HIGH-PROBABILITY PARALLEL TRANSITIVE CLOSURE ALGORITHMS

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

ABSTRACT: There is a straightforward algorithm for computing the transitive closure of an n-node directed graph in $O(\log^2 n)$ time on an EREW-PRAM, using $n^3/\log n$ processors, or indeed with $M(n)/\log n$ processors if one can do serial matrix multiplication in time $M(n)$. This algorithm is within a log factor of optimal in *work* (processor-time product), for solving the all-pairs transitive-closure problem for dense graphs. However, this algorithm is far from optimal when either (a) the graph is sparse, or (b) we want to solve the single-source transitive closure problem. Ideally, we would like an $\mathcal{NC}$ algorithm for transitive closure that took about e processors for the single-source problem on a graph with n nodes and $e \geq n$ arcs, or about en processors for the all-pairs problem on the same graph. While we cannot offer an algorithm that good, we can offer algorithms with the following performance. (1) For single-source, $\tilde{O}(n^\epsilon)$ time with $\tilde{O}(en^{1-2\epsilon})$ processors,[1] provided $e \geq n^{2-3\epsilon}$, and (2) for all-pairs, $\tilde{O}(n^\epsilon)$ time and $\tilde{O}(en^{1-\epsilon})$ processors, provided $e \geq n^{2-2\epsilon}$. Each of these claims assumes $0 < \epsilon \leq 1/2$. Importantly, the algorithms are (only) *high-probability* algorithms; that is, if they find a path, then a path exists, but they may fail to find a path that exists with probability at most $2^{-\alpha c}$, where α is some positive constant, and c is a multiplier for the time taken by the algorithm. However, we show that incorrect results can be detected, thus putting the algorithm in the "Las Vegas" class. Finally, we show how to do "breadth-first-search" with the same performance as we are able to achieve for single-source transitive closure.

† Work partially supported by ONR contract N00014-88-K-0166 and a Guggenheim fellowship.

[1] $\tilde{O}$ is the notation, proposed by Luks and furthered by A. Blum for "within some number of log factors of." That is, we say $f(n)$ is $\tilde{O}(g(n))$ if for some constants c and k, and all sufficiently large n, we have $f(n) \leq c\log^k n g(n)$.

1. INTRODUCTION

As mentioned in the abstract, we address the apparently difficult problem of doing parallel transitive closure when the (directed) graph is sparse and/or, only single-source information is desired. We want to use less-than-linear time, and use *work*, the product of time and number of processors, that approximates the time of the best serial algorithm. An algorithm whose work is of the same order as the best known serial time is referred to as *optimal*.

For the single-source transitive-closure problem, depth-first search (see Aho, Hopcroft, and Ullman [1974], e.g.) takes $O(e)$ time on a graph of e arcs.[2] Thus, $O(e)$ work is our target for the single-source problem. When the graph is sparse,[3] then the all-pairs transitive closure problem can be solved by performing a depth-first search from each node, taking $O(ne)$ time; that is our target for the all-pairs problem. As seen from the abstract, we do not reach either target, except for the all-pairs case when e is fairly large. However, we make significant progress, in the sense that we have the first algorithms that simultaneously use time much less than linear and use work that is less than $M(n)$.

High-Probability Algorithms

Unless otherwise stated, all algorithms described in this paper are *high probability* algorithms, meaning that

1. If they report a path from one node to another, then there truly is such a path.

[2] Throughout, we assume that n is the number of nodes, e the number of arcs, and that $n \leq e$.

[3] "Sparse" normally means that $e \ll n^2$, although if one believes that the various "fast" matrix multiplication algorithms (see Coppersmith and Winograd [1987], e.g.) taking $M(n)$ time where $M(n)$ is some power of n between 2 and 3 are practical, then "sparse" should be taken to mean $e \ll (M(n)\log n)/n$.

2. If such a path exists, then the probability that the algorithm fails to report its existence is at most some $p_0 < 1$. By doubling the time taken by the algorithm, we can square the probability of an error. Thus, a linear-factor increase in the running time of the algorithm yields an exponential decrease in the probability of error.

Moreover, we shall show that it is possible to check our result for validity with little additional work. Thus, the algorithms can be run in a "Las Vegas" mode, where termination only occurs when the correct answer has been obtained. In that case, the times we quote should be interpreted as expected times for termination, and the probability of the actual time being greater than that by a factor c decreases exponentially with c.

$\tilde{O}$ and $\tilde{\Omega}$ Notation

The algorithms we discuss introduce factors of $\log n$ from several sources. To avoid cluttering the expressions we use, these factors are elided. All our claims involve factors of at least n^ϵ for some $\epsilon > 0$, so logarithmic factors do not dominate. As mentioned, we use the notation $\tilde{O}$ to subsume factors of $\log n$, just as the conventional "big-oh" subsumes constant factors. Similarly, when dealing with lower bounds, we use $\tilde{\Omega}$ as a version of "big-omega" that subsumes factors of $\log n$.

Previous Work

Solutions for several related problems are known. Shiloach and Vishkin [1982] gives the best known deterministic algorithm for connected components, which is essentially transitive closure on an undirected graph. Gazit [1986] gives an optimal randomized algorithm for the same problem.

Gazit and Miller [1988] offers an $\mathcal{NC}$ algorithm for "breadth-first search," which is really computing the distance, measured in number of arcs, from a given node. That problem generalizes single-source transitive closure, but they do not get below the $M(n)$-work barrier.

