SIAM J. COMPUT.
Vol. 20, No. 1, pp. 100-125, February 1991

HIGH-PROBABILITY PARALLEL TRANSITIVE-CLOSURE ALGORITHMS*

JEFFREY D. ULLMAN† AND MIHALIS YANNAKAKIS‡

Abstract. There is a straightforward algorithm for computing the transitive-closure of an n-node 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 serial matrix multiplication in $M(n)$ time can be done. 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. It would be ideal to have an $\mathcal{NC}$ algorithm for transitive-closure that took about e processors for the single-source problem on a graph with n nodes and $e \geqq n$ arcs, or about en processors for the all-pairs problem on the same graph. While an algorithm that good cannot be offered, algorithms with the following performance can be offered. (1) For single-source, $\tilde{O}(n^\varepsilon)$ time with $\tilde{O}(en^{1-2\varepsilon})$ processors, provided $e \geqq n^{2-3\varepsilon}$, and (2) for all-pairs, $\tilde{O}(n^\varepsilon)$ time and $\tilde{O}(en^{1-\varepsilon})$ processors, provided $e \geqq n^{2-2\varepsilon}$.[1] Each of these claims assumes that $0 < \varepsilon \leqq \frac{1}{2}$. 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, it is shown that incorrect results can be detected, thus putting the algorithm in the "Las Vegas" class. Finally, it is shown how to do "breadth-first-search" with the same performance as can be achieved for single-source transitive closure.

Key words. transitive closure, reachability, breadth-first search, parallel algorithms, probabilistic algorithms

AMS(MOS) subject classifications. 68Q20, 68Q25, 68R10

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, e.g., Aho, Hopcroft, and Ullman [1974]) 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

* Received by the editors July 12, 1989; accepted for publication (in revised form) April 17, 1990.

† Department of Computer Science, Stanford University, Stanford, California 94305. The work of this author was partially supported by Office of Naval Research contract N00014-88-K-0166 and by a Guggenheim Fellowship.

‡ AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

[1] $\tilde{O}$ is the notation, proposed by Luks and furthered by 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 $f(n) \leqq c(\log n)^k g(n)$. Intuitively, just as "big-oh" elides constant factors, $\tilde{O}$ elides logarithmic factors.

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

[3] "Sparse" normally means that $e \ll n^2$, although if we believe that one in the sequence of "fast" matrix multiplication algorithms that have been proposed (see, e.g., Coppersmith and Winograd [1987]), each taking $M(n)$ time where $M(n)$ is some power of n between two and three, will turn out to be practical, then "sparse" should be taken to mean $e \ll (M(n) \log n)/n$.

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) 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.

In this paper, we shall describe versions of algorithms in which we only claim that there is a positive probability of finding a path when one exists. Naturally, if the algorithm is repeated $O(\log(1/\varepsilon))$ times, we can reduce the probability of failing to find a path to whatever $\varepsilon > 0$ we desire.

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 running 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 $\varepsilon > 0$, 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. Transitive-closure on an undirected graph is really the "connected components" problem. Shiloach and Vishkin [1982] gives the basic algorithm, while a series of improvements culminating in Cole and Vishkin [1986] improved the total work or elapsed time for very sparse graphs. Gazit [1986] gives an optimal randomized algorithm for the same problem.

Gazit and Miller [1988] offer 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 et al. [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, because it is easy to intuitively "trap" a random walk on a directed graph.

Organization of the paper. In § 2 we introduce 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. Section 3 provides the analysis of this algorithm. Then, in § 4 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 § 5 we give a general algorithm for the case of sparse graphs. Section 6 shows that this algorithm is the best that can be achieved using the four reductions of § 4. Section 7 gives some simplifications of the general algorithm of § 5 for interesting special cases: dense graphs, single-source, and all-pairs problems. Finally, in § 8, 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.

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.

