Hybrid Transitive Closure Algorithms

Rakesh Agrawal

IBM Almaden Research Center
San Jose, California 95120

H. V. Jagadish

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We present a new family of *hybrid* transitive closure algorithms, and present experimental results showing that these algorithms perform better than existing transitive closure algorithms, including matrix-based algorithms that divide a matrix into stripes or into square blocks, and graph-based algortihms. This family of algorithms can be generalized to solve path problems and to solve problems in which some selection criteria have been specified for source or destination nodes.

1. INTRODUCTION

Transitive closure is regarded to be an important operation for the next generation of database systems [2, 5, 6, 12, 13, 15, 17, 19, 21], and considerable research has been devoted to designing algorithms for computing the transitive closure of database relations [1, 4, 9-11, 16, 24]. These algorithms can be classified into three major families. *Iterative* algorithms, such as semi-naive [4], logarithmic [10, 24], and variations thereof [9, 10, 16], compute transitive closure by repeatedly computing a relational algebraic expression, stopping when no more new answer tuples are generated, after a number of iterations that depends on the underlying database. *Direct* algorithms, on the other hand, process each element (a node or an edge) a constant number of times (usually once), and terminate after such processing is complete independent of the underlying data. In direct algorithms, there are two families. *Matrix-based* direct algorithms, such as in [1, 25, 26], are best understood in terms of a matrix representation and manipulation. *Graph-based* direct algorithms, such as in [7, 8, 11, 18, 20], are best understood in terms of a graph traversal. Graph-based algorithms often coalesce nodes belonging to the same strongly connected component into one node since these nodes will have identical successors, and process nodes of the condensed acyclic graph so obtained in a reverse topological order, adding to a node the successor sets of its immediate successors.

There is empirical evidence that blocked matrix-based direct algorithms perform significantly better than the iterative algorithms [1]. Three major factors contribute to their better performance: i) better memory utilization due to blocking, ii) efficient removal of duplicates, and iii) use of a careful processing order, rather than iteration, for termination. As noted in [11], duplicates can be removed efficiently in the graph-based algorithms as well, and they also do not require repeated iteration for termination. An advantage of the graph-based algorithms over the matrix-based algorithms is that they are $O(n \cdot e)$ algorithms whereas the matrix algorithms are $O(n^3)$ algorithms, where n is the number of nodes and e the number of arcs in the graph. The problem with the graph-based algorithms is that these algorithms are difficult to implement efficiently in an environment where the database is disk-resident [14].

We present a new family of *hybrid* transitive closure algorithms and present experimental results showing that these algorithms perform better than existing matrix-based and graph-based algorithms.

Recently a new transitive closure algorithm that processes the matrix in squares rather than stripes has been proposed [23], and the *worst case* I/O complexity of this algorithm has been shown to be better than than an algorithm in which the matrix is divided into stripes. We show, through experiments, that the new hybrid algorithm also outperform this algorithm for a wide range of graph size and memory size choices.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

Besides computing reachability, the hybrid algorithms can also be used to solve the class of well-formed decomposable path problems. Included in this class are many problems of practical interest such as bill of materials, shortest path, critical path, path of maximum reliability, etc. They can also be used to solve problems in which some selection criteria has been specified for source or destination nodes. For lack of space, this generalization is not discussed here. See [3] for details.

The rest of the paper is organized as follows. In Section 2, we give a brief review of matrix-based and graph-based algorithms. Hybrid algorithms are introduced in Section 3. Section 4 presents the result of the performance evaluation of hybrid algorithms. We conclude with some final observations in Section 5.

2. BACKGROUND

We briefly review the features of matrix-based and graph-based transitive closure algorithms that bear comparison with the hybrid algorithms.

2.1 Matrix-Based Direct Algorithms

Given an $n \times n$ adjacency matrix of elements a_{ij} over an n-node graph, with a_{ij} being 1 if there is an arc from node i to node j, and 0 otherwise, the Warshall algorithm [26] computes the transitive closure of the given graph as follows:

$$\bigvee_{j=1}^{n} \bigvee_{i=1}^{n} \text{process } a_{ij}$$

"Processing" of an element a_{ij} involves examining whether a_{ij} is 1, and if it is, then making every successor of j a successor of i. Thus, the Warshall algorithm computes closure by "processing" every element of the matrix exactly once, column by column from left to right, and from top to bottom within a column.

It has been shown [1] that the matrix elements can be processed in *any* order, provided the following two constraints are maintained:

1. For all i, j, k, processing of the element a_{ik} precedes processing of the element a_{ij}, *iff* $k < j$, and
2. For all i, j, k, processing of the element a_{kj} precedes the processing of the element a_{ij}, *iff* $k < j$.

Various processing orders can be derived subject to these two constraints, giving rise to a whole family of *Warshall-derived* algorithms. The Warren algorithm [25] that processes matrix elements in row order but in two passes[1]:

$$\bigvee_{i=1}^{n} \bigvee_{j=1}^{i-1} \text{process } a_{ij}$$

$$\bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \text{process } a_{ij}$$

can be viewed as a Warshall-derived algorithm, since it observes the two precedence constraints listed above. The blocked row and blocked column algorithms presented in [1] are Warshall-derived algorithms that process matrix elements in such a way that the I/O traffic between disk and memory is minimized.

