area 0–281kBytes. At memory addresses 287–607kBytes are the auxiliary structures of BTC. The topological order of vertices and the stacks are at addresses 607–728kBytes. The rest of the memory is used for representing the transitive closure (addresses 769–4337kBytes).

The two phases of BTC can be clearly seen in Figure 5. In the first phase, immediate successors of the vertices are moved into the closure representation. The input is scanned heavily and it causes many page faults. 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

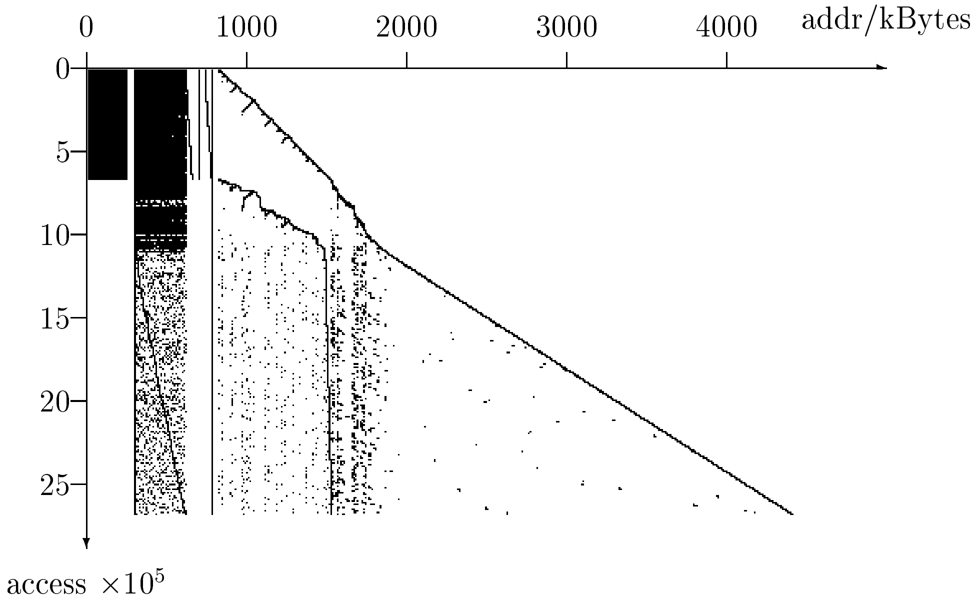

Figure 5: Memory access map for BTC, input = G(10000, 0.14, 10). A pixel at position (x, y) shows that yth memory access has referred to the memory location x.

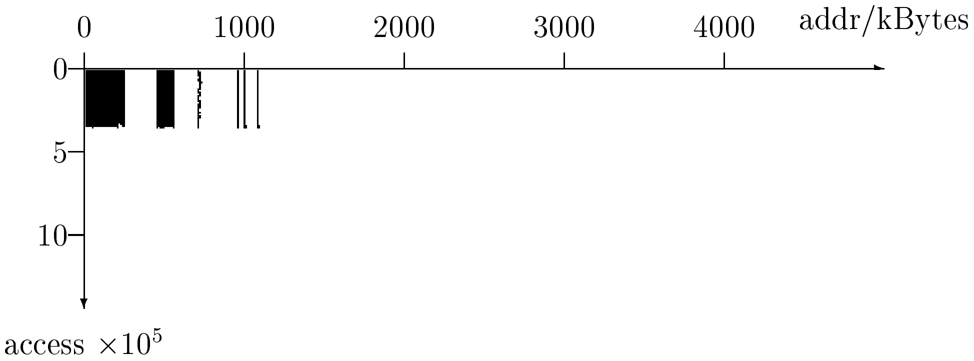

Figure 6: Memory access map for DISK_TC, input = G(10000, 0.14, 10). A pixel at position (x, y) shows that yth memory access has referred to the location x.

becomes more dense and accesses are more scattered. This causes page faults. The auxiliary data structures are heavily used in both phases. They cause a significant number of page faults.

In Figure 6, DISK_TC is given the same input as BTC in Figure 5. Note that the pictures are presented on the same scale. BTC uses much memory and does many memory accesses compared to DISK_TC. 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 462-542kBytes. At 543-703kBytes is the work stack and at 704-944kBytes is the control stack. The rest of the memory is used for transitive closure representation (addresses 944-1127kBytes), 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 page faults are caused by reading the input. Some page faults 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 page faults.

7 Conclusions

We presented a new transitive closure algorithm, called DISK_TC and an experimental study of its performance in a disk environment. We compared its performance with BTC, which was reported to be the best algorithm in disk memory environment [26]. We analyzed and compared the performance of the algorithms for large inputs and used statistically sound methods to state the accuracy of our results.

To analyze the algorithms, we built an environment for simulating memory I/O.