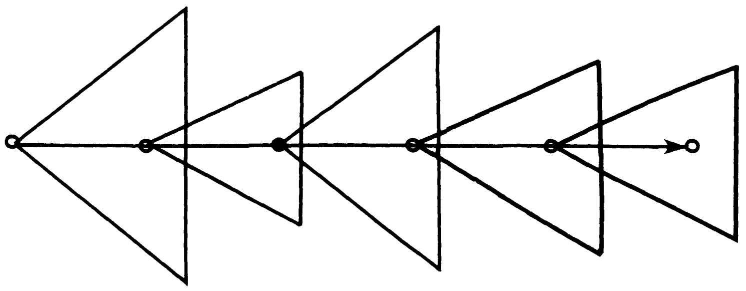

FIG. 2.1. *Finding a path by short searches.*

Unfortunately, it appears that any deterministic selection of a subset of the nodes is of little help. Given a selection of, say, half the nodes, we can always pick a graph in which the only path from the source to some node has a gap of length $n/2$ along which no selected node occurs. Thus, we would have to search distance $n/2$ from our $n/2$ selected nodes, which is almost the same as computing the full transitive-closure. However, if we pick s "distinguished" nodes at random, it is well known (see, e.g., Greene and Knuth [1982]) that a path is unlikely to have a gap longer than $O((n \log n)/s)$ with no distinguished node. More formally, we have Lemma 2.2.

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 now assume that all nodes have in-degree at most two and out-degree at most two. 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 two. 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 § 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 $u \to v$ 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 zero. To reduce the error rate as far as we like (but not to zero), we can repeat steps (1)–(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.

Details of step (2). 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 next section. Step 2 is not hard to accomplish within these limits either. Suppose we divide the n processors equally among the distinguished nodes. Then each would get $\tilde{\Omega}(\sqrt{n})$ processors. The processors assigned to w could perform a breadth-first search from w cooperatively. If at any level, there were at most $\tilde{O}(\sqrt{n})$ new nodes, the construction of the next level could be done in $O(1)$ time, since we assume out-degree at most two. If there are more than $\tilde{O}(\sqrt{n})$ nodes at a level, we must treat them in groups of $\sqrt{n}$. However, there are only n nodes among all the levels, so the total delay due to "excess" nodes at a level is no more than $O(\sqrt{n})$.

The above approach depends on being able to assign processors consistently, even though a node may be reached from two different predecessors. The coordination can be done, but appears to require $O(\log n)$ time to do so. 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. Note that the predecessors of distinguished node w know that their successor w is "reached by w," but the predecessors do *not* know that they are reached by w themselves; indeed they may not be. However, it is important that, if they turn out to be reached by w, they do not waste time telling w this fact. This avoidance of wasted messages is the reason why nodes need to know what their successors know.

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.

If there are s distinguished nodes, and we want to propagate reachability information for distance at least d, then we require only $d+s$ rounds to do so, as we shall prove in § 3. The intuitive reason 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. 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 two, 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 two. For each node v with more than two successors, we create a

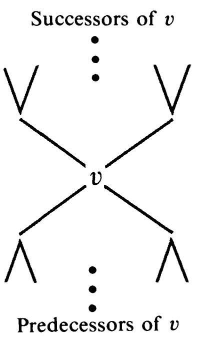

FIG. 2.4. *Forcing in- and out-degree two.*

balanced binary tree to fanout to those successors, and for each node v with more than two predecessors, we create a balanced binary tree to fanin, as suggested by Fig. 2.4.

The number of introduced nodes for a node v is no greater than the sum of the in- and out-degrees of v. Thus, G' has $O(e)$ nodes, if e is the number of edges of G. Since G' clearly has in- and out-degree two, G' also has $O(e)$ edges.

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)-(5) of Algorithm 2.3 on the graph G', but by searching distance $\sqrt{n} \log n$, instead of distance $\sqrt{n}$, and by 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.[4] Our conclusion is that it is possible to do single-source transitive-closure for an arbitrary graph in $\tilde{O}(\sqrt{n})$ time with e processors.

3. Correctness and efficiency of the simple algorithm.

We now need to show that the algorithm described in § 2 is correct with high probability, and we must show that its running time and processor utilization are as claimed in that section. The key point is the correctness of step (2) of Algorithm 2.3, which is implied by the first lemma.

LEMMA 3.1. *The algorithm for step* (2) *described in the previous section, 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.*

Proof. Consider a path $v_0 \rightarrow \cdots \rightarrow v_k$ along which propagates the information that distinguished node v_0 reaches node v_k. After r rounds, suppose that the reachability of v_0 has propagated to v_i, but not to v_{i+1}; that is, v_i knows it is reached by v_0, but v_{i+1} does not.[5] Let the *delay* be $\delta = r - i$, that is, the number of rounds on which v_0 failed to make progress along this path. We claim, and shall prove by induction on r, that there is a "wedge of knowledge," suggested in Fig. 3.2, where v_{i+1} knows about at least δ different distinguished nodes that reach it, v_{i+2} knows about at least $\delta - 1$, and so on.