1. Only the lower triangular half is examined in the first pass, and the upper triangular half is examined in the second pass.

2.2 Graph-Based Direct Algorithms

Purdom, in [18], made two key observations:

i. During the computation of transitive closure of a directed acyclic graph, if node A precedes node B in a topological sort of the nodes in the graph, additions to the successor set of node A cannot affect the successor set of node B. One should, therefore, compute the successor set of B first and then that of A. By thus processing nodes in reverse topological order, one need add to a node only the successor lists of its immediate successors, since the latter would already have been fully expanded.

ii. All nodes within a strongly connected component in a graph have identical reachability properties, and the condensation graph obtained by collapsing all the nodes in each strongly connected component into a single node is acyclic.

Tarjan [22] developed an $O(e)$ algorithm for determining strongly connected components of a graph by means of a depth-first search, which also produces as a by-product a topological sort on the components. It has been observed [7, 8, 11, 20] that it is possible to modify Tarjan's algorithm in a way that the successor lists are also expanded as the strongly connected components are being determined, and thus compute the transitive closure.

3. HYBRID ALGORITHMS

The hybrid algorithms we propose in this section are best described starting with the matrix-based algorithms described in the previous section. In matrix-based algorithms, row i of the adjacency matrix corresponds to the successor set of the node numbered i, but the nodes are numbered arbitrarily. Instead of arbitrary numbering, we use topological ordering to assign node numbers, and then exploit this ordering to incorporate the optimizing features of the graph-based algorithms in a matrix framework.

Our algorithms have two distinct passes. In the first pass, we obtain a condensation graph for the given graph in which each non-trivial strongly connected component is identified and coalesced into one node. A topological sort of the condensation graph is also obtained at the same time. The transitive closure is computed in the second pass. We present algorithms only for the second pass, assuming that the first pass has already been performed using, say, the Tarjan algorithm [22].

3.1 Basic Algorithm

Let us consider an acyclic graph G and number its nodes in a topological sort order. Thus, the source node of any arc has a higher node number than its destination node. Obtain an adjacency matrix representation M of G, such that row i

represents to successor set of node i, and matrix element (i,j) is 1 if there is an arc (i,j) in G, and 0 otherwise. M will be a lower triangular matrix.

Here is the basic hybrid algorithm:

Algorithm 1 (The basic hybrid algorithm):

```
For i from 1 to n
    Copy row i into a temporary ĩ
    For j from i−1 to 1
            /* process from right to left within a row */
        If ĩ(i,j) ≠ 0
                /* immediate successor optimization */
            call add_succ(i, j, ĩ)
                    /* add successors of j to i */

procedure add_succ(i, j, ĩ):
    For k from 1 to j−1
        If (j,k) = 1
            If ĩ(i,k) = 1
                ĩ(i,k) = 0  /* marking optimization */
            else
                (i,k) = 1
```

This algorithm is similar to the Warren algorithm [25] in that it processes matrix elements in row order. However, unlike Warren

A. only those elements a_{ij} which were 1 to begin with result in addition of successors of j to i (immediate successor optimization);

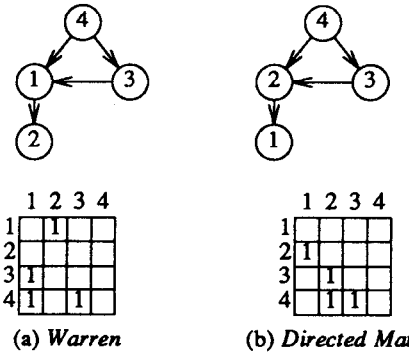
B. while a row is being processed, some elements which were 1 to begin with are treated as if they were 0 (marking optimization); and

C. matrix elements are processed right to left.

(A) implies that this algorithm, like graph-based algorithms, adds to a node only the successor sets of its immediate successors. The temporary row $\tilde{i}$ is initialized to the set of immediate successors of i, and $\tilde{i}$ is used to determine whether the successors of nodes j should be added to i. Before row i is processed, all rows numbered less than i have already been processed. Thus, before processing any node, it is guaranteed that all its successors have been processed and fully expanded, since successors correspond to rows that have a lower row number in the matrix than the row number of the node being processed.

The effect of (B) and (C) is similar to the marking optimization proposed in [11]. If a node i has two immediate successors j and k such that k is also a successor of j and it is guaranteed that the node j has been fully expanded before i is processed, then it is sufficient to add the successors of j to i and the successors of k need not be added to i. (C) ensures that if two immediate successors j and k of i are such that k is also a successor of j, then j is processed before k. (B) ensures that later on, when element (i,k) is processed, the successor set of k will not be added to i.

