# THE INPUT/OUTPUT COMPLEXITY OF TRANSITIVE CLOSURE

Jeffrey D Ullman†
*Stanford University*
Mihalis Yannakakis
*ATT Bell Laboratories*

*ABSTRACT* Suppose a directed graph has its arcs stored in secondary memory, and we wish to compute its transitive closure, also storing the result in secondary memory We assume that an amount of main memory capable of holding $s$ "values" is available, and that $s$ lies between $n$, the number of nodes of the graph, and $e$, the number of arcs The cost measure we use for algorithms is the *I/O complexity* of Kung and Hong, where we count 1 every time a value is moved into main memory from secondary memory, or vice versa

In the dense case, where $e$ is close to $n^2$, we show that I/O equal to $O(n^3/\sqrt{s})$ is sufficient to compute the transitive closure of an $n$-node graph, using main memory of size $s$ Moreover, it is necessary for any algorithm that is "standard," in a sense to be defined precisely in the paper Roughly, "standard" means that paths are constructed only by concatenating arcs and previously discovered paths This class includes the usual algorithms that work for the generalization of transitive closure to semiring problems For the sparse case, we show that I/O equal to $O(n^2\sqrt{e/s})$ is sufficient, although the algorithm we propose meets our definition of "standard" only if the underlying graph is acyclic We also show that $\Omega(n^2\sqrt{e/s})$ is necessary for any standard algorithm in the sparse case That settles the I/O complexity of the sparse/acyclic case, for standard algorithms It is unknown whether this complexity can be achieved in the sparse, cyclic case, by a standard algorithm, and it is unknown whether the bound can be beaten by nonstandard algorithms

We then consider a special kind of standard algorithm, in which paths are constructed only by concatenating arcs and old paths, never by concatenating two old paths This restriction seems essential if we are to take advantage of sparseness Unfortunately, we show that almost another factor of $n$ I/O is necessary That is, there is an algorithm in this class using I/O $O(n^3\sqrt{e/s})$ for arbitrary sparse graphs, including cyclic ones Moreover, every algorithm in the restricted class must use $\Omega(n^3\sqrt{e/s}/\log^3 n)$ I/O, on some cyclic graphs

† The work of this author was partially supported by NSF grant IRI–87–22886, IBM contract 476816, Air Force grant AFOSR–88–0266, and a Guggenheim fellowship



## I. THE PROBLEM

Let us assume that data is stored in secondary memory, and computation must be performed in main memory We suppose that the cost of moving a datum from secondary to main memory or back is very large, so much so, that we shall consider the cost of data movement to the exclusion of computation cost For us, a datum is a value of small size, such as an integer whose value is polynomial in the size of the input [and thus can be represented in $O(\log n)$ bits]

We shall use $s$ throughout as the *capacity of main memory* The quantity $s$ measures the number of data items that can be stored in main memory at one time Because of our imprecision in the size of a datum, we can only talk about $s$ to within an order of magnitude Since we generally buy machines with, say, exactly 4 megabytes of main memory, not "on the order of 4 megabytes," all our results should, strictly speaking, be revised to lower, by a constant factor, the problem size that we can handle with a given amount of memory in a given time Nevertheless, the notion that main memory is expandable by a constant factor is a convenient fiction

We are primarily interested in transitive closure problems and its generalization to *closed semiring* problems as described in Aho, Hopcroft, and Ullman [1974] In the basic problem, we are given a directed graph of $n$ nodes and $e$ arcs, and we wish to find its *transitive closure*, that is, the set of pairs of nodes $(v, w)$ such that there is a path of length one or more from $v$ to $w$ In the generalized problem, we are given a label on each

arc, chosen from a closed semiring, which is a certain algebraic structure defined in Aho et al [1974], having multiplication and addition operators We wish to compute the sum over all paths from $v$ to $w$, of the product of the labels along that path, in order

### The Kung-Hong Model

This model was studied by Kung and Hong [1980], who proved some basic results about a variety of problems Their notion of an algorithm is a DAG (directed acyclic graph), in which the nodes correspond to values, the predecessors of a value $v$ are the values used to compute $v$ For example, they considered the standard algorithm for multiplication of $n \times n$ matrices, and showed that the I/O required is $\Omega(n^3/\sqrt{s})$ That amount of I/O is sufficient, as well, by an algorithm of McKellar and Coffman [1969]

They could equally as well have showed the same lower bound for the usual algorithm, due to Warshall [1962], for taking the transitive closure Indeed, our lower bound for a family of transitive closure algorithms makes heavy use of important combinatorial ideas from Kung and Hong [1980]

### Our Model

Our model is somewhat different We focus only on transitive closure and related problems, so it is natural to view steps of the algorithm not as nodes of a DAG, but as operations in which paths are discovered That is, each operation involves some number of facts that must simultaneously be in main memory A *fact* is a variable that we shall designate either $arc(v, w)$ ("there is an arc from node $v$ to node $w$") or $path(v, w)$ ("there is a path from $v$ to $w$") In the basic transitive closure problem, all these facts are Boolean-valued, we are given the arc facts, and need to compute the path facts For example, an operation might take data items representing the facts $path(v, x)$ and $path(x, w)$ and assign a new value to the fact $path(v, w)$ In the generalized, closed-semiring case, the values of arc facts are labels of arcs What we call fact $path(v, w)$ is intended, after the algorithm is complete, to be the sum over all paths from $v$ to $w$, of the product of the labels along each path