Formally, we shall show by induction on r that if $v_0, \cdots, v_i$ know about v_0, and v_{i+1} does not, then for $j = 1, \cdots, \delta = r - i$, v_{i+j} knows about at least $\delta + 1 - j$ distinguished nodes that reach it. The basis, $r = 0$, is trivial, since then $i = 0$ and $\delta = 0$, so the "wedge" has height zero.

[4] But note that had we started with a bounded degree graph, we could choose $\sqrt{n \log n}$ sources and search for exactly that distance, thus improving the running time of step (2) by a factor of $\sqrt{\log n}$.

[5] Note that the reachability of v_0 may be known further along the path, because that information has propagated by another route.

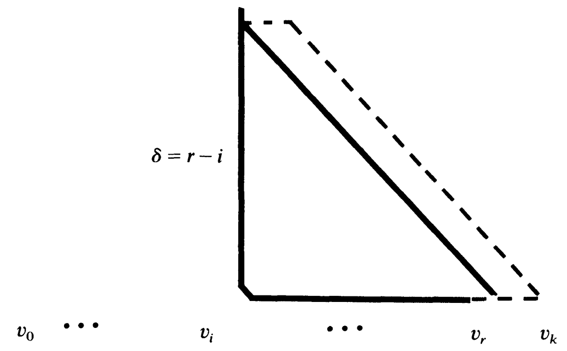

FIG. 3.2. *Wedge of knowledge.*

Now, suppose we have proceeded for r rounds, and the inductive hypothesis holds. Consider what happens on round $r+1$. First, make the following observation.

(∗) If at some time, v_p knows about strictly more distinguished nodes than v_{p+1}, then on the next round v_{p+1} will learn at least one new fact.

Thus, the number of facts known by each of $v_{i+2}, \cdots, v_{r+1}$ will, after round $r+1$, be at least one greater than that implied by the solid wedge of Fig. 3.2; this profile is suggested by the dashed wedge in that figure. That is, by (∗), we deduce that for $j = 2, \cdots, \delta+1$, node v_{i+j} knows about at least $\delta+2-j$ distinguished nodes that reach it by round $r+1$.

Case 1. v_{i+1} does not learn about v_0 at round $r+1$. Then v_i must tell v_{i+1} something else, because at each round, a node tells each successor something new if it can. Thus, after round $r+1$, v_{i+1} knows about at least $\delta+1$ distinguished nodes. Thus, for $j = 1, \cdots, r+1-i$, node v_{i+j} knows about at least $\delta+2-j$ distinguished nodes. The delay at round $r+1$ is $\delta+1$, so the inductive hypothesis is proved.

Case 2. v_{i+1} learns about v_0 at round $r+1$. Suppose that now $v_0, \cdots, v_m$ know about v_0, but v_{m+1} does not. If $m \geqq r+1$, the inductive hypothesis holds vacuously. Otherwise, the new delay is $\delta' = r+1-m$. Since

(1) $m > i$,

(2) $\delta' = \delta - m + i + 1$, and

(3) for $j = 2, \cdots, \delta+1$, node v_{i+j} knows about $\delta+1$ distinguished nodes,

we know that for $j' = 1, \cdots, \delta'$, node $v_{m+j'}$ knows about $\delta'+1-j'$ distinguished nodes (substitute $j'+m-i$ for j in (3)). Thus, the inductive hypothesis holds for round $r+1$.

Now, we observe that there are only s distinguished nodes that any node can ever know about. Thus, s is a limit on the height of a "wedge." That is, the delay cannot exceed s. Suppose that after r rounds, v_k does not know about v_0 yet. Then there is some $i < k$ such that $v_0, \cdots, v_i$ know about that v_0 and v_{i+1} does not. The delay is $r-i$, which is strictly greater than $r-k$. Thus, $s > r-k$.

Finally, consider $r = d+s$. We deduced that $s > r-k = d+s-k$, or equivalently, $k > d$. That is, after $s+d$ rounds, if v_k does not know about v_0, then $k > d$. Put another way, if some node v is ignorant of the fact that v_0 reaches v, then there can be no path of length d or less from v_0 to v. Thus, after $s+d$ rounds, all nodes learn about all distinguished nodes that reach them along paths of length d or less. □

Correctness of the algorithm. We now need to check that the remaining steps of Algorithm 2.3 and its extension to the general case are correct. It is easy to check that whenever a path is discovered, there is truly a path, so we need only to verify that if a path exists, it is discovered with positive probability. We may then argue that by increasing the number of distinguished nodes by a constant factor, the probability of error can be made as small as we wish.