Consider, for example, the simple graph shown in Figure 3.1 and contrast the computation of its transitive closure using the Warren algorithm and Algorithm 1. In the Warren algorithm, nodes are numbered arbitrarily, whereas nodes are assigned numbers in the topological sort order in Algorithm 1. Figure 3.1 also shows the adjacency matrix corresponding to the two node numberings.



(a) *Warren* (b) *Directed Matrix*

Figure 3.1. Difference in computations in Warren and Directed algorithms

When processing row 3 using the Warren algorithm, first the element (3,1) is processed, the successor set of 1 is fetched into memory and added to the successor set of 3, thus transforming the element (3,2) into a 1. Now the element (3,2) is processed and the successor set of 2 is fetched. When processing row 3 using Algorithm 1, only the successor set of 2 is fetched, and the successor set of 1 is not fetched due to the immediate successor optimization. Similarly, when processing row 4 using the Warren algorithm, the successor sets of 1, 2, and 3 are fetched. However, when using Algorithm 1, only the successor set of 3 is fetched. The successor set of 1 is not fetched due to the marking optimization, and the successor set of 2 is not fetched due to the immediate successor optimization.

There is never a case when Algorithm 1 will fetch a successor set, but the Warren algorithm will not, irrespective of node numbering. Provided that the node numbering is same, the sizes of successor sets, when fetched, are identical. Algorithm 1, therefore, for a given (reverse topologically sorted) node ordering, performs less or equal I/O than the Warren algorithm. The disadvantage of Algorithm 1 is that it requires a topological sort of the given graph. However, our experimental results (reported in Section 4) show that the cost of topological sort is insignificant compared to the cost saving when computing the transitive closure.

The major difference between Algorithm 1 and the graph-based algorithms is that the graph-based algorithms are depth-first recursive descent algorithms, whereas Algorithm 1 is a breadth-first algorithm, making it amenable to efficient blocking. Moreover, processing of elements from right to left within a row in Algorithm 1 guarantees that the marking optimization is performed in all possible cases. However, the marking optimization in a graph-based algorithm depends on the order in which children of a node are examined. Consider, for example, the graph shown in Figure 3.2. In a graph-based algorithm, if node $i+1$ is visited before node $i+2$, the successor set of node $i+1$ will be added to the successor set of node $i+3$ twice: once directly and then through the addition of the successor set of node $i+2$. The hybrid algorithm, on the

other hand, will add the successors of node i+1 to node i+3 only indirectly by adding the successors of node i+2 to node i+3.

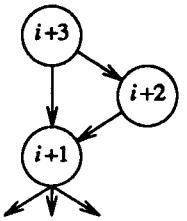


Figure 3.2. Order dependence of the marking optimization in the graph-based algorithm

3.2 Blocked Algorithm

We now discuss how Algorithm 1 can be blocked. Partition the matrix into blocks of contiguous rows. As we will see shortly, blocks can be determined dynamically, and the number of rows in a block could be different for different blocks. If a block b_l consists of rows i_s through i_e, then the elements (i,j) such that $i_s \le i \le i_e$ and $i_s \le j \le i_e$ will be referred to as the diagonal block elements of b_l and the remaining elements in b_l will be referred to as the off-diagonal block elements. The rest of the elements in the lower triangular half of the matrix will be referred to as the off-block elements (see Figure 3.3).

Algorithm 2 (The blocked hybrid algorithm):

Assume matrix partitioned into m blocks.

Do the following for each block b_l, l = 1, 2, ..., m:

```
Let the block b_l consist of rows i_s to i_e.
Fetch rows i_s through i_e into memory.
Copy into rows ĩ_s through ĩ_e, respectively.

/* process the elements in the off-diagonal block
   column-by-column from right to left */

For j from i_s−1 to 1
    For i from i_s to i_e
        if (ĩ,j) ≠ 0 /* immediate successor optimization */
            fetch the row j if not already in memory
                                /* blocking benefit */
            add_succ(i, j, ĩ)

/* process the elements in the diagonal block
   row-by-row from right to left */

For i from i_s to i_e
    For j from i to i_s
        if (ĩ,j) ≠ 0  /* immediate successor optimization */
            add_succ(i, j, ĩ)
```

Since the elements in the off-diagonal block are processed column by column, an off-block successor set is fetched at most once during the processing of a block. Consider, for example, the graph shown in Figure 3.4. When processing the

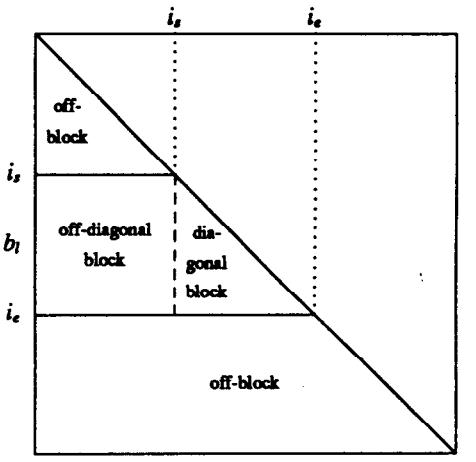


Figure 3.3. Diagonal block, off-diagonal block, and off-block elements for the block b_l

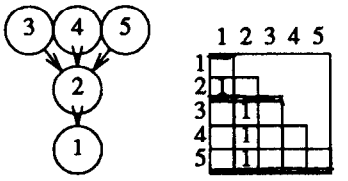


Figure 3.4. Benefit of blocking

block consisting of rows 3 through 5, the elements (3,2), (4,2), and (5,2) are processed in that order, the successor set of 2 is read once, and is added to the successor sets of 3, 4 and 5. With the basic hybrid algorithm, elements are processed in row-order, and the successor set of 2 will be read three times.[2]

The elements in the diagonal block may be processed in row-order without affecting the I/O performance because all the relevant rows are already in memory.