In our model, we do not assume a precedence in the order of operations, as is assumed by the DAG model of Kung and Hong Rather, when proving lower bounds, we characterize an algorithm by the set of sets of facts that must be in main memory simultaneously We should note, however, that when proving upper bounds, that is, exhibiting algorithms, we sometimes perform operations involving data other than the arc and path facts If so, we shall account for the main memory space used by these facts, under the assumption that integers whose number of bits is logarithmic in the input size may be stored in one unit of main memory Domain values from a closed semiring are also assumed to take one memory unit, even though their nature and representation is left unspecified

That model is both more and less restrictive than the Kung-Hong model it is more restrictive in that we assume a new value of a "fact," such as $path(v, x)$, always replaces its previous value in main memory The Kung-Hong, or DAG, model, allows executions in which two different DAG nodes are in memory simultaneously, even though they conceptually represent different versions of the same variable On the other hand, the Kung-Hong model only allows us to discuss one DAG at a time, which implies that the order of operations is fixed, except for operations that are independent We, on the other hand, can consider whole families of algorithms at once For example, we speak of a family that includes all $n!$ versions of Warshall's algorithm where the order in which we pivot on the various nodes is allowed to vary Each of these versions would have a different DAG, although the DAGs would differ only by renaming of nodes

```
for all i and j do
    path(i,j)  = arc(i j),
for k  = 1 to n do
    for all i and j do
        path(i,j)  = path(i,j) OR
                     (path(i,k) AND path(k,j)),
```

**Fig. 1.1.** Warshall's algorithm

### Standard Algorithms

When proving lower bounds for the dense case, we shall focus on *standard* algorithms These have the property that (for at least one input) for every triangle of nodes $\{v, w, x\}$, there is at some time simultaneously in main memory a fact about $v$ and $w$, a fact about $w$ and $x$, and a fact about $v$ and $x$ What algorithms are "standard"?[2]

[2] We do not care whether triangles are regarded as ordered or unordered, since all our results are order-of-magnitude anyway, and order only introduces a factor of 6 in the count of triangles

1 *Warshall's algorithm*, as described in Fig 1 1 We assume the nodes are numbered $1, 2, \ldots, n$ Each node, in turn, becomes $k$, the *pivot* node After $k$ has had its turn at pivot, $path(i, j)$ is 1 if and only if there is a path from $i$ to $j$ that goes through no node numbered above $k$ (although $i$ or $j$ may exceed $k$) Notice that at the last line of Fig 1 1, three facts involving the triangle of nodes $i$, $j$, and $k$ must be in main memory at once In fact, that would be true even if we counted ordered triangles

2 More generally, Kleene's algorithm, as described in Aho et al [1974], applied to an arbitrary closed semiring, is a standard algorithm

3 The improvement of Warshall's algorithm due to Warren [1975] is also a standard algorithm

4 *Seminaive evaluation*[3] of the logical rules

```
path(V,W) .- arc(V,W)
path(V,W)  - path(V,X) & path(X,W)      (1 2)
```

Here, we use Prolog notation, and the two rules are read "there is a path from $V$ to $W$ if there is an arc from $V$ to $W$," and "there is a path from $V$ to $W$ if there exists a node $X$ such that there is a path from $V$ to $X$ and a path from $X$ to $W$" Seminaive evaluation applies the rules repeatedly, starting from the data (the arc facts), and getting successive approximations to the path facts In seminaive (as opposed to *naive*) evaluation, care is taken that when applying a rule, at least one of the facts in the body (right side) is "new," in the sense that it was just discovered on the previous round Seminaive evaluation might not look at facts for every triangle, for example, it would not if the set of arcs were empty However, if the set of arcs is a complete graph, then on the second round, the recursive rule finds all path facts "new," and attempts to apply the recursive rule in all possible ways Then, all triangles $\{V, X, W\}$ of path facts will appear simultaneously in memory, when the recursive rule is applied to that triple of nodes (in any of the six orders)

5 Similarly, we might use either of the two linear forms of the recursion for transitive closure Seminaive evaluation of the rules

```
path(V,W)  - arc(V,W)
path(V,W)  - arc(V,X) & path(X,W)       (1 3)
```

or the rules

```
path(V,W)  - arc(V,W)
path(V,W)  - path(V,X) & arc(X,W)       (1 4)
```

are standard algorithms Here, the triangles each involve two path facts and an arc fact, but there is nothing in the definition of "standard" that requires the facts all be path facts

Are there algorithms in use that are not "standard" in our sense? Yes, there are several different approaches that do not use triangles of nodes

1 Algorithms for transitive closure can be based on a fast matrix-multiplication algorithm For example, if we square the adjacency matrix $\log n$ times, using the algorithm of Coppersmith and Winograd [1987] to square matrices, we get an $O(n^{2.37} \log n)$ time algorithm, that uses I/O equal to $O((n^{2.37}/s^{0.19}) \log n)$

2 Algorithms can be based on depth-first search As we shall see, for graphs with cycles, the discovery of strongly connected components allows us to infer the existence of paths without actually constructing them