Lemma 2.2 assures us that step (1), the selection of random distinguished nodes, leaves us a positive probability that for every node v, the shortest path from the source to v is *gapless*; that is, it has no gap of more than $\sqrt{n}$ consecutive, unselected nodes. Gaps of length $\sqrt{n}$ cannot grow to more than $\sqrt{n}\log n$ nodes after we convert graph G to the graph G' by adding fanin and fanout trees for the general case. Lemma 3.1 just showed that step (2) finds the paths between successive distinguished nodes on any gapless path. If we run for $2\sqrt{n}\log n$ rounds in step (2), then we are sure to have the necessary connections between distinguished nodes.

In particular, if our path is gapless, and w is the last distinguished node on the path (w may be the source, v_0), then steps (3) and (4) (construction of the graph H and computation of its transitive-closure) determine that there is a path from v_0 to w. Also, as the path is gapless, step (2) established that there is a path from w to v, the last node on the path. Thus, step (5) deduces that there is a path from v_0 to v. We have thus proved Theorem 3.3.

THEOREM 3.3. *The algorithm of* § 2 *determines whether there is a path from* v_0 *to each node* v, *with high probability.* □

Analysis of the algorithm. Now, we need to verify that each of the steps of Algorithm 2.3 and its extension to unbounded-degree graphs can be performed in $\tilde{O}(\sqrt{n})$ time on a PRAM with n processors. First, let us agree on a representation for graphs in the shared memory of the PRAM. The nodes will be assumed to be numbered $1, 2, \cdots, n$. Suppose that there is an array indexed by nodes, giving the number of in- and out-arcs for each node, a pointer to an array where the successors of the node are listed, and a pointer to an array where the predecessors of the node are listed.

Step (1), the selection of $\sqrt{n}\log n$ random nodes, can be done by one processor in the requisite time. However, further on in this paper, we shall need to do somewhat better: generate $\tilde{O}(n^{1-\varepsilon})$ random nodes in $\tilde{O}(n^{\varepsilon})$ time, using n processors. We can do so by the following steps.

(1) Each of the n processors selects a random node and creates a record (i, j), where j is the processor number and i the number of the node selected.

(2) The records are sorted lexicographically.

(3) Each record r looks at its predecessor to see if it has the same node (first component). If so, r is eliminated. Thus, for each selected node, only one record remains, and it is the record with the lowest-numbered processor that selected that node.

(4) Sort the remaining records by processor number, and select as many as desired from the front of this list, to be the chosen random nodes.

Thus, the entire process takes polylog time.

Step (2) of Algorithm 2.3 was analyzed in Lemma 3.1. There we proved that $\tilde{O}(\sqrt{n})$ rounds sufficed. We have only to observe that each round can be accomplished in $O(1)$ time. Message passing is done by creating for each node four words in the shared memory, for receiving messages from its predecessors and successors. Each arc into or out of the node is associated with a particular word. To send a message, a node writes into the proper word belonging to the receiving node.

For step (3), note that the graph H has $\tilde{O}(\sqrt{n})$ nodes and $\tilde{O}(n)$ arcs. Its arcs can be read from the tables created in step (2) for the distinguished nodes. That is, assign $\tilde{O}(1)$ arcs to each processor. To consider whether there is an arc $u \to v$ in H, the processor goes to the table for v and the entry for u. To get H into our form for graphs, we can compact the lists of predecessors and successors in $\tilde{O}(1)$ time by assigning $\sqrt{n}$ processors to each list and performing a parallel prefix operation. Thus, the entire construction of H is accomplished in $\tilde{O}(1)$ time.

Step (4), taking the transitive closure of H, can be done in $\tilde{O}(\sqrt{n})$ time with n processors by the ordinary, path-doubling-by-matrix-multiplication method. That is, for a graph with $\tilde{O}(\sqrt{n})$ nodes, the method usually is done in $\log^2 n$ time with $\tilde{O}(n^{3/2})$ processors, but we can trade time for processors by "Brent's theorem" (Brent [1974]), to achieve the desired bounds. Now, we look at the successors of the source v_0 in the transitive closure of H, and make a table, or vector, indicating for each distinguished node whether that node is reached in H from v_0. This part requires $O(1)$ time with n (or even $\sqrt{n}$) processors.