The immediate successor optimization is performed as in the case of the basic algorithm. Within an off-diagonal block and a diagonal block, elements are processed right to left, but the off-diagonal block is processed before processing the diagonal block. The result is that the algorithm performs marking optimization, but separately within the off-diagonal block and diagonal block.

3.3 Dynamic Blocking

Block sizes can be determined dynamically as in [1] using the following greedy algorithm. Partition the memory into three logical segments. In the first segment, called the *loading area*, the successor sets are loaded one at a time until the loading area fills up. The number of successor sets that could

2. The successor sets, in general, are large, so that there is a small probability of finding the successor set of 2 in system buffers when processing row 4 after row 3 has been processed.

be accommodated in the loading area determines the size of the current block.

As the successors sets expand, new tuples are created in the *expansion area*. The third segment of the memory, called the *off-block* area is reserved for reading one successor set at a time. This successor set is used for expanding the successor sets in the current block. The expansion area grows toward the off-block area. As successors are added to the nodes in current block, the expansion area may fill up and hit the boundary of the off-block area. This situation can be handled by dynamically reducing the size of the current block. Reblocking simply involves taking out the last row in the current block and freeing up the space in the loading and expansion areas devoted to it.

4. PERFORMANCE EVALUATION

We now present the results of simulation experiments evaluating the performance of the hybrid algorithms. We describe the algorithms studied, make a few observations on the performance evaluation methodology, discuss the datasets, and then present the results.

4.1 Algorithms

The performance of the hybrid algorithm was compared against the Blocked Row algorithm presented in [1] and the graph-based algorithm (referred to as the *DFS algorithm* in the rest of the paper) presented in [11].

Blocked Row and Hybrid algorithms were implemented by partitioning the memory into three segments: i) the loading area, for initial loading of successor sets in the current block, ii) the expansion area, for creating new tuples, and iii) the off-block area, for reading one successor set that is used for expanding the successor sets in the current block. Block sizes were determined using the greedy algorithm described in Section 3.3. The simulation kept track of old values of tuples in the current block, necessary in the hybrid algorithm, and reduced accordingly the memory availability for the hybrid algorithm.

The strategy for implementing the DFS algorithm in a disk-based environment is not presented in [11]. Our implementation of the DFS algorithm tries to keep as much of the successor sets stack in memory as possible. If space in memory runs out, the successor set at the bottom of the stack is paged out. If this set has been updated since it was last read in to memory, then it is written out to disk, otherwise it is simply purged from memory. The successor set at the bottom of the stack is selected for paging out since the activity is typically concentrated at the top of the stack.

To fully utilize the memory available, we added a further optimization. After a successor set is fully expanded and popped from the stack, it is written to disk, but not purged from the memory. This buffering strategy avoids, for example, re-reading of the successor set of D when processing the node Z in Figure 4.1. The successor sets still on the stack have priority for memory residency over these buffered popped-off successor sets, so that when memory fills up, all these extra buffered sets are purged one by one, before any on the stack is paged out.

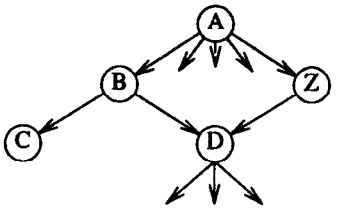


Figure 4.1. Buffering in the DFS algorithm

Marking optimization was also performed. Thus in a graph such as in Figure 3.5, if the successor set of node 4 is added to the successor set of node 5, it is not necessary also to add the successor set of node 2, another immediate successor of node 5 that is also a successor of node 4. However, as noted in Section 3.1, the entire saving possible from marking optimization may not be realized depending on the order in which the immediate successors of a node are expanded. At the expense of some additional book-keeping and some additional memory space, it is possible to defer the unioning of successor sets until the marking optimization can be applied. But the optimization then applies only to the effort to perform the union in memory and not to the effort in fetching the successor sets from disk. To the extent that the I/O is the primary cost determinant for the algorithm, the deferred unioning provides little benefit, and has the disadvantage of consuming additional memory. We, therefore, did not defer successor set unions.

4.2 Experimental Set Up

Synthetic graphs were used in the performance evaluation experiments. Two parameters of a graph were identified as important: the number of nodes, and the average degree of each node. These two parameters were varied to create a set of random graphs.