There is also a subtlety regarding algorithms that do not behave in an "oblivious" way, but base their action on the values computed For example, we could modify Warshall's algorithm in two ways First, if we find $path(i, k)$ AND $path(k, j)$ to be false, then we need not modify $path(i, j)$ Hence, it appears we don't need to have the triangle $\{i\ j\ k\}$ in memory at one time However, this modified version of Warshall's algorithm is still standard, because when the given graph is complete, it must assign to $path(i, j)$ all the time

Another modification to Warshall's algorithm is to write a 1 into the copy of $path(i, j)$ in secondary memory whenever $path(i, k)$ AND $path(k, j)$ is true, regardless of its current value Again, it appears we do not need $path(i, j)$ in main memory when we consider the triple $(i, j, k)$ However, at the expense of doubling the I/O and increasing the memory by half, we could insist that $path(i, j)$ be brought into main memory anyway, and then written out Since our results are all order-of-magnitude anyway, this modification cannot be better than a standard algorithm Thus, any lower bound for standard algorithms applies to modifications that write new values for variables into secondary storage without examining their current value first Moreover, if we are performing a generalized transitive closure, then the analog of Warshall's algorithm (Kleene's algorithm) cannot write a new value for $path(i, j)$ without having the old value in main memory Actually, all algorithms

[3] See Ullman [1988] for a discussion of seminaive evaluation

that we know of for the generalized transitive closure problem are standard

### Extension to Block-Oriented I/O

Usually, we don't want to count only the volume of traffic between main and secondary memory, but rather count the number of blocks or pages that must move between the two memories As in the model of Ullman [1988], blocks carry some fixed, but large number of facts It turns out that, under reasonable assumptions about the size of blocks, each of the algorithms we propose allows complete blocking, that is, if $b$ facts fit on one block, then the amount of I/O can be reduced by a factor $b$ Since the lower bounds cannot be reduced by a greater factor, our results will carry over to the model with blocked I/O, simply by dividing expressions by $b$

## II AN ALGORITHM FOR THE DENSE CASE

We begin with some relatively easy results, these cover the dense case, where the number of arcs in an $n$-node graph can be as high as $n^2$ A recent paper by Agrawal and Jagadish [1987] offers an algorithm for this case By partitioning the adjacency matrix of the graph into stripes, they use I/O equal to $O(n^4/s)$ As usual, $n$ is the number of nodes and $s$ the amount of main memory McKellar and Coffman [1969] studied the problem of performing matrix operations such as transposition, or matrix multiplication in a paged memory system They compared various matrix partitioning schemes, and found that storing and manipulating matrices by stripes (of rows or columns) is inferior to a scheme that partitions the matrices into square submatrices of appropriate size As we shall see, the same fact is true of the transitive closure problem

Divide the nodes of the graph into $n/\sqrt{s}$ *zones* as follows Nodes 1 through $\sqrt{s}$ are in zone 1, nodes $\sqrt{s}+1$ through $2\sqrt{s}$ are in zone 2, and so on This partitioning of the nodes corresponds to a partitioning of the adjacency matrix $M$ of the graph into squares of size $\sqrt{s}$, there being $n^2/s$ submatrices, which we designate $M_{i,j}$, for $1 \leq i,j \leq n/\sqrt{s}$ Figure 2 1 illustrates this partition Thus, the submatrix $M_{i,j}$ tells about arcs from nodes in zone $i$ to nodes in zone $j$

If $M$ is a Boolean matrix, or in fact, a matrix whose elements come from any closed semiring, then the $\sqrt{s} \times \sqrt{s}$ submatrices of $M$ are elements of a closed semiring, with the usual matrix multiplication and addition as $\times$ and $+$ in the closed semiring (see Exercise 15 28 of Ullman [1989] for some details) Thus, we can apply Kleene's algorithm, as shown in Fig 2 2, to compute the transitive closure of $M$ There, note that $M^*_{k,k}$ is the

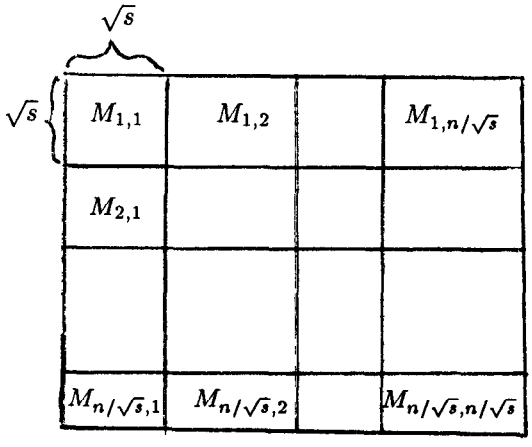


**Fig 2.1.** Partition of a matrix

reflexive and transitive closure of $M_{k,k}$ An induction on $k$ shows that, after executing lines (2) through (5) with a fixed value of $k$, the value in row $v$ and column $w$ is 1 if and only if there is a path from node $v$ to node $w$ that goes through no zone higher than $k$ (Nodes $v$ and/or $w$ may themselves be in higher zones ) We omit the details of the proof, from which we have the following theorem

```
(1) for k = 1 to n/√s do begin
(2)     M_{k,k} = M*_{k,k},
(3)     for i = 1 to n/√s do
(4)         for j = 1 to n/√s do
(5)             M_{i,j} = M_{i,j} + M_{i,k} × M_{k,k} × M_{k,j}
    end
```