In step (5), we assign one processor to each node v, and examine the table of distinguished nodes that reach v. For each distinguished node, we examine the table constructed in step (4) to see whether v_0 reaches that distinguished node. The table lookup requires $O(1)$ time, so the time for considering all distinguished nodes is $\tilde{O}(\sqrt{n})$.

Finally, we must note the difference when the initial graph G does not have bounded degree. Now we are given e processors to convert G to G' by introducing fanin and fanout trees, as described at the end of § 2. First, assign to each node a number of processors equal to the out-degree of that node. The fanout tree can then be constructed in $\tilde{O}(1)$ time by obvious means. Similarly, we build the fanin trees in the same time. From that point, we proceed as in Algorithm 2.3, but with e processors, at most $n+2e$ nodes, and at most $5e$ arcs. In step (2), there are $\tilde{O}(\sqrt{n})$ distinguished nodes that must be explored for distance $\tilde{O}(\sqrt{n})$, so the time bound $\tilde{O}(\sqrt{n})$ still holds, as long as there are n processors. Steps (3) and (4) do not change, since H still has $\tilde{O}(\sqrt{n})$ nodes, and we have $e \geqq n$ processors. Step (5) is similar, since we can still assign one processor to each node and do the work in $\tilde{O}(\sqrt{n})$ time. We have thus proved Theorem 3.4.

THEOREM 3.4. *We can compute with high probability the single-source transitive-closure of an n-node, e-arc graph in $\tilde{O}(\sqrt{n})$ time on a* PRAM *with e processors.* □

4. 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 that 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 two sections 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 we have devised a number of "reductions" among instances of the general problem, so a solution to a case that interests us can involve the solution to subproblems that appear less interesting. For example, in § 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, is not 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. 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. This strategy may seem useless, but in fact it is needed for certain subproblems with a small number of nodes, as we found for graph H in Algorithm 2.3.

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 3.1, this operation requires $\tilde{O}(d)$ time. We therefore put the work for this basis rule at $es+ed$.[6] Technically, the work is $\tilde{O}(e\lceil s/d \rceil d)$, but the reader can easily check that $\tilde{O}(es+ed)$ is the same. That is, if $s \leqq d$, then $\lceil s/d \rceil = 1$, and the term ed predominates. If $s>d$, $\lceil s/d \rceil d$ is about s, and es predominates.

Recursive Rule R1. We can solve $S(s, d, n, e)$ by the following steps.

(1) Select s_1 distinguished nodes at random, from among all the nodes of the graph. 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 this distance, which is $\tilde{O}(n/s_1)$.

(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 that is sufficient to reach any node from any source. This distance is $\tilde{O}(ds_1/n + n/s_1)$.

The intuitive reason the distance suggested in (3) suffices is suggested in Fig. 4.1. A path of length d in the original graph is collapsed in the new graph so that we only have to follow from the source to the first distinguished node (at most n/s_1, since we assume no "big" gaps), then along introduced arcs to the last distinguished node, and finally to the end of the path (again, at most distance n/s_1). If there are three distinguished nodes within distance n/s_1 along the path, then we can skip the middle one. Thus, if there is distance at most d along the path, and we follow more than $2ds_1/n$ introduced arcs between distinguished nodes, we can find a shorter path, assuming there are no gaps longer than n/s_1. We conclude that distance $\tilde{O}(n/s_1)$ suffices for the *prefix* and *suffix* of the path (outside the first and last distinguished

[6] Recall that we are dropping logarithmic factors from our work estimates, as well as from all parameters. The work is really $\tilde{O}(es+ed)$.

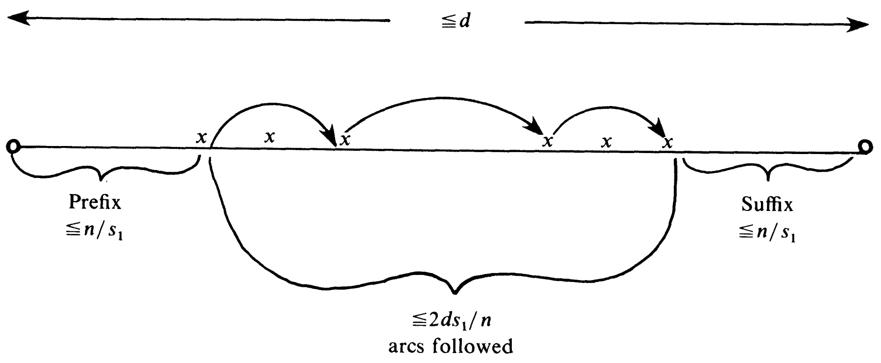

FIG. 4.1. *Shortening paths in rule* R1.

nodes), and $\tilde{O}(ds_1/n)$ suffices in the middle, jumping among distinguished nodes. In the special case in which the path has no distinguished nodes, we may assume that it has length no greater than the maximum gap length, or n/s_1.