We report here the results for the bill of materials problem. We also considered reachability computations for all the algorithms, and found trends to be similar to those for the bill of material problem. Since bill of materials problem is ill-defined for cyclic graphs, experiments were restricted to acyclic graphs.

The number of tuple I/Os was used as the performance metric. The size of memory was also specified in number of tuples. Memory sizes were chosen so that the complete closure of the graph would not fit in main memory, as would be the case in a disk-based environment.

4.3 Performance Results

Figure 4.2 shows the relative performance of Hybrid, Blocked Row, and DFS algorithms. We have normalized the total number of tuple I/Os required to compute the closure with respect to the tuple I/Os required for the directed matrix algorithm. Total I/Os have been plotted by varying both the number of nodes and the average degree. The numbers for the Blocked Row algorithm are for a version of the algorithm in which the graph was first topologically sorted and then processed using only the first pass of the Blocked Row

algorithm. This version of the Blocked Row algorithm was found to always perform better than the two pass version. Both for hybrid and blocked row algorithms, the total I/O includes the I/O for topologically sorting the graph and writing out the sorted result. It is clear from the graphs that the hybrid algorithm consistently performs better than both DFS and Blocked Row algorithms.

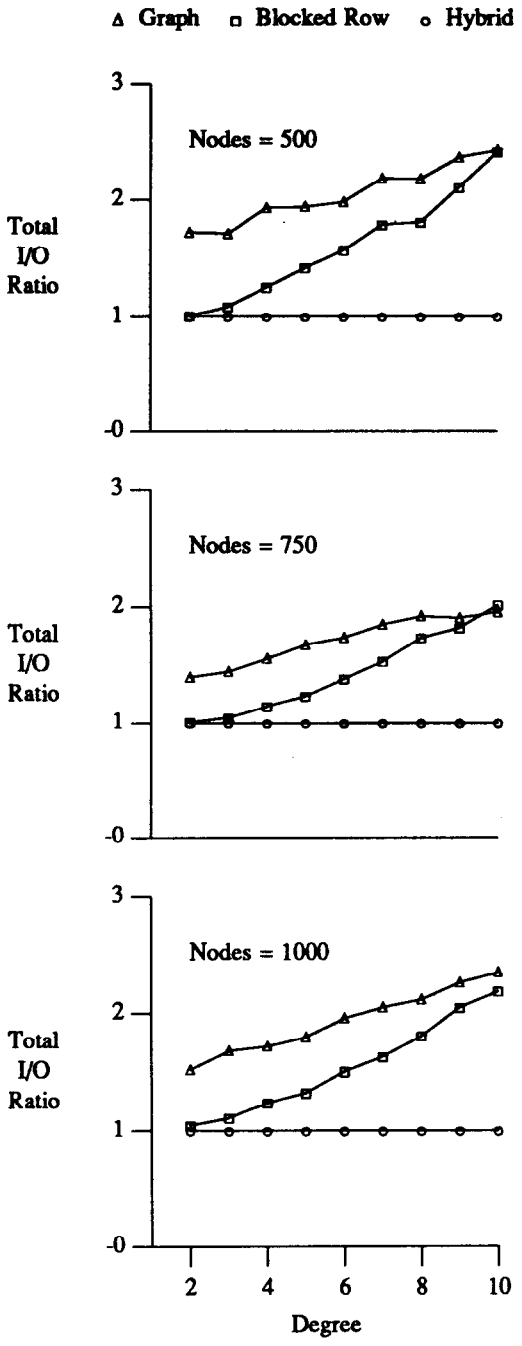


Figure 4.2. Comparative performance

Let us now analyze these performance results in detail.

The cost of topological sort in Hybrid and Blocked Row algorithms turns out to be a small fraction of the cost of computing transitive closure. Figure 4.3 shows the topological sort component as a fraction of the total closure cost for these algorithms for 500 node graphs. Similar results were obtained for graphs of other sizes.

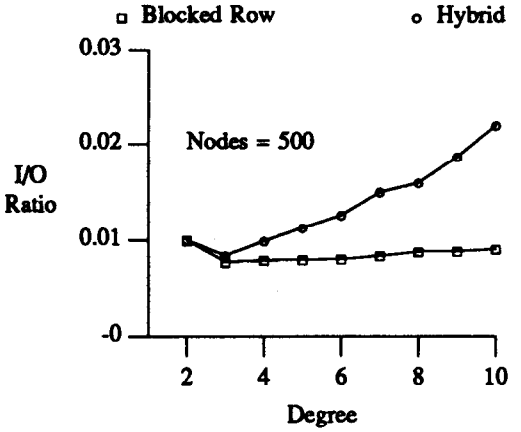


Figure 4.3. Cost of topological sort as a fraction of total cost

The sorting cost in number of tuple I/Os for all the relations was twice the number of tuples in the relation — for each tuple, one I/O was incurred to read it into memory and one to write it back in the sorted order. This result is not surprising. Although relations were larger than the memory size, the maximum number of tuples that need to be memory resident at any time depends on the length of the longest path in the corresponding graph, which explains why no tuple was re-read during the topological sort.