**Fig. 2.2.** Kleene's algorithm

**Theorem 2.3:** The algorithm of Fig 2 2 correctly computes the transitive closure of the graph with adjacency matrix $M$ □

**Theorem 2.4:** The algorithm of Fig 2 2 requires I/O equal to $O(n^3/\sqrt{s})$, provided $s \leq n^2$

**Proof:** Note that any submatrix $M_{i,j}$ will fit in main memory at once Since we assume we have $O(s)$ memory, we can hold any finite number of submatrices, in particular, $M_{i,j}$, $M_{k,k}$, and $M_{k,j}$ at once When we read $M_{k,k}$ at line (2), we can take its transitive closure by Warshall's algorithm That algorithm is executed "in place," and needs no additional main memory To execute line (5), we surely need no more than $O(s)$ I/O, since we read and write a finite number of $O(s)$-sized submatrices

Step (2) requires only $O(n\sqrt{s})$ I/O, since it is executed $n/\sqrt{s}$ times and uses $O(s)$ I/O each time Step (5) dominates, since it is executed $n^3/s^{3/2}$ times, with $O(s)$ I/O each time The I/O cost is thus $O(n^3/\sqrt{s})$ □

Note that $n^3/\sqrt{s}$ is always less than the Agrawal and Jagadish [1987] upper bound of $n^4/s$, as long as $s < n^2$, furthermore, the constants of the two algorithms are comparable When $s \geq n^2$, the upper bound on I/O is $O(n^2)$, since that is required just to read the data Once read, we can compute the transitive closure in main memory with no further I/O, except the $n^2$ steps needed to write the answer

**Corollary 2.5:** If for some closed semiring, the domain elements require $O(1)$ memory space, and the $+$, $\times$, and * operations can be performed with at most $O(s)$ scratch space, then the generalized transitive closure can be taken for this closed semiring with $O(n^3/\sqrt{s})$ I/O □

## III LOWER BOUND FOR DENSE GRAPHS

We shall now show that the upper bound $n^3/\sqrt{s}$ for dense graphs is also a lower bound on the I/O needed by any standard algorithm As we mentioned, it is possible to beat the upper bound, in principle, by using a fast matrix-multiplication algorithm, although it is unclear whether these schemes are practical for problems of realistic size The following combinatorial lemma is the key

**Lemma 3.1:** With $m$ edges, we can make no more than $O(m^{3/2})$ triangles □

Now we can prove the lower bound result for dense graphs The argument is similar to the one found in Kung and Hong [1980] for the standard matrix multiplication algorithm

**Theorem 3.2:** Every standard algorithm for computing the transitive closure of a dense graph with $n$ nodes, using main memory of size $s$ requires $\Omega(n^3/\sqrt{s})$ I/O in the worst case □

## IV. AN UPPER BOUND FOR SPARSE, ACYCLIC GRAPHS

We shall now introduce $e$, the number of arcs, as a separate parameter It is useful to study the case of acyclic graphs first, because they appear to be able to take advantage of sparseness in a way that cyclic graphs cannot In what follows, we shall assume that $e \geq n$, and $s \ll e$,[4] since if $e$ main memory space is available, we can read the entire graph, and then compute its transitive closure by searching from each node, in main memory Then, $n^2$ I/O suffices to write the answer

A standard algorithm for sparse graphs must combine arc facts with path facts, or it cannot take advantage of sparseness The reason is that $O(n)$ arcs can yield paths between any pair of nodes Thus, a standard algorithm that combines paths (probably) will have to consider all triangles, even if $e$ is much less than $n^2$ Thus, it is reasonable to consider algorithms that only combine arc and path facts, such as seminaive evaluation of the linear rules (1 3) or (1 4) or the basic algorithm of Ioannidis and Ramakrishnan [1988] for acyclic graphs

### Sparse-Standard Algorithms

We shall call an algorithm *sparse-standard* if it has in memory at some time every triangle $(v, x, w)$ consisting of an arc fact $arc(v, x)$, and path facts $path(v, w)$ and $path(x, w)$ We call such a triangle an *app-triangle* Note that there are $e(n-2)$, or about $en$ app-triangles, since each arc can have any of the $n-2$ nodes that are not its endpoints as the opposite vertex

It turns out that there is a sparse-standard algorithm to compute transitive closure of a sparse, acyclic graph in $O(n^2\sqrt{e/s})$ time [5] Moreover, any sparse-standard algorithm must take at least this amount of time, even on acyclic graphs Interestingly, the upper bound also holds for sparse graphs with cycles, but the algorithm is not even standard

### The "Big" and "Small" Parameters

In what follows, it is useful to define the shorthands

$$\beta = n\sqrt{\frac{s}{e}}$$

$$\sigma = \frac{\sqrt{es}}{n}$$

We think of $\beta$ as the size of a "big" set of nodes and $\sigma$ as the size of a "small" set Notice that in the dense case, where $e = n^2$, we have $\beta = \sigma = \sqrt{s}$

The intuition regarding $\beta$ and $\sigma$ is as follows In Lemma 3 1, we observed that $s$ edges make at most $s^{3/2}$ triangles To attain that ratio of triangles to edges, we must select about $\sqrt{s}$ nodes and all edges between these nodes Now, suppose we are only interested in app-triangles If the graph is random, that is, each arc