Note that the subproblem of step (1) is $S(s_1, n/s_1, n, e)$ and the subproblem of step (3) is $S(s, ds_1/n + n/s_1, n, e + s_1^2)$. Step (2), the addition of arcs, can be done with work proportional to the number of those arcs, which could not be greater than the work of step (2), constructing those arcs in the first place. Furthermore, the addition of the arcs to the graph can be done in parallel, to the extent allowed by the number of processors. Thus, we claim that the work of step (2) could not exceed that of step (1). Consequently, we shall write rule R1 as

$$\text{(R1)} \qquad S(s, d, n, e) \rightarrow S\left(s_1, \frac{n}{s_1}, n, e\right) + S\left(s, \frac{ds_1}{n} + \frac{n}{s_1}, n, e + s_1^2\right).$$

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. 4.1, 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 one, which is $\tilde{O}(n/s_1)$, as suggested by Fig. 4.2. 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

$$\text{(R2)} \qquad S(s, d, n, e) \rightarrow S\left(s_1, \frac{n}{s_1}, n, e\right) + D\left(s_1, \frac{ds_1}{n}, s_1\right) + S\left(s, \frac{n}{s_1}, n, e + s_1^2\right).$$

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.

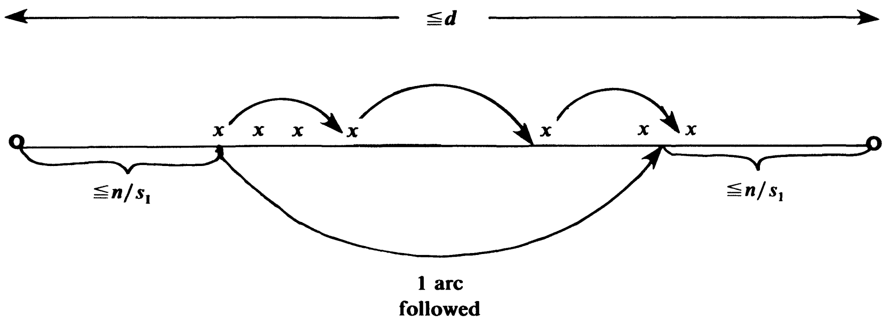

FIG. 4.2. *Shortening paths in rule* R2.

Recursive Rule R3. In this variation, we do the following.

(1) Select s_1 distinguished nodes at random and search forward from them for distance $\tilde{O}(n/s_1)$, as in R1 and R2.

(2) Reverse the arcs and search backward in the graph from the same nodes for the same distance.

(3) 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) between distinguished nodes. Also add the arcs found in (2) 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 $v \to w$. The maximum number of arcs in the new graph is s_1^2 from step (1) and ss_1 from step (2).

(4) In the graph constructed in step (3), search forward from the sources for distance $2ds_1/n$. As we argued concerning R1, this distance is sufficient to get us from any source to the last distinguished node on any path from that source.

(5) Add to the original graph all the arcs $v \to w$ such that v is a source node and w a distinguished node found reachable from v in step (4). The number of arcs added to the original graph is at most ss_1.

(6) In the graph constructed by step (5), 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, since in the augmented graph we can go in one arc to the last distinguished node, and then need only to follow the suffix. Similarly, a path with no distinguished nodes is assumed to be no longer than n/s_1.