Coming to the transitive closure cost, the I/O cost consists of:

1. R_i: Reads of tuples when a successor set is brought into memory to be expanded.
2. W_i: Writes of tuples when an expanded successor set is written back to disk.
3. R_j: Reads of tuples when a successor set is brought into memory to expand another successor set.

Ignoring (3) for the moment, both Hybrid and Blocked Row are "read-once" and "write-once" algorithms in that during the computation of a transitive closure a successor set is read into memory, expanded, and written back to disk only once. For both of these algorithms, R_i equals the number of tuples in the original relation and W_i equals the number of tuples in the closure. However, the DFS algorithm does not have this "read-once" and "write-once" property. If the graph is such that all successor sets currently on the stack cannot be memory resident, some successor sets from the stack must be paged out. If any of these successor sets have been updated, writes become necessary. In any event, the paged out successors are re-read. Let the number of tuples in the original relation be $|R|$ and in the closure relation $|TC|$. Define excess reads as $R_i - |R|$, and excess writes as $W_i - |TC|$. Figure 4.4 shows excess reads and excess writes in the DFS algorithm due to stack paging. Note that the size of the closure relation, $|TC|$, is several times the size of the original relation, $|R|$. (In this particular example, 20 to 70 times).

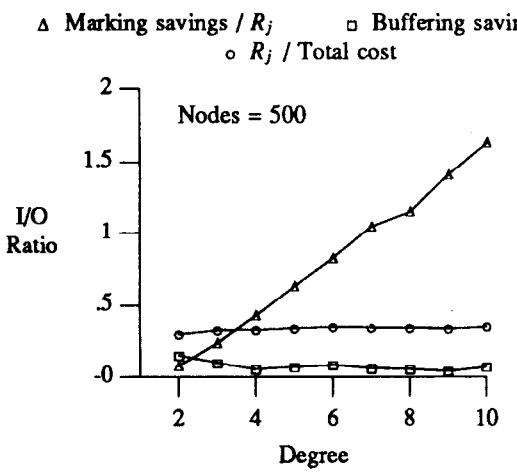


Figure 4.7. Savings due to marking and buffering in the DFS algorithm

paging of successor sets at the bottom of the stack.

Finally, we note that the performance of the Hybrid algorithm can be further improved by using a buffering strategy similar to the one implemented for the DFS algorithm to reduce R_j. After processing block B_l consisting of rows i_s to i_e, we first process the off-diagonal block elements in the next block B_l+1 column by column and from right to left, that is, we first process elements in the column j such that $j = i_e$, then elements in the column j such that $j = i_e - 1$, and so on. If there is a 1 for an element (i,i_e) in the column b_e, the successor set of i_e is added to the successor set of i. We, therefore, can buffer the expanded successor sets resulting from processing B_l and purge the successor set of i_e only after all elements in the column i_e have been processed in B_l+1, etc.

4.4 Comparison with the Grid algorithm

Recently a new transitive closure algorithm that processes the matrix in squares rather than stripes has been proposed [23], and the *worst case* I/O complexity of this algorithm has been shown to be better than the blocked row algorithm by a factor of *(number of nodes)/(memory size in tuples*$^{1/2}$*)*. The algorithm (referred to as the *grid* algorithm henceforth) is reproduced here for reference:

Partition the matrix into square sub-matrices
  that will each fit into a specified fraction of memory.
Let there be $f \times f$ sub-matrices.
If $M_{i,j}$ is a sub-matrix,
  write its (reflexive and) transitive closure as $M^*_{i,j}$.

Then execute:

For $k = 1$ to f
    $M_{k,k} = M^*_{k,k}$;
    for $i = 1$ to f
        for $j = 1$ to f
            $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,k} \times M_{k,j}$

We compared the performance of the hybrid algorithm with this algorithm also. A straightforward implementation of the grid algorithm requires four sub-matrices $M_{k,k}$, $M_{i,k}$, $M_{k,j}$, and $M_{i,j}$ to be in memory at the same time. However, for matrix multiplication, the entire matrix need not be in memory at the same time. Therefore, we implemented the grid algorithm as follows:

For $k = 1$ to f
  Read $M_{k,k}$ from disk ;
  $M_{k,k} = M^*_{k,k}$;
  for $i = 1$ to f
(*) Read $M_{i,k}$ from disk, row by row ;
    $T_{i,k} = M_{i,k} \times M_{k,k}$;
    for $j = 1$ to f
(**) Read $M_{i,j}$ and $M_{k,j}$ from disk, column by column ;
      $M_{i,j} = M_{i,j} + T_{i,k} \times M_{k,j}$;

In step (*), after one row of $M_{i,k}$ has been read from disk, the corresponding row of $T_{i,k}$ can be computed. The next row of $M_{i,k}$ can then overwrite the current row of $M_{i,k}$. Thus only one row of $M_{i,k}$ needs to be memory resident at a time, during step (*). However, storage is required for all of $T_{i,k}$ and all of $M_{k,k}$.