Several algorithms for transitive closure on "random" graphs, that is, on the population of graphs constructed by picking each arc with a fixed probability, have been given; see Bloniarz, Fischer, and Meyer [1976], Schnorr [1978], and Simon [1986]. These offer expected time close to n^2, but their worst-case performance is as bad as can be — $O(n^3)$ or $O(ne)$, as appropriate. It should be emphasized that our performance measure is independent of the actual graph to which it is applied.

We should also note the paper by Broder, Karlin, Raghavan, and Upfal [1989], which deals with connectivity in undirected graphs. They, like us, use a technique of guessing a subset of the nodes and doing a search from each, hoping to connect all the guessed nodes that are found along a given path. Their search technique is random walks, while we use exhaustive, deterministic exploration for a limited distance. It is easy to show that random walks will not serve when directed graphs are concerned, intuitively because it is easy to "trap" a random walk on a directed graph.

Organization of the Paper

In Section 2 we introduce and analyse our algorithm for solving the single-source problem in $\tilde{O}(\sqrt{n})$ time and $\tilde{O}(e)$ processors. This algorithm contains most of the ideas needed for the general case, but their use in the general case is much more complex. In Section 3 we introduce the general problem of solving transitive closure for a graph of n nodes and e arcs, with s sources and paths of distance up to d allowed. Four reduction strategies that work with high-probability are proposed.

In Section 4 we give a general algorithm and show that it is the best that can be achieved using the four reductions of Section 3. Section 5 gives some simplifications of the general algorithm of Section 4 for interesting special cases: dense graphs, single-source, and all-pairs problems. Finally, in Section 6, we show how to extend the ideas to solve the breadth-first-search problem for a single source in the same time as we can do single-source reachability. For space reasons, all the proofs and many details are omitted from this extended abstract.

2. A SIMPLE ALGORITHM

Exploration of a directed graph from a single source node appears to be an inherently sequential process, especially in a case where the graph is something like a single line emanating from the source. If we are to explore a path of length $\Omega(n)$ in less than time n, we must explore from many of the nodes along the path before we even know that they are on the path. However, carried to extremes, we search from every node in parallel, and we arrive at the standard path-doubling method of solving the all-pairs problem in $\mathcal{NC}$ by doing $\log n$ Boolean matrix multiplications.

What we would like to do is to search from some of the nodes on the path forward for a short distance, until the searches link up; that is, we explore from each of the selected nodes at least as far as the next selected node, as suggested by Fig. 2.1. The searches can be done in parallel, and if nodes on the path are not too far apart, we can discover any path from the source to any

node without computing the entire all-pairs transitive closure. We need to compute the nodes reached from only a subset of the nodes, and we need to know only about the nodes reached from this subset along paths of limited length.

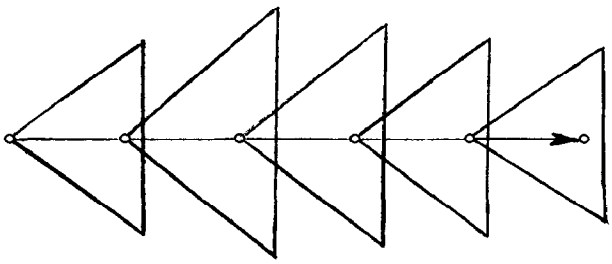


Fig. 2.1. Finding a path by short searches.

Unfortunately, it appears that any deterministic selection of a subset of the nodes is of little help. However, if we pick s "distinguished" nodes at random, it is well known (see Greene and Knuth [1982], e.g.) that a path is unlikely to have a gap longer than $O((n \log n)/s)$ with no distinguished node. More formally,

Lemma 2.2: If we choose s "distinguished" nodes at random from an n-node graph, then the probability that a given (acyclic) path has a sequence of more than $(cn \log n)/s$ nodes, none of which are distinguished is, for sufficiently large n, bounded above by $2^{-\alpha c}$ for some positive α. □

The case we shall exploit in this section uses about $\sqrt{n} \log n$ distinguished nodes, and therefore needs to search forward for about $\sqrt{n}$ distance from each distinguished node. As we shall see, there is an important "trick" that works best when the number of distinguished nodes and the distance explored are as close as possible. We begin by giving the algorithm for bounded-degree graphs, and then show how it extends naturally to all graphs.

An Algorithm for Bounded-Degree Graphs

Let us assume that all nodes have in-degree and out-degree at most 2. The time we shall take is $\tilde{O}(\sqrt{n})$, and we shall use n processors. The algorithm is outlined below.

Algorithm 2.3: Parallel, Single-Source Transitive Closure with High Probability, $\tilde{O}(\sqrt{n})$ Time, and n Processors.

INPUT: A directed graph G with n nodes, in- and out-degree 2. Also, a source node v_0.

OUTPUT: For each node of G, a decision whether the node is reached from v_0. The decision is correct with high probability, in the sense defined in Section 1.

METHOD: Perform each of the following steps.