As for R1, we can argue that steps (3) and (5) can be performed with work that does not exceed the work of constructing the arcs, and that these steps can be performed with the maximum parallelism allowed by the number of processors. Thus we may neglect the cost of these steps. The work of constructing a new graph in step (2) with the arcs reversed is no greater than $\tilde{O}(e)$, and it can be done with the maximum amount of parallelism allowed by the number of processors, down to $\tilde{O}(1)$ time. The details require some thought, but are similar to the techniques already mentioned in the construction leading to Theorem 3.4. Since the initial subproblem surely requires work $\tilde{\Omega}(e)$, or else we cannot even look at all the arcs, we shall neglect the cost of reversing the arcs. We thus can express R3 as

$$\text{(R3)} \quad S(s, d, n, e) \to S\left(s_1, \frac{n}{s_1}, n, e\right) + S\left(s, \frac{ds_1}{n}, s+s_1, s_1(s+s_1)\right) + S\left(s, \frac{n}{s_1}, n, e+ss_1\right).$$

The first term represents the cost of the searches in steps (1) and (2); note that the multiplier 2 would be appropriate but unnecessary. The second term represents step (4), and the last term represents step (6).

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

$$\text{(R4)} \qquad S(s, d, n, e) \to kS\left(\frac{s}{k}, d, n, e\right).$$

We may summarize the arguments given in connection with each of the rules by Theorem 4.3.

THEOREM 4.3. *Each of the rules* R1, R2, R3, *and* R4 *is correct, in the sense that if the subproblems on the right can be solved with high probability in a certain amount of time and work, then the problem on the left can likewise be solved with high probability, with the same amount of work, neglecting factors of* log n. □

5. 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. The strategy is suggested by the tree of Fig. 5.1. More formally, the strategy is given by the following algorithm.

ALGORITHM 5.2. 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 consists of the following decisions.

(1) If $d \leqq t$ and $d \leqq \sqrt{n}$ then apply basis rule B2.
(2) Otherwise (i.e., $d > t$ and/or $d > \sqrt{n}$), if $t \geqq \sqrt{n}$ then apply recursive rule R1 with $s_1 = \sqrt{n}$, and then apply B2 to the two subproblems that result.
(3) Otherwise (i.e., $d > t$ and/or $d > \sqrt{n}$, and also $t < \sqrt{n}$), if $s \geqq n/t$ then apply R2 with $s_1 = n/t$. Apply B2 to the first and last subproblems, and apply B1 to the middle subproblem $S(s_1, ds_1/n, s_1, s_1^2)$.

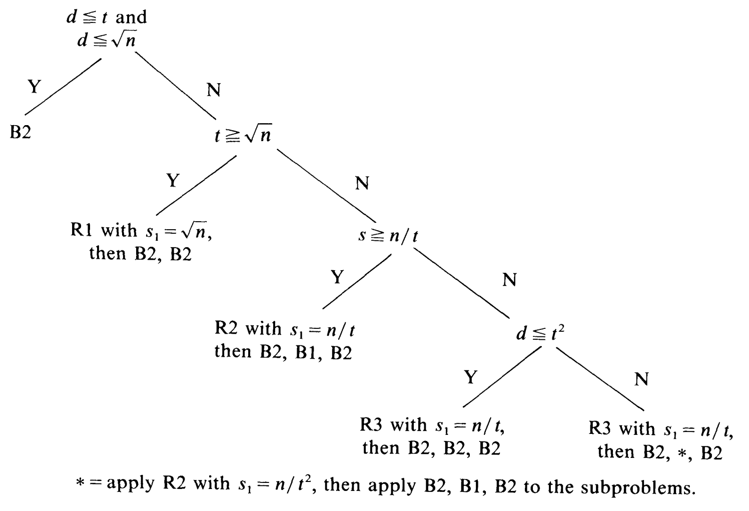
$d \leqq t$ and $d \leqq \sqrt{n}$
Y
N
B2
$t \geqq \sqrt{n}$
Y
N
R1 with $s_1 = \sqrt{n}$, then B2, B2
$s \geqq n/t$
Y
N
R2 with $s_1 = n/t$ then B2, B1, B2
$d \leqq t^2$
Y
N
R3 with $s_1 = n/t$, then B2, B2, B2
R3 with $s_1 = n/t$, then B2, *, B2
* = apply R2 with $s_1 = n/t^2$, then apply B2, B1, B2 to the subproblems.

FIG. 5.1. *Summary of Algorithm* 5.2.

(4) Otherwise (i.e., $d > t$ and/or $d > \sqrt{n}$, also $t < \sqrt{n}$, and also $s < n/t$), apply R3 with $s_1 = n/t$. If $d \leqq t^2$, apply B2 to each of the three generated subproblems. If $d > t^2$, there is not enough time to apply B2 to the middle subproblem, so use B2 for the first and third subproblems, and apply R2 with $s_1 = n/t^2$ to the middle subproblem. Apply B2, B1, and B2, respectively, to the three subproblems so generated.