[4] That is, $s$ grows asymptotically more slowly, as a function of $n$, than $e$

[5] Note that this formula reduces to the formula for dense graphs when $e = n^2$

is chosen with probability $e/n^2$, then, approximately, the greatest number of app-triangles will be obtained if we have two "big" sets of $\beta$ nodes each, and a "small" set of size $\sigma$ For our $s$ facts, we select all the arcs of the given graph that run from one big set to the other, and we also select path facts running from each node in the small set to each node in either of the big sets The expected number of arcs from one big set to another is $\beta^2 e/n^2 = s$, and the number of pairs of nodes, one from a big set and one from a small set, is $\beta\sigma = s$ Thus, there are an expected $O(s)$ facts selected, and by constant-factor adjustments in the size of the big and small sets, we can make that be exactly $s$

Like the algorithm of Section II, we shall "pivot" on sets of nodes However, instead of using "zones" of size $\sqrt{s}$, when we pivot on zone $k$, and establish paths from zone $i$ to zone $j$ that go through zone $k$, zones $i$ and $k$ will be big sets, and zone $j$ will be a small set Moreover, we use only arcs from zone $i$ to zone $k$, not paths That is why the algorithm only works for acyclic graphs, and then only if the pivot zones are chosen in reverse topological order

The algorithm is best explained as doing something more general than computing transitive closure, which we shall call *extended transitive closure* The input to the algorithm is as follows We are given $m$ nodes that are topologically sorted and numbered $1, 2, \ldots, m$ We are also given $e$ arcs among the first $n \leq m$ nodes and any number of paths from nodes among the first $n$ to any of the $m$ nodes (including the first $n$) All arcs and paths go from lower- to higher-numbered nodes, as suggested by Fig 4 1 We have an amount of main memory $s \geq n$ The problem is to find which nodes among the first $n$ have paths to which of the $m$ nodes, each such path will follow zero or more arcs among the first $n$, and then an (optional) path fact The case of ordinary sparse transitive closure is extended transitive closure with $m = n$, and no given paths

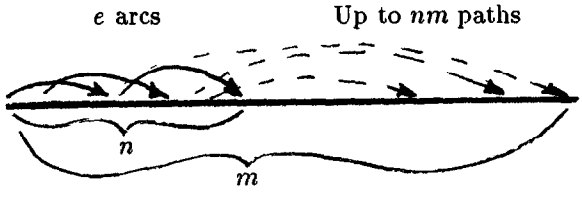


**Fig. 4.1.** Given data for Algorithm 4 2

We now give an algorithm that computes the extended transitive closure with I/O equal to $O(mn(1 + \sqrt{e/s}))$

**Algorithm 4.2:** Sparse-Standard Algorithm for Extended Transitive Closure of Acyclic Graphs

INPUT As above

OUTPUT The set of pairs of nodes $(i, j)$ such that there is a path from $i$ to $j$ consisting of zero or more given arcs followed by a given path

METHOD If $e = O(s)$, then read all the given edges into main memory For each of the $m$ nodes, say node $v$, do the following

1 Read all given paths from one of the first $n$ nodes to $v$ into main memory There are at most $n$ such paths, so they fit, since $s \geq n$ is assumed

2 In main memory, find all the nodes among the first $n$ that can reach $v$ following arcs and then a path directly to $v$ No I/O is needed

3 Write the up-to-$n$ path facts telling which of the first $n$ nodes reach $v$ along an allowable path

Evidently, $O(n)$ I/O per node is needed in steps (1) and (3), so the total I/O is $mn$ That cannot exceed our predicted upper bound of $mn(1+\sqrt{e/s})$ Note also that the $e$ I/O steps needed to bring in the $e$ given arcs cannot exceed $mn$, since $m \geq n$, and $e \leq n^2$

Now, let us consider the case where $e$ grows more rapidly than $s$ Define $\beta = n\sqrt{s/e}$ and $\sigma = \sqrt{es}/n$, as before Note that $\beta$ grows more slowly than $n$, since $s$ grows more slowly than $e$ We shall partition the first $n$ nodes into $n/\beta$ "big zones" of size $\beta$ each All nodes are partitioned into "small zones" of size $\sigma$, there are $m/\sigma$ such zones In each case, the zones consist of consecutive nodes, as suggested in Fig 4 3 In that figure, we have made the tacit assumption that $n/\beta = 3$, and $\beta = 2\sigma$ Technically, we should see Fig 4 3 as two Boolean matrices One, of size $n \times m$, represents path information the second, of size $n \times n$, and located at the left end, represents arc information

We execute the algorithm sketched in Fig 4 4 This algorithm is similar to the pivoting algorithms seen so far Here we pivot on a square of the matrix whose side is $\beta$, that is, we use a set of $\beta$ consecutive nodes as the "pivot" In order to take what serves as the transitive closure of a pivot block, we must call Algorithm 4 2 recursively In the recursive call, the pivot block plays the roll of the $n \times n$ matrix at the left, as in Fig 4 3, while that block and everything to its left plays the role of the entire matrix in Fig 4 3