In step (**), one column of $M_{k,j}$ can be read from disk, the corresponding column of $M_{i,j}$ can be updated, and then we can proceed to the next column. Thus storage is required in memory only for one column each of these matrices rather than the entire matrix. Thus the size of the partition, b, is determined from the equation $2 \times b^2 + 2 \times b$ = memory size.

In [23], better asymptotic bounds have been proved for sparse acyclic graphs, and an intricate algorithm has been presented. We did not implement that algorithm because of its complexity, but to take advantage of the sparseness and acyclicity of the graphs we are studying, we also considered a version of the grid algorithm in which the graph is topologically sorted before the transitive closure computation begins. Then the upper triangular half of the matrix will consist of zeros and the grid algorithm can take advantage of this property. In the following performance results, this version of the grid algorithm is referred to as the *triangularized grid algorithm.*

Figure 4.8 shows the performance of the grid algorithm compared to the hybrid algorithm for graphs of different sizes. Clearly, the directed matrix algorithm uniformly outperforms the grid algorithm.

Let us see why we see this performance difference. Observe that the grid algorithm requires that the blocks all be equal, in consequence of which, dynamic block sizing is difficult. (One may have long finished computing with one block before one discovers that it has to be decreased in size since some other block overflowed). As such, one has to be pessimistic, and assume that each block may potentially fill up, as we have done in the equation given in the previous paragraph. Moreover, the grid algorithm does not have the "read-once, write-once" property, which the hybrid algorithm has. Finally, the marking and immediate successor optimizations described in this paper are not applicable to this algorithm either.

We also studied the effect of memory size on the relative performance of two algorithms, since the asymptotic bound for

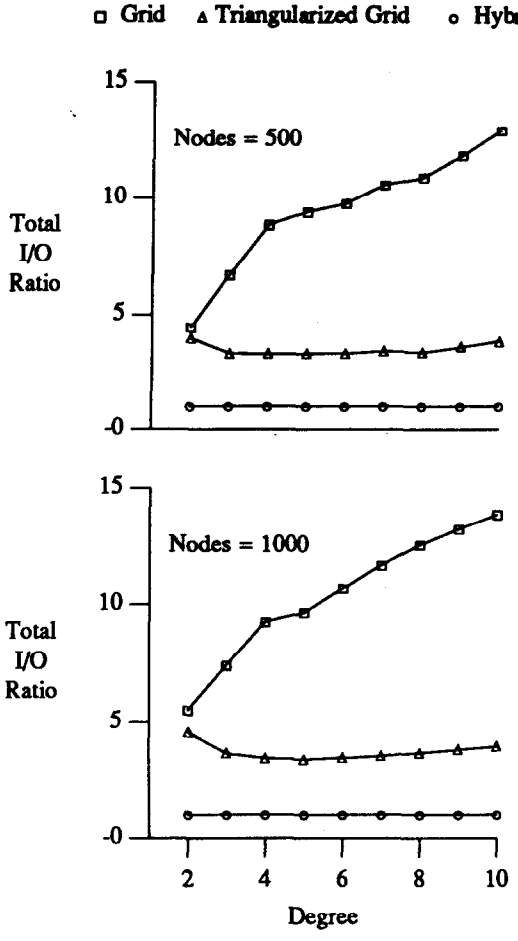


Figure 4.8. Comparative performance of grid and hybrid algorithms

the grid algorithm improves as the memory size is reduced. Figure 4.9 shows the performance of the triangularized grid algorithm relative to the hybrid algorithm for 500 node graph. The hybrid algorithm requires at least two successor sets worth of main memory, which in the worst case can be 1000 tuples. We, therefore, varied memory size from 1000 tuples and up. This graph shows that the hybrid algorithm has uniformly better performance than the grid algorithm over all memory sizes, with the relative performance of the hybrid algorithm being even somewhat better for small memory sizes.

5. SUMMARY

We considered the problem of computing transitive closure in an environment in which the database is disk-resident and the transitive closure too big to fit in memory. We introduced a new family of hybrid transitive closure algorithms and presented experimental results showing that these algorithms perform better than the blocked row [1] and the grid [23] matrix-based algorithms, and the graph-based algorithms [11]. The hybrid algorithms benefit from efficient blocking, immediate successor optimization, and marking optimization. The blocked row algorithm can also benefit from the immediate successor optimization and blocking, but loses to the hybrid algorithm due to the absence of marking