1. Select $\sqrt{n} \log n$ nodes to be distinguished, at random. In what follows, we shall include the source v_0 among the distinguished nodes, even if it was not picked.
2. Search from all the distinguished nodes, to find, for each node v, a set of distinguished nodes that reach v. The set for node v must include all distinguished nodes that reach v along a path of length $\sqrt{n}$ or less, and it may include other distinguished nodes reaching v along longer paths, but may not include a node that does not reach v.
3. Construct a new graph H whose nodes are the distinguished nodes of G (including v_0). There is an arc $u \rightarrow v$ in H if it was determined in step (2) that u can reach v.
4. Compute the all-pairs transitive closure of H, and thus determine which of the distinguished nodes are reachable from the source v_0.
5. For each node v, determine whether there is a distinguished node w that
   a) Reaches v along a short path, as determined in step (2), and
   b) Is reached by v_0, as determined in step (4).

The algorithm above will detect all reachable nodes with some probability greater than 0. To reduce the error rate as far as we like (but not to 0), we can repeat steps (1) through (5) as many times as we like. For each desired error rate, there is some number of repetitions necessary, but this number does not depend on n. Combine the repetitions by saying a node v is reached from v_0 if any iteration says v is reachable. □

All but step (2) of Algorithm 2.3 can be accomplished in time $\tilde{O}(\sqrt{n})$ with n processors on a PRAM by straightforward means that will be reviewed in the full paper. Step 2 is not hard to accomplish within these limits either. One method involves dividing the n processors equally among the distinguished nodes and performing a breadth-first search from each one for $\sqrt{n}$ levels. This approach appears to require a $O(\log n)$ factor overhead to coordinate the processors. The method we actually use saves this logarithmic factor in running time.

To execute step (2), we assign one processor to each node. This processor creates a table in the shared PRAM memory indexed by the distinguished nodes.

The entry for distinguished node w in the table for node v tells

a) If it is known that w can reach v.
b) For successors x and y of v, whether w is known to reach x and whether w is known to reach y.

We also create two doubly linked lists for v of

a) Those distinguished nodes known to reach v but not known to reach successor x.
b) Those distinguished nodes known to reach v but not known to reach successor y.

The algorithm proceeds in $\tilde{O}(\sqrt{n})$ rounds to propagate information about which nodes are reached by which distinguished nodes. Initially, each distinguished node knows that it is reached by itself, and nothing else is known. This information is passed to its predecessors, so they can properly initialize their tables. In one round, the following messages are passed between nodes, and tables are updated accordingly.

1. If node v knows it is reached by a distinguished node w, and one of its successors x does not know that it is reached by w, then v sends x a message telling it *one* new distinguished node that reaches x. Note that each successor of v gets to learn about only one new distinguished node from each predecessor, in one round.

2. If v has learned from one predecessor z that v is reached by distinguished node w, then v tells its other predecessor, if it has one, that v now knows it is reached by w. Of course, z also knows that v now knows about w, so each predecessor of v can update its table accordingly.

The key property is stated in the following lemma.

Lemma 2.4: The algorithm for step (2) described above, when run with s distinguished nodes, will find all distinguished nodes that reach any given node along a path of distance d or less, provided we run the algorithm for at least $s + d$ rounds. □

The intuitive reason for this property is that, while a fact can be delayed in its propagation because a given node v has many other facts to tell one of its successors, no one fact can be delayed twice by the same fact. However, the true picture is somewhat more complex than that, since it is often unclear which fact causes the delay of another fact; we defer the proof for the full paper. In the case at hand, where s and d are both $\tilde{O}(\sqrt{n})$, we require $\tilde{O}(\sqrt{n})$ rounds, each of which can be executed in $O(1)$ time on a PRAM, to accomplish step (2) of Algorithm 2.3.

The Unlimited-Degree Case

If the graph does not have in- and out-degree 2, we begin with step (1) of Algorithm 2.3, selecting distinguished nodes at random. However, before proceeding to step (2), we convert the graph G to a graph G' that does have in- and out-degree 2. For each node v with more than two successors, we create a balanced binary tree to fan-out to those successors, and for each node v with more than two predecessors, we create a balanced binary tree to fan in.

The new graph G' has $O(e)$ nodes and edges, if e is the number of edges of G. Moreover, since the trees are balanced, no path through an introduced tree is longer than $\log n$. As a result, to search for distance d in G, it suffices to search for distance $d \log n$ in G'. In particular, we can now complete steps (2) through (5) of Algorithm 2.3 on the graph G', but searching distance $\sqrt{n} \log n$, instead of distance $\sqrt{n}$, and using e processors instead of n. Since in step (2) we have $\sqrt{n} \log n$ distinguished nodes, and we search for the same distance, the number of rounds $(s + d)$ that we need does not even go up by more than a factor of 2. Summarizing, we can show:

Theorem 2.5: The above algorithm determines whether there is a path from a given source node v_0 to each node v, with high probability. If the graph has n nodes and e arcs, the algorithm can be implemented to run in $\tilde{O}(\sqrt{n})$ time on a PRAM with e processors. □

3. HIGH-PROBABILITY REDUCTIONS FOR TRANSITIVE CLOSURE

In this section we shall consider the general "sparse" problem, which we call $S(s, d, n, e)$, of determining, in an $\tilde{O}(n)$-node, $\tilde{O}(e)$-arc graph, what nodes are reached along paths of distance at most $\tilde{O}(d)$, from each of $\tilde{O}(s)$ source nodes. We may optionally include in our answer other nodes reached from a source, but only along paths of length greater than $\tilde{O}(d)$. Our answer must be correct with high probability.