Note that, as we assume $s \ll e$, there are only a finite number of $n$'s for which $n < \beta + 1$ Therefore, the basis case at line (2) applies only to problems of limited size Thus, all the arcs can be stored in $O(1)$ memory,

```
(1) if n is sufficiently small that n < β + 1 then
(2)     compute the extended transitive closure in any manner
    else
(3)     for k = n/β downto 1 do begin
(4)         apply Algorithm 4 2 to the nodes of big zone k (playing the
              role of the first n nodes) and all higher nodes (playing
              the role of the remaining m-n nodes),
(5)         for i := 1 to k-1 do
(6)             for j = 1 to m/σ do
(7)                 for each node v in big zone i, each node x in big
                      zone k, and each node w in small zone j do
(8)                     path(v,w) =
                          path(v,w) OR (arc(v,x) AND path(x,w))
        end
```

**Fig. 4.4.** Algorithm 4 2 for the case $s \ll e$

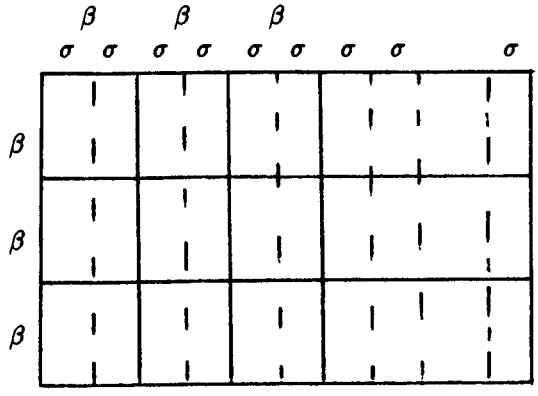


**Fig. 4.3.** Partitions into big and small zones

and can be kept in main memory no matter what $s$ is As a result, an $O(mn)$ I/O algorithm for line (2) is easy

One last detail concerns the loop of line (5) If there are $O(s)$ arcs from big zone $i$ to big zone $k$, then for each $j$ we can execute the loop of lines (6) to (8) with $O(s)$ I/O As we mentioned, in the average case, there are no more than $s$ arcs between two big zones However, sometimes there will be more, and if so, we must break up zone $i$ into sets of nodes that each have $O(s)$ arcs to zone $k$ Since $n \leq s$ is assumed, we can always make the partition When we perform the analysis of the I/O in Theorem 4 5, we shall see that the effect of big zones with more than $s$ arcs to the pivot zone can be neglected □

**Theorem 4.5.** Algorithm 4 2 correctly computes the extended transitive closure of an acyclic graph, and requires I/O equal to $O(mn(1 + \sqrt{e/s}))$ □

Now, we can apply Theorem 4 5 to the case we are really interested in, when $m = n$, $n \leq s \leq e \leq n^2$, and there are no given paths

**Corollary 4.6:** If $n \leq s \leq e \leq n^2$, Algorithm 4 2 computes the transitive closure of an $n$-node, $e$-arc acyclic graph using $s$ main memory space with I/O equal to $O(n^2\sqrt{e/s})$ □

## V. A LOWER BOUND FOR SPARSE-STANDARD ALGORITHMS

We shall now show that every sparse-standard algorithm for computing transitive closure, even on acyclic graphs, requires $\Omega(n^2\sqrt{e/s})$ I/O The key is the following combinatorial lemma

**Lemma 5.1:** For any $\sigma = \sqrt{es}/n$ there is a constant $\alpha$ with the property that no matter what $n$, $e$, and $s$ are, provided that $n$ is sufficiently large, that $n \leq e \leq \binom{n}{2}$, and that $\sigma$ is $\Omega(\log n)$, there is an acyclic graph of $n$ nodes and $e$ arcs, such that no set of $i$ nodes, where $\sigma \leq i \leq \beta$, has more than $\alpha\sigma i$ arcs in its generated subgraph □

Now, we can prove an analog of Lemma 3 1

**Lemma 5.2:** Under the same conditions as Lemma 5 1, there is an acyclic graph $G$ such that any set of $s$ arc and path facts forms no more than $O(\sigma s)$ app-triangles with respect to the arcs of $G$ □

**Theorem 5 3·** If $n \leq s \leq e \leq n^2$, and $\sigma = \Omega(\log n)$, then any sparse-standard algorithm to compute the transitive closure requires I/O at least $\Omega(n^2\sqrt{e/s})$ □

We should observe that there is a gap in our knowledge, in that we don't know whether our lemmas, or Theorem 5 3, are true when $\sigma \ll \log n$ That condition can only occur when $e$ and $s$ are both very close to $n$, however Note that when $e$ and $s$ are both $O(n)$, then Theorem 5 3 does hold, just because $\Omega(n^2)$ I/O is necessary to write the answer Also, even if $\sigma \ll \log n$, we cannot improve on the upper bound by more than an $O(\log n)$ factor

## VI AN ALGORITHM FOR GENERAL SPARSE GRAPHS

With very little I/O, we can reduce the general case to the acyclic case We find strong components, by the depth-first search technique of Hopcroft and Tarjan [1973] As we shall see, $O(e)$ I/O suffices Once we have strong components, we can find the intercomponent arcs, which form an acyclic graph, and apply Algorithm 4 2 to it Finally, we translate the reachability information about strong components into information about the nodes in those components Notice that this algorithm is not sparse-standard, in fact it is not even standard, because we infer the existence of certain paths by the fact that two nodes are in the same strong component, even though we may not have constructed a path from one node to the other As a consequence, this algorithm will not generalize to arbitrary closed semirings