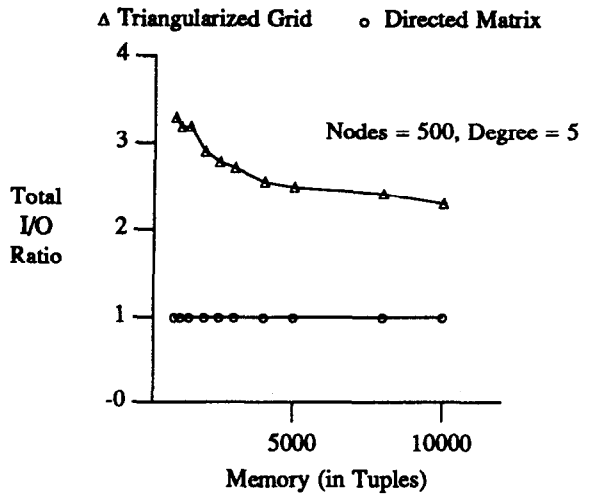


Figure 4.9. Comparative performance as memory is varied

optimization. The immediate successor optimization is an inherent property of a graph-based algorithm, but the hybrid algorithm wins over the graph-based algorithm due to better blocking and larger savings in marking optimization and excess I/O in the graph-based algorithm due to paging of successor sets at the bottom of the stack. The grid algorithm benefits from blocking that is very efficient in the worst case, but is static and hence may not do so well in the normal case. In addition, it does not benefit from the immediate successor or marking optimizations.

The algorithms presented in this paper may be used to construct building blocks for future extended database systems. Although presented in the context of database systems, these algorithms have larger applicability and may be used in other problem domains that require reachability or path computation over a large graph.

ACKNOWLEDGEMENTS

We wish to thank Shaul Dar and Bruce Hillyer for their insightful comments and suggestions.

REFERENCES

[1] R. Agrawal, S. Dar, and H. V. Jagadish, "Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM Trans. Database Syst.*, to appear. (Preliminary version appeared as: R. Agrawal and H.V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987).

[2] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590. Also in *IEEE Trans. Software Eng.* 14, 7 (July 1988), 879-885.

[3] R. Agrawal and H. V. Jagadish, "Hybrid Transitive Closure Algorithms," AT&T Bell Laboratories Technical Memorandum, 1990.

[4] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," Tech. Rept. DB-004-85, MCC, Austin, Texas, 1985.

[5] J. Biskup, U. Raesch, and H. Stiefeling, "An Extended Relational Query Language for Knowledgebase Support," Institut fuer Informatik, Hildesheim, West Germany, 1987.

[6] I. F. Cruz and T. S. Norvell, "Aggregative Closure: An Extension of Transitive Closure," *Proc. IEEE 5th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1989.

[7] J. Ebert, "A Sensitive Transitive Closure Algorithm," *Information Processing Letters*, **12**, 1981, 255-258.

[8] J. Eve and R. Kurki-Suonio, "On Computing the Transitive Closure of a Relation," *Acta Informatica*, **8**, 1977, 303-314.

[9] U. Guntzer, W. Kiessling, and R. Bayer, "On the Evaluation of Recursion in Deductive Database Systems by Efficient Differential Fixpoint Iteration," *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 120-129.

[10] Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators," *Proc. 12th Int'l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 403-411.

[11] Y. E. Ioannidis and R. Ramakrishnan, "An Efficient Transitive Closure Algorithm," *Proc. 14th Int'l Conf. Very Large Data Bases*, Aug.-Sept. 1988, 382-394.

[12] H. V. Jagadish, R. Agrawal, and L. Ness, "A Study of Transitive Closure as a Recursion Mechanism," *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Francisco, California, May 1987, 331-344.

[13] B. Jiang, "Making the Partial Transitive Closure an Elementary Database Operation," *Proc. GI Conf. Database Systems for Office Automation, Engineering, and Scientific Applications*, Zurich, 1989.

[14] B. Jiang, "A Suitable Algorithm for Computing Partial Transitive Closures in Databases," *Proc. IEEE 6th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1990.

[15] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro, and M. Stonebraker, "Heuristic Search in Data Base Systems," *Proc. 1st Int'l Workshop Expert Database Systems*, Kiawah Island, South Carolina, Oct. 1984, 96-107.

[16] H. Lu, "New Strategies for Computing the Transitive Closure of a Database Relation," *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987.

[17] T. H. Merrett, *Relational Information System*, Reston Publishing, Reston, Virginia, 1984.

[18] P. Purdom, "A Transitive Closure Algorithm," *BIT*, **10**, 1970, 76-94.

[19] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 166-176.

[20] L. Schmitz, "An Improved Transitive Closure Algorithm," *Computing*, **30**, 1983, 359-371.

[21] S. Sippu and E. Soisalon-Soininen, "A Generalized Transitive Closure for Relational Queries," *Proc. 7th Symp. Principles of Database Systems*, March 1988.

[22] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal of Computing*, **1**, 1972, 146-160.

[23] J. D. Ullman and M. Yannakakis, "On the Input/Output Complexity of Transitive Closure," *Proc. of the ACM-SIGMOD Int'l Conf. on the Management of Data*, Atlantic City, NJ, May, 1990.

[24] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," *Proc. 1st Int'l Conf. Expert Database Systems*, Charleston, South Carolina, April 1986, 197-208.

[25] H. S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun. ACM*, **18**(4), April 1975, 218-220.

[26] S. Warshall, "A Theorem on Boolean Matrices," *J. ACM*, **9**(1), Jan. 1962, 11-12.