We shall assume there is a time limit $\tilde{O}(t)$ for obtaining the answer, and we need to compute the necessary work, that is, the product of the time taken and the number of processors used. For example, the problem studied in the previous section is $S(1, n, n, e)$, with a time limit $t = \sqrt{n}$. The other problem of significant interest is $S(n, n, n, e)$, the all-pairs problem. Note that factors of $\log n$ are elided in all parameters, and of course, will be elided in the measure of work.

While we are probably not very interested in instances other than the two mentioned, we need to consider this more general framework because a solution

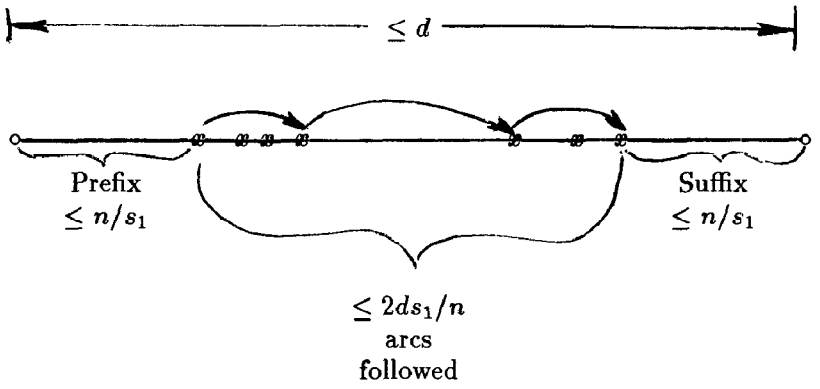


Fig. 3.1. Shortening paths in rule R1.

to a case that interests us can involve the solution to subproblems that appear less interesting. For example, in Section 2, we reduced the problem $S(1, n, n, e)$ to the problems $S(\sqrt{n}, \sqrt{n}, n, e)$ and $S(\sqrt{n}, \sqrt{n}, \sqrt{n}, n)$. The former is the problem we solved in step (2) of Algorithm 2.3, and the latter is the all-pairs transitive closure of the graph H.

We shall also have need to consider the "dense" problem on occasion. We let $D(s, d, n)$ stand for $S(s, d, n, n^2)$.

We use two "basis" (nonrecursive) rules for solving problems and four recursive rules. Each of these rules has some role to play, although in certain circumstances we shall show that one dominates the others as far as reducing the work is concerned. Incidentally, the reader may legitimately worry if, as we elide constant factors and factors of $\log n$, whether a recursion that hides, say, a factor of $\log n$ at each of n levels, isn't really hiding a dominant factor. However, when we analyze the rules for optimal strategies, we shall see that in no case do we need to recurse more than twice. Thus, hidden factors cannot accumulate.

Basis Rule B1:

Our first strategy is to solve the all-pairs transitive closure problem for the graph by path-doubling . We may use this strategy with any time limit whatsoever, since even $t = 1$ really means polylog time. We take work $\tilde{O}(n^3)$ to solve transitive closure this way.

Basis Rule B2:

This rule generalizes the method for step (2) of Algorithm 2.3. Divide the s sources into $\lceil s/d \rceil$ groups. As in step (2) of Algorithm 2.3, use e processors to search forward from the $O(d)$ sources of each group for distance d. By Lemma 2.4, this operation requires $\tilde{O}(d)$ time. We therefore put the work for this basis rule at $\tilde{O}(e\lceil s/d \rceil d) = \tilde{O}(es + ed)$.

Recursive Rule R1:

We can solve $S(s, d, n, e)$ by the following steps.

1. Select s_1 distinguished nodes at random. We may assume with high probability that there are no gaps greater than $(n \log n)/s_1$ in paths, so search forward from each of the distinguished nodes for distance $\tilde{O}(n/s_1)$.
2. Add to the given graph all arcs $u \rightarrow v$ between distinguished nodes that are discovered in step (1). Note that at most $\tilde{O}(s_1^2)$ arcs are added.
3. In the resulting graph, search forward from the original s sources for distance

   $$\tilde{O}(ds_1/n + n/s_1)$$

The intuitive reason this technique works is suggested in Fig. 3.1. Notice that the subproblem of step (1) is $S(s_1, n/s_1, n, e)$ and the subproblem of step (3) is $S(s, ds_1/n + n/s_1, n, e + s_1^2)$. It is easy to see that the work of step (2) could not exceed that of step (1). Consequently, we shall write rule R1 as

$$S(s, d, n, e) \rightarrow S(s_1, \frac{n}{s_1}, n, e) + S(s, \frac{ds_1}{n} + \frac{n}{s_1}, n, e + s_1^2)$$

The arrow can be interpreted as saying that the work of the problem on the left is no greater than the sum of the costs of the problems on the right. It can also be interpreted as saying that if we can solve the problems on the right with high probability, then we can solve the problem on the left with high probability, using, within some logarithmic factors, no more time or work than the sum of what is used to solve the problems on the

right.

Recursive Rule R2:

A similar strategy begins with step (1) of R1. However, we next consider the "dense" problem of computing the transitive closure of the graph consisting of the distinguished nodes only, and all arcs that were discovered among them in step (1); this graph is analogous to H in Algorithm 2.3. Referring to Fig. 3.1, once we take the transitive closure and then install the resulting arcs in the original graph, we can leap from the first distinguished node to the last, in one arc. The distance we need to follow is thus just the length of the prefix and suffix, plus 1, which is $\tilde{O}(n/s_1)$. That is also a bound on the length of a path that is so short it has no distinguished nodes.