### Space-Efficient Depth-First Search

We assume that the graph is stored in secondary memory by the successor lists of the nodes Suppose that we have main memory $s$ at least $n$, so we can build in main memory tables indexed by the nodes If we could read into main memory just one edge at a time, then a straightforward implementation of the standard depth-first algorithm would achieve $O(e)$ I/O However, if we have to read a block of edges each time, then this may not be true any more for example, if the search follows a long path so that many nodes are open, then we may not be able to keep their arcs in main memory, and therefore we will have to read the arcs for the same node more than once Ioannides and Ramakrishnan [1988] analyze the standard depth-first search algorithm under a model where the unit of transfer between secondary and main memory is the successor list of a node We will show here how to implement depth-first search so that the successor list of each node is read only once, regardless of the density of the graph

The "trick" is to keep information regarding all the successors of a node in main memory, even if they have not yet been placed in the depth-first spanning tree, but prune the successor lists selectively as the algorithm proceeds, to fit in main memory The pruning has the property that the second and subsequent times we need to find a successor of a given node $v$, those of its successors that have not already been placed in the tree remain in a linked list, and we simply take the first, with no I/O needed, and place it in the tree as a child of $v$ The main-memory data structure we need is an array, indexed by nodes The elements of the array are structures capable of holding the following

1 The parent of the node in the tree being formed
2 Leftmost-child and right-sibling pointers, so we can construct a list, for each node, of its children in the tree, and its other successors that have not yet been placed in the tree In practice, this list should be doubly-linked
3 A bit telling whether the node is in the tree
4 The depth-first number of the node (the order in which the depth-first search retreated from each node)

**Algorithm 6.1:** Depth-First Search with Minimal I/O

INPUT A directed graph of $n$ nodes and $e$ arcs

OUTPUT A depth-first spanning forest for the graph, and a depth-first ordering for the nodes

METHOD Start with any node, which becomes the root of the current tree and also becomes the current node In what follows we can reach the current node by "advancing," which is the first time we visit the node, or by "retreating" Each time we advance to a new current node, which includes the case where we have just made the current node the root of its tree, we do the following

Bring the successors of the current node into main memory Each successor is examined, in turn If it is already in the tree, it is deleted from the list If not already in the tree, it is added to the list of children of the current node If it is on the list of children for some other node (but not yet placed in the tree), we remove it from the list of children for that other node Thus, no node is ever in the list of children for two different nodes None of the children of the current node are placed in the tree, if they were not already in the tree

Now, we move to the first node on the list of children of the current node that is not yet in the tree If there are none, we retreat from the current node, and make its parent the current node Before we do, we give the current node its depth-first number, which is one more than the previously-awarded depth-first num-

ber

If we find a child of the current node that is not yet in the tree, then we place that node in the tree and make it the current node Whether the new current node was determined by retreating or by advancing to a child of the current node, we repeat the above loop, with the new current node

If we retreat from the root of the current tree, then we are done However, there may be other nodes not yet inserted into any tree, because they were not reachable from any root selected so far If so, select one as a new root, and repeat the entire process above □

**Theorem 6.2:** Algorithm 6 1 constructs a depth-first spanning forest It reads each successor list once and uses I/O equal to $O(e)$, provided only that $e$ and $s$ are at least $n$ □

**Corollary 6 3:** With $O(e)$ I/O, we can find a topological order of an acyclic graph, provided only that $e$ and $s$ are each at least $n$ □

**Theorem 6.4:** With $O(e)$ I/O, we can partition nodes into strong components, provided $e$ and $s$ are at least $n$ □

### The General Algorithm

We can now describe an algorithm for computing the transitive closure of general graphs

**Algorithm 6 5:** Transitive Closure for Sparse, Cyclic Graphs

INPUT A directed graph of $n$ nodes and $e$ arcs, and an amount, $s$, of main memory We assume $n \leq s \leq e \leq n^2$

OUTPUT The transitive closure of the given graph

METHOD Perform the following steps

1 Find strong components of the graph by the algorithm described in Theorem 6 4
2 Storing the strong component for each node in main memory, examine the arcs of the graph to create a list of arcs that run between two strong components, these become the arcs of an acyclic graph for the next step
3 Perform Algorithm 4 2 on the acyclic graph from (2)
4 For each pair of strong components $a$ and $b$ such that it is determined in step (3) that there is an arc from $a$ to $b$, output the fact that there is an arc from each node in $a$ to each node in $b$ □

**Theorem 6 6:** Algorithm 6 5 correctly determines the transitive closure and uses I/O equal to $O(n^2\sqrt{e/s})$ □

## VII SPARSE-STANDARD ALGORITHMS FOR CYCLIC GRAPHS

We do not know whether one can solve the *generalized* transitive closure problem on cyclic graphs taking advantage of sparsity as Algorithm 6 5 does For sparse-standard algorithms it seems that another factor of $n$ in I/O is needed in the cyclic case First, observe that we can run Algorithm 4 2 $n$ times,[6] and each time, we make progress of at least one arc along any path whatsoever, even if the next arc on the path goes from a high-numbered node to a lower-numbered node Thus, the following is immediate

**Theorem 7.1:** There is a sparse-standard algorithm to compute the transitive closure of a general graph with I/O equal to $O(n^3\sqrt{e/s})$, under our usual assumptions that $n \leq s \leq e \leq n^2$ □