The environment, called DBE, has a special purpose compiler, which instruments the algorithm code to produce memory access trace instead of performing true disk I/O. The trace is given to a simulator that simulates disk I/O according to a memory management scheme. The implemented environment is suitable for performance studies of memory intensive algorithms in general. It can be used, for example, to study processor cache performance instead of disk cache performance.

In our analysis, DISK_TC was found superior to BTC. Besides the quantitative analysis, we conducted a qualitative analysis. The qualitative analysis revealed why BTC is slower than DISK_TC. The main reason is the representation of the data. DISK_TC uses a very compact representation for the closure and has only a few auxiliary data structures. The closure representation of BTC is large and it has large auxiliary data structures that it uses very intensively. Ioannidis *et al.* [26] assumed in their analysis that auxiliary data structures are kept in main memory. However, the auxiliary data structures of BTC often occupy a larger portion of the memory than the input graph.

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] R. Agrawal, S. Dar, and H.V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, September 1990.

[3] R. Agrawal and H.V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International VLDB Conference*, 1987.

[4] R. Agrawal and H.V. Jagadish. Multiprocessor transitive closure algorithms. In *Proceedings of International Symposium on Databases and Distributed Systems*, pages 55–66, Austin, Texas, December 1988.

[5] R. Agrawal and H.V. Jagadish. Hybrid transitive closure algorithms. In *Proceedings of the 16h International VLDB Conference*, pages 326–334, Brisbane, Australia, 1990.

[6] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[7] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *ACM-SIGMOD 1986 Conference on Management of Data*, pages 16–52, 1986.

[8] Bolognesi and Smolka. Fundamental results for the verification of observational equivalence: A survey. In *Proceedings of the IFIP Int. Conf. on Protocol Specification, Testing and Verification VII.* North-Holland, 1988.

[9] F. Cacace, S. Ceri, and M.A.W. Houtsma. An overview of parallel strategies for transitive closure on algebraic machines. In *Proceedings of the Workshop on Parallel Database Systems*, volume 503 of *Lecture Notes in Computer Science*, pages 44–62. Springer-Verlag, 1990.

[10] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[11] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 1–6, 1987.

[12] S. Dar. *Augmenting Databases with Generalized Transitive Closure*. PhD thesis, University of Wisconsin, Madison, Computer Sciences Department, 1993. Technical Report #1206.

[13] S. Dar and H.V. Jagadish. A spanning tree transitive closure algorithm. In *Proceedings of the IEEE 8th International Conference on Data Engineering*, pages 2–11, Tempe, Arizona, February 1992.

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

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

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

[17] M.J. Fischer and A.R. Meyer. Boolean matrix multiplication and transitive closure. In *Conference Record 1971 12th Annual Symposium on Switching and Automata Theory*, pages 129–131, East Lansing, Michigan, October 1971. IEEE Computer Society.

[18] 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, April 1991.

[19] A. Goralcikova and V. Koubek. A reduct and closure algorithm for graphs. In *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 301–307. Springer-Verlag, 1979.

[20] J.N. Hooker. Needed: an empirical science of algorithms. *Operations Research*, 42(2):201–212, March-April 1994.

[21] M.A.W. Houtsma, P.M.G. Apers, and G.L.V. Schipper. Data fragmentation for parallel transitive closure strategies. In *Proceedings of the IEEE 9th International Conference on Data Engineering*, pages 447–456, Vienna, Austria, April 1993.

[22] Y-N. Huang and J-P. Cheiney. Parallel computation of direct transitive closures. In *Proceedings of the IEEE 7th International Conference on Data Engineering*, pages 192–199, Kobe, Japan, April 1991.

[23] Y.E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proceedings of the 12th International VLDB Conference*, pages 403–411, 1986.

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

[25] 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.

[26] 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.

[27] 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.

[28] B. Jiang. A suitable algorithm for computing partial transitive closures in databases. In *Proceedings of the IEEE 6th International Conference on Data Engineering*, pages 264–271, Los Angeles, California, February 1990.

[29] R. Kabler, Y.E. Ioannidis, and M. Carey. Performance evaluation of algorithms for transitive closure. *Information Systems*, 17(5):415–441, September 1992.

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

[31] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proceedings of the 13th International VLDB Conference*, pages 267–274, 1987.

[32] C. McGeoch. Analyzing algorithms by simulation. *Computing Surveys*, 24(2):195–212, June 1992.

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

[34] E. Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52:207–213, 1994.

[35] E. Nuutila. *Efficient transitive closure computation in large digraphs.* PhD thesis, Helsinki University of Technology, 1995. Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, No. 74.

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

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

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

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

[40] A. D. Samples. Mache: No-loss trace compaction. In *Proceedings of the Sigmetrics 89 Conference*, pages 89–97, 1989.

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

[42] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transaction on Software Engineering*, SE-3(1), 1977.

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

[44] J. Ullman and M. Yannakakis. The input/output complexity of transitive closure. In *Proceedings of the ACM-SIGMOD 1990 International Conference on Management of Data*, pages 44–53, 1990.

[45] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.

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