We may thus express R2 as

$$S(s,d,n,e) \rightarrow S(s_1, \frac{n}{s_1}, n, e) + D(s_1, \frac{ds_1}{n}, s_1) + \\ + S(s, \frac{n}{s_1}, n, e + s_1^2)$$

Recall that D denotes the dense case, where e is the square of the number of nodes, s_1^2 here. The middle term represents the transitive closure, and the last term represents the final step, where we search for distance $\tilde{O}(n/s_1)$ from the original s sources in the augmented graph.

Recursive Rule R3:

In this variation, we do the following.

1. Select s_1 distinguished nodes at random. Reverse the arcs and search backward in the graph from the distinguished nodes for distance $\tilde{O}(n/s_1)$.
2. Construct a new graph whose nodes are the distinguished nodes and sources of the original graph, a total of at most $s+s_1$ nodes. Add to the new graph the arcs found in step 1: (a) between distinguished nodes, and (b) from a source to a distinguished node. That is, if distinguished node w was found to reach source v following arcs backward, then add arc $v \rightarrow w$. The maximum number of arcs in the new graph is s_1^2 from (a) and ss_1 from (b).
3. In the graph constructed in step 2, search forward from the sources for distance $2ds_1/n$.
4. Add to the original graph all the arcs $v \rightarrow w$ such that v is a source node and w a distinguished node found reachable from v in step 3. The number of arcs added to the original graph is at most ss_1.
5. In the graph constructed by step 4, search forward from the sources for distance $\tilde{O}(n/s_1)$. This distance suffices to follow any path of length d in the original graph.

As for R1, the cost of steps (2) and (4) can be neglected. We can express then R3 as

$$S(s,d,n,e) \rightarrow \\ S(s_1, \frac{n}{s_1}, n, e) + S(s, \frac{ds_1}{n}, s+s_1, s_1(s+s_1)) + \\ + S(s, \frac{n}{s_1}, n, e + ss_1)$$

Recursive Rule R4:

The last rule is rather different from the first three, and also quite simple. We split the number of sources into some number of groups, say k, and treat the groups independently. We shall see that in practice, this trick simplifies things when $s > d$, but it is never formally necessary; it was, in fact, incorporated into basis rule B2. The description of R4 is

$$S(s,d,n,e) \rightarrow kS(\frac{s}{k}, d, n, e)$$

4. A GENERAL STRATEGY

It turns out that we can solve the problem $S(s,d,n,e)$ for all values of the parameters, and all time limits t, by a fairly simple algorithm that applies at most two recursive rules and then applies one of the basis rules to each of the resulting subproblems.

Algorithm 4.1: Solution to the Generalized, Sparse Transitive Closure Problem.

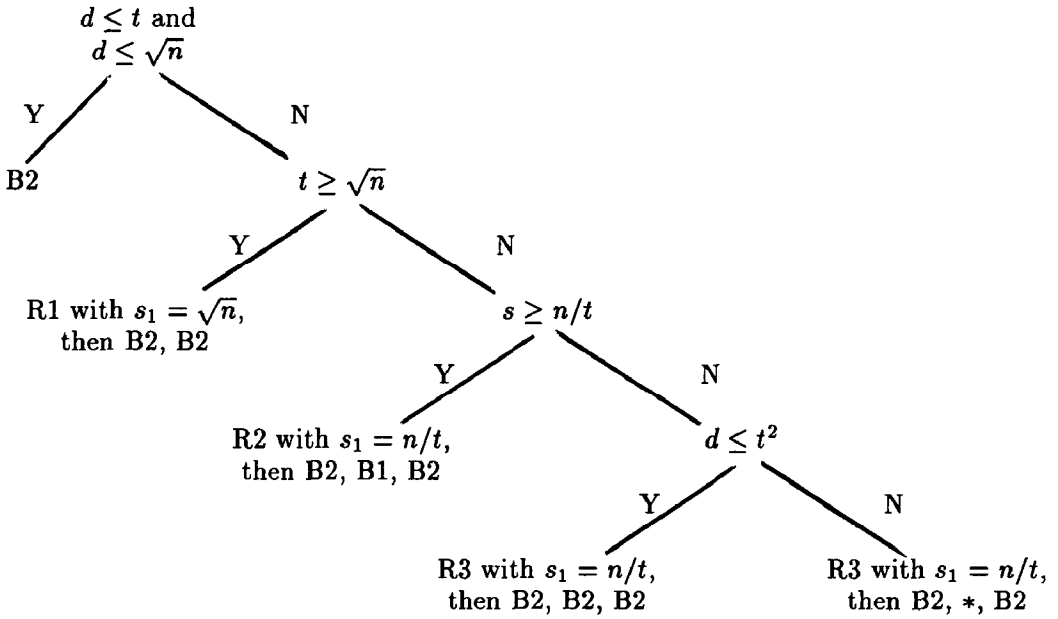
INPUT: A problem $S(s,d,n,e)$ and a time limit t.

OUTPUT: A strategy that solves the given problem in $\tilde{O}(t)$ time on a PRAM and uses as few processors as any strategy built from the rules B1, B2, R1, R2, R3, and R4 of the previous section.