### Algorithms That Follow All Paths

It is not known whether Theorem 7 1 can be beaten by an algorithm that only combines arc facts with path facts (rather than combining two path facts, as the algorithm of Fig 2 2 does) However, if we add the additional restriction that the algorithm must follow each acyclic path, then we can show that the bound of Theorem 7 1 is tight Let us define an *all-paths standard algorithm* to be one in which, for each acyclic path $v_1 \rightarrow v_2 \rightarrow \quad \rightarrow v_k$, there is a sequence of times $t_2, \quad, t_k$, with $t_2 \leq t_3 \leq \quad \leq t_k$, such that at time $t_i$, the facts $arc(v_i, v_{i+1})$, $path(v_1, v_i)$, and $path(v_1, v_{i+1})$ are in memory simultaneously That is, the sequence of triangles, each of which has $v_1$ as a vertex and has, as opposite side, one of the arcs along the path (other than the first), must appear in main memory in the same order as the arcs appear on the path

The requirement that all paths be followed is very stringent For example, when computing the ordinary transitive closure, we only require that at least one path from node $v$ to node $w$ be discovered, not that all acyclic paths be discovered An algorithm like seminaive evaluation will not normally follow all paths either, since it stops when no new path facts are discovered on a round On the other hand, there are closed semirings where it appears all acyclic paths must be followed, if we are constrained only to combine arc and path facts, never two path facts

**Theorem 7.2** If $s \ll n^2$, then the minimum amount of I/O needed for an all-paths standard algorithm on dense graphs of $n$ nodes with main memory of size $s$ is

[6] But in line (5), make $i$ range from 1 to $n/\beta$, rather than 1 to $k-1$

| Graph Class | Algorithm Class | Lower Bound | Upper Bound | Restrictions for Lower Bound | Upper Bound Generalizes? |
|---|---|---|---|---|---|
| Dense | Standard | $n^3/\sqrt{s}$ | $n^3/\sqrt{s}$ | – | Yes |
| Sparse, Acyclic | Sparse-Standard | $n^2\sqrt{e/s}$ | $n^2\sqrt{e/s}$ | $\sigma = \Omega(\log n)$ | Yes |
| Sparse, Cyclic | Any | ? | $n^2\sqrt{e/s}$ | – | ? |
| Dense | All-Paths Standard | $n^4/\sqrt{s}$ | $n^4/\sqrt{s}$ | – | Yes |
| Sparse, Cyclic | All-Paths Standard | $n^3\sqrt{e/s}/\log^3 n$ | $n^3\sqrt{e/s}$ | $\sigma = \Omega(\log n)$ $s \ll e$ | Yes |

**Fig. 8.1.** Summary of upper and lower bounds

$\Omega(n^4/\sqrt{s})$ □

In the general sparse case we can show

**Theorem 7.3:** If $n \leq s \ll e$, any all-paths standard algorithm requires I/O at least $\Omega(n^3\sqrt{e/s}/\log^3 n)$ □

## VIII. SUMMARY

We summarize the results of this paper in the table of Fig 8 1 For all these results, the assumption $n \leq s \leq e \leq n^2$ applies (For acyclic graphs, we assume $e \leq \binom{n}{2}$ ) In addition, for the lower bounds we assume $s \ll e$, and there may be a further assumption needed in the sparse cases All functions are to within an order of magnitude The last column indicates whether the algorithm for the upper bound generalizes to arbitrary closed semirings, or in the case of the all-paths standard algorithms, to all closed semirings in which following acyclic paths is sufficient to get the correct answer

## REFERENCES


Agrawal, A and H V Jagadish [1987] "Direct algorithms for computing the transitive closure of database relations," *Proc International Conference on Very Large Data Bases*, pp 255–266

Aho, A V , J E Hopcroft, and J D Ullman [1974] *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading Mass

Benes, V E [1965] *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York

Coppersmith, D and S Winograd [1987] "Matrix multiplication via arithmetic progressions," *Proc Nineteenth Annual ACM Symposium on the Theory of Computing*, pp 1–6

Hong, J -W and H -T Kung [1980] "The red-blue pebble game," *Proc Thirteenth Annual ACM Symposium on the Theory of Computing*, pp 326–333

Hopcroft, J E and R E Tarjan [1973] "Efficient algorithms for graph manipulation," *Comm ACM* **16** 6, pp 372–378

Ioannidis, Y E and R Ramakrishnan [1988] "Efficient transitive closure algorithms," CSTR–765, Dept of CS, Univ of Wisconsin

Lovasz, G L [1979] *Combinatorial Problems and Exercises*, Noth-Holland, Amsterdam

McKellar, A C and E G Coffman [1969] "Organizing matrices and matrix operations for paged memory systems," *Comm ACM* **12** 3, pp 153–165

Ullman, J D [1988] *Principles of Database and Knowledge-Base Systems*, Vol I *Classical Database Systems*, Computer Science Press, New York

Ullman, J D [1989] *Principles of Database and Knowledge-Base Systems*, Vol II *The New Technologies*, Computer Science Press, New York

Warren, H S [1975] "A modification of Warshall's algorithm for the transitive closure of binary relations," *Comm ACM* **18** 4, pp 218–220

Warshall, S [1962] "A theorem on Boolean matrices," *J ACM* **9** 1, pp 11–12