METHOD: The algorithm follows the decision tree indicated in Figure 4.2. Depending on the value of the parameters, it arrives at a leaf and applies the strategy that is specified there. □

A Las Vegas Algorithm

We can modify Algorithm 4.1 to check the validity of its answer, using $O(es)$ work and $\tilde{O}(1)$ time; the work and time of this modification can be neglected. Suppose we have set X of nodes deemed by the algorithm to be reachable from source node v. Examine the arcs out of each node in X, in parallel, and if any are found to enter a node not in X, then we conclude that our answer is



$*$ = apply R2 with $s_1 = n/t^2$, then apply B2, B1, B2 to the subproblems.

Fig. 4.2. Summary of Algorithm 4.1.

incorrect. However, if for each source node, the set of reachable nodes is closed under successor, then we claim the answer is correct. We cannot say we reach a node not actually reached, because all paths found are real paths. We cannot say we fail to reach a node that we actually reach, or else some set X, ostensibly the nodes reachable from source v, would have an arc to a node not in X.

We can thus modify Algorithm 4.1 to be a Las Vegas algorithm by adding the check for validity just described. If the check fails, then we repeat the algorithm, until eventually the test succeeds. Since there is positive probability that the algorithm succeeds on any given round, the expected time of the algorithm is still $\tilde{O}(t)$, and the expected work is also not affected.

Analysis and Optimality of the General Algorithm

We shall now analyze Algorithm 4.1 and show that it is the best we can do, given the tools at hand — recursive rules R1 through R4 and basis rules B1 and B2. Define an *instance* of the transitive-closure problem to be a 4-tuple $I = (s, d, n, e)$, representing the arguments of the problem $S(s, d, n, e)$. We can define formally the notion of a *strategy* for solving instance I with time limit t using the rules of the previous section. We view a strategy as a tree (see for example Fig. 5.2) whose nodes are labelled by instances, with the root being labelled I. Attached to every internal node we have a recursive rule and a value for the number s_1 of distinguished nodes chosen, indicating how the instance associated with the node is solved, and attached to each leaf is a basis rule. Naturally, there are several requirements of consistency in the labelling, which we omit here. We can then define the *work* of a strategy recursively, up the tree.

The work performed by the strategy of Algorithm 4.1 depends on which of the five cases, corresponding to the leaves in Fig. 4.2, applies. We can combine the expressions into one function as follows. Define

$$f(I,t) = es + e\min(d, \sqrt{n}) + \alpha_1\left[e\frac{n}{t} + s\frac{n^2}{t^2}\right] + \alpha_2\left[\frac{n^3}{t^4}\right]$$

where $\alpha_1 = 1$ if $d > t$ and $\alpha_1 = 0$ otherwise; $\alpha_2 = 1$ if $d > t^2$, and $\alpha_2 = 0$ otherwise.

Theorem 4.4: The work of Algorithm 4.1's strategy is $\tilde{O}(f(I,t))$. □

We then prove:

Theorem 4.5: The cost of any strategy for solving $I(s, d, n, e)$ in time t is $\tilde{\Omega}(f(I,t))$. □

Corollary 4.6: The strategy of Algorithm 4.1 is optimal. □

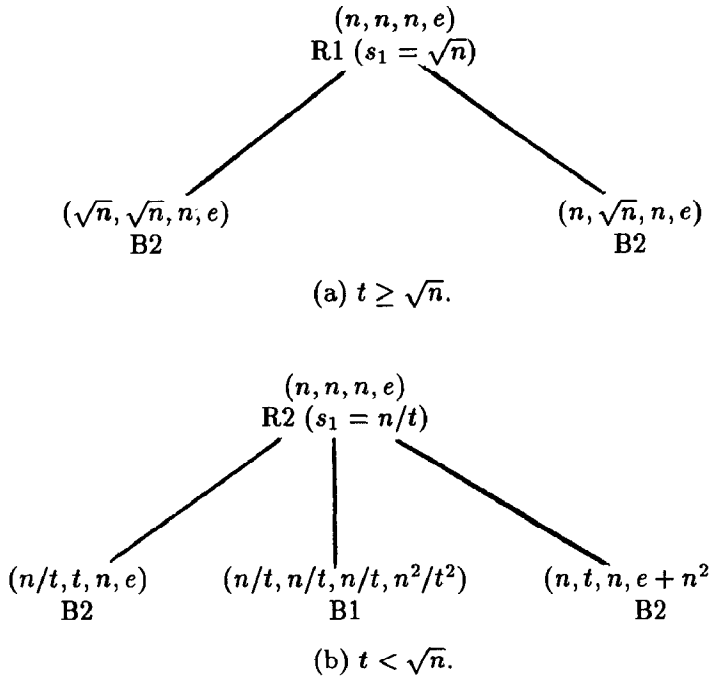


Fig. 5.2. Strategies for the all-pairs problem.

5. IMPORTANT SPECIAL CASES

The most important instances of the general transitive closure are the single-source and the all-pairs problems. These have d, the distance, equal to n, the number of nodes, and have s, the number of sources, equal to 1 or n.

All-Pairs Problems

When we let $s = d = n$ in the formula $f(I,t)$ of the previous section, we can simplify it to

$$f(n,n,n,e,t) = en + n^3/t^2 \tag{5.1}$$

Fig. 5.2(a) gives the strategy for the all-pairs case, when $t \geq \sqrt{n}$; here, Case (2) of Fig. 4.2 applies, and the cost is en. Figure 5.2(b) shows the strategy when $t < \sqrt{n}$; here Case (3) of Fig. 4.2 pertains.

A useful simplification is to assume that $t = n^\epsilon$ for some small ϵ, and $e = n^\alpha$, for some α between 1 and 2. In (5.1), the en term dominates as long as $\alpha \geq 2 - 2\epsilon$. We can thus solve the all-pairs problem optimally, as long as the graph is not too sparse. For time $\sqrt{n}$, that is, $\epsilon = 1/2$, there is no constraint on e, but as the time shrinks, the number of arcs must increase if the algorithm is to be optimal, until as the time approaches polylog time, the graph must be dense.

Single-Source Problems

Letting $s = 1$ and $d = n$ in the formula $f(I,t)$, we can simplify it to

$$f(1,n,n,e,t) = e\sqrt{n} + en/t + n^3/t^4 \tag{5.3}$$

Figure 5.4(a) shows the strategy of Algorithm 4.2 for the single-source problem when $t \geq \sqrt{n}$. It represents an alternative to Algorithm 2.3, which was defined only for the case $t = \sqrt{n}$, but evidently works for larger t.

If we assume $t = n^\epsilon$ for $\epsilon \leq 1/2$, then the $e\sqrt{n}$ term in (5.3) can be dropped. The term en/t dominates as long as $e = n^\alpha$, and $\alpha \geq 2 - 3\epsilon$. Put another way, we can solve the problem with time $t = \tilde{O}(n^\epsilon)$ and work $\tilde{O}(en^{1-\epsilon})$ (i.e., $en^{1-2\epsilon}$ processors), as long as $\epsilon \geq (2-\alpha)/3$. For example, even in the sparsest case, $\alpha = 1$, we can use $\tilde{O}(n^{1/3})$ time and $\tilde{O}(n^{4/3})$ processors.

For the dense case, where $e = n^2$, (5.3) reduces to $n^{2.5} + n^3/t$, or just n^3/t if we assume only $t \leq \sqrt{n}$ is of interest. Thus, for the single-source problem, even in the dense case we offer some improvement over the obvious solution of applying B1; the more time we're willing to take, up to $t = \sqrt{n}$, the more improvement in the total work we achieve. Notice, however, that if t is polylog in n, then we achieve nothing, since n^3/t stands for $\tilde{O}(n^3/t)$, which is $\tilde{O}(n^3)$ in this case. Thus, B1 is still the best $\mathcal{NC}$ algorithm we know for the dense, single-source problem.

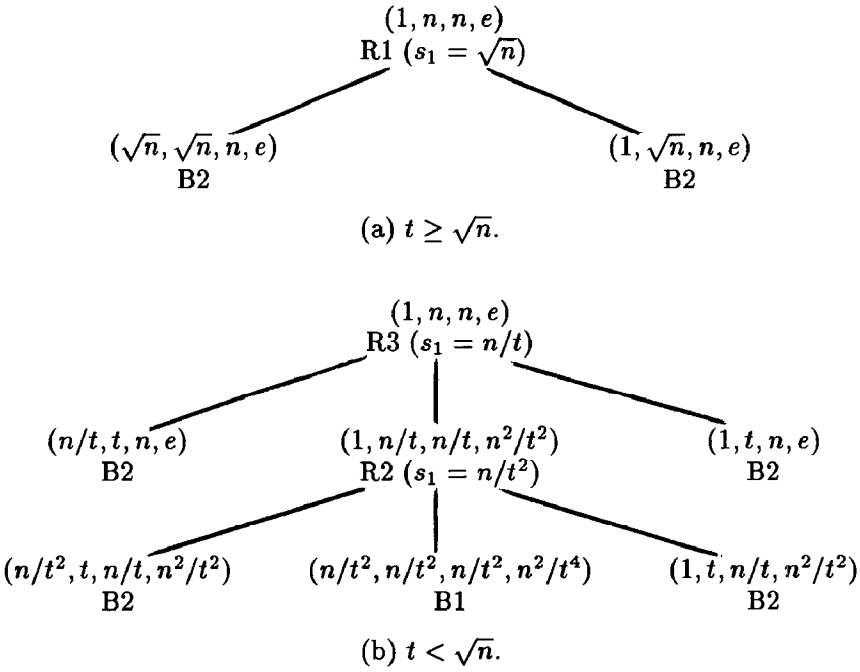


Fig. 5.4. Strategies for the single-source problem.

6. EXTENSION TO BREADTH-FIRST SEARCH

Ideally, we would like the techniques described here to work for generalized transitive closure, such as shortest-paths, or even general closed semirings, as in Aho, Hopcroft, and Ullman [1974]. We cannot do so, principally because the efficiency associated with basis rule B2 depends on each node being reached with only one fact about any distinguished node. However, we can push our techniques somewhat further, to solve the single-source BFS problem in the time given by (5.3); it is not known whether the same holds for the all-sources BFS problem and formula (5.1).

We shall discuss only the hard (and interesting) case when $t \leq \sqrt{n}$. The strategy we use is based on that illustrated in Fig. 5.4(b) for single-source transitive closure.

Algorithm 6.1: Single-Source Breadth-First Search.

INPUT: A graph G of n nodes and e arcs, and a source node v_0. Also, a time limit $t \leq \sqrt{n}$.

OUTPUT: For each node v in G, the length of (number of arcs on) the shortest path from v_0 to v.

METHOD: We perform each of the following steps.

1. Pick $\tilde{O}(n/t)$ distinguished nodes at random, and add v_0 to the set of distinguished nodes, whether or not it was picked.
2. Perform a breadth-first-search from each distinguished node for distance t.
3. Construct graph H whose nodes are the distinguished nodes selected in step (1). There is an arc $u \rightarrow v$ in H, weighted d, if in step (2) it was determined that the shortest path from u to v is of length $d \leq t$.
4. Select $\tilde{O}(n/t^2)$ superdistinguished nodes in H, and include v_0 among them.
5. Explore H from each superdistinguished node for weighted distance t^2.
6. Construct a graph J of the superdistinguished nodes, with weighted arcs as discovered in (5). Compute shortest paths in J by path-doubling-by-matrix-multiplication, using $\tilde{O}(t)$ time and $\tilde{O}(n^3/t^6)$ work, since the number of nodes is $\tilde{O}(n/t^2)$. Since v_0 is among the nodes of J, we now have the length of the shortest path from v_0 to each superdistinguished node, which we may regard as a vector of length $\tilde{O}(n/t^2)$.
7. Treat the information obtained in step (5), giving shortest paths of length up to t^2 from the superdistinguished nodes to all distinguished nodes as an $\tilde{O}(n/t^2) \times \tilde{O}(n/t)$ matrix. Multiply this matrix by

the vector from step (6), using + for scalar multiplication and min for scalar addition, thus obtaining the length of the shortest path from v_0 to every distinguished node. The obvious algorithm can be performed in time $\tilde{O}(t)$ and work $\tilde{O}(n^2/t^3)$. Again, regard this information as a vector, this time of length $\tilde{O}(n/t)$.

8. Treat the information obtained in step (2), giving shortest paths of length up to t from the distinguished nodes to all nodes, as an $\tilde{O}(n/t) \times n$ matrix. Multiply this matrix by the vector from step (7), again using + for scalar multiplication and min for scalar addition. The result, is the length of the shortest path from v_0 to every node. The obvious algorithm can be performed in $\tilde{O}(t)$ time and $\tilde{O}(n^2/t)$ work. □

The main complication in the implementation of Algorithm 6.1 is that although we start with an unweighted graph, when we recurse we need to consider a weighted graph H, and performing step 5 would seem to require the extension of basis rule B2 to weighted graphs, which in general we do not know how to do. However, H is not a general weighted graph, and one can take advantage of its structure to perform step 5 in time $\tilde{O}(t)$ using work $\tilde{O}(n^3/t^4)$. We can then show:

Theorem 6.2: Algorithm 6.1 takes time $\tilde{O}(t)$ and work $\tilde{O}(en/t + n^3/t^4)$. □

7. OPEN PROBLEMS

We have made some progress toward the real goal of deterministic, optimal parallel algorithms for single-source and sparse cases of transitive closure and related problems, such as shortest paths. There is a long sequence of open problems suggested by these results. We enumerate some of them here.

1. Can the algorithms be made not to depend on probabilistic choice; that is, are there deterministic algorithms with the same performance?

2. Can we reduce the work still further, using a one-sided-error, high-probability algorithm, like the ones presented here?

3. Can we generalize the algorithm for sparse, all-pairs transitive closure to the breadth-first-search problem? We note that work $en + n^3/t$ is achievable, which is better than the obvious algorithm, but does not match the bound of (5.1).

4. Does any of this material generalize to the general shortest-path problem, where arcs initially have arbitrary weights?

ACKNOWLEDGEMENTS

We were helped by discussions of this material with Amotz Bar-Noy, Joan Feigenbaum, Seffi Naor, and Vijay Vazirani. The use of computer equipment belonging to Princeton University is acknowledged by the first author.

REFERENCES

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

Bloniarz, P. A., M. J. Fischer, and A. R. Meyer [1976]. "A note on the average time to compute transitive closure," *Intl. Conf. on Automata, Languages, and Programming.*

Brent, R. P. [1974]. "The parallel evaluation of general arithmetic expressions," *J. ACM* **21**:2, pp. 201–208.

Broder, A. Z., A. R. Karlin, P. Raghavan, and E. Upfal [1989]. "Trading space for time in undirected *s-t* connectivity," *Proc. Twenty-first ACM Symp. on Theory of Computing.*

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

Gazit, H. [1986]. "An optimal randomized parallel algorithm for finding connected components in a graph," *Proc. Twenty-Seventh Annual IEEE Symposium on Foundations of Computer Science*, pp. 492–501.

Gazit, H. and G. L. Miller [1988]. "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Information Processing Letters* **28**:1, pp. 61–65.

Greene, D. H. and D. E. Knuth [1982]. *Mathematics for the Analysis of Algorithms*, Birkhauser, Boston.

Schnorr, C. P. [1978]. "An algorithm for transitive closure with linear expected time," *SIAM J. Computing* **7**:1, pp. 127–133.

Shiloach, Y. and U. Vishkin [1982]. "An $O(\log n)$ parallel connectivity algorithm," *J. Algorithms* **3**:1, pp. 57–63.

Simon, K. [1986]. "An improved algorithm for transitive closure on acyclic digraphs," *Intl. Conf. on Automata, Languages, and Programming*, pp. 376–386.