Analysis of Algorithm 5.2. Let us now compute the function that gives the work required by Algorithm 5.2. That function has five cases, corresponding to the five leaves of Fig. 5.1. The table of Fig. 5.3 gives the definitions of the cases and the work for each. Recall that all functions elide factors of $\log n$.

THEOREM 5.4. *The work used by the strategy of Algorithm* 5.2 *is as given by Fig.* 5.3.

Proof. Case 1 corresponds to the leftmost leaf of Fig. 5.1. The basis B2 is used, and Lemma 3.1 tells us that $\tilde{O}(d)$ rounds suffice. Since each round takes $O(1)$ time, and the number of processors used is $e\lceil s/d\rceil$, the work is $es + ed$, as mentioned when we introduced rule B2. Since $d \leqq t$ is given, we finish within the time limit.

Case	Condition	Work
1	$d \leqq t,\ d \leqq \sqrt{n}$	$es + ed$
2	$t \geqq \sqrt{n},\ d > \sqrt{n}$	$es + e\sqrt{n}$
3	$d > t,\ t < \sqrt{n},\ s \geqq n/t$	$es + sn^2/t^2$
4	$t < d \leqq t^2,\ t < \sqrt{n},\ s < n/t$	$en/t + sn^2/t^2$
5	$d > t^2,\ t < \sqrt{n},\ s < n/t$	$en/t + sn^2/t^2 + n^3/t^4$

FIG. 5.3. *Work used by Algorithm* 5.2.

Case 2 corresponds to the second node in Fig. 5.1, where we apply R1 with $s_1 = \sqrt{n}$. The two subgoals in R1 reduce to $S(\sqrt{n}, \sqrt{n}, n, e)$ and $S(s, d/\sqrt{n} + \sqrt{n}, n, e + n)$. Since $d \leqq n$ can be assumed, and $n \leqq e$ is a global assumption, the second term can be simplified to $S(s, \sqrt{n}, n, e)$. B2 applied to the first requires $e\sqrt{n}$ work, and B2 applied to the second requires $es + e\sqrt{n}$ work. Since $t \geqq \sqrt{n}$ is assumed in this case, we finish within the time limit.

Case 3 is for the third node of Fig. 5.1, where R2 is applied with $s_1 = n/t$. The terms of R2 simplify in this case to $S(n/t, t, n, e)$, $D(n/t, d/t, n/t)$, and $S(s, t, n, e + n^2/t^2)$. B1 applied to the middle term takes work n^3/t^3 and can be done within any time limit, since it requires only polylog time, and even $t = 1$ really means $t = \tilde{O}(1)$ according to our conventions. The first term, with basis rule B2, requires $en/t + et$, and the third term with B2 requires $(e + n^2/t^2)(s + t)$. If we expand all these terms we get

$$\frac{n^3}{t^3} + \frac{en}{t} + et + es + \frac{sn^2}{t^2} + \frac{n^2}{t}.$$

Because $e \geqq n$, we can eliminate the last term in favor of the second. As $s \geqq n/t$ is assumed in Case 3, we can eliminate the first term in favor of the fifth, and we can eliminate the second in favor of the fourth. Finally, we claim that $s > t$, because $t < \sqrt{n}$ is given. That, with $s \geqq n/t$, gives us $s > \sqrt{n}$, which in turn exceeds t. Thus, the third term, et, can be eliminated in favor of the fourth, es.

Cases 4 and 5 correspond to the last two nodes of Fig. 5.1, where $d > t$, $t < \sqrt{n}$, and $s < n/t$ are assumed, and R3 is used. R3, with $s_1 = n/t$ simplifies to the subgoals $S(n/t, t, n, e)$, $S(s, d/t, n/t, n^2/t^2)$, and $S(s, t, n, e + sn/t)$. B2 applied to the first subgoal yields work $e(n/t + t)$, but since $t < \sqrt{n}$, we know that $n/t > t$, and we can write this work as en/t. B2 applied to the last subproblem requires work $(e + sn/t)(s + t)$.

In Case 4, where $d \leqq t^2$, that is, $d/t \leqq t$, there is enough time to apply B2 to the middle subgoal. Thus, the work for that subgoal is $n^2(s+d/t)/t^2$. These expressions, expanded out, are

$$\frac{en}{t}+es+et+\frac{s^2n}{t}+sn+\frac{sn^2}{t^2}+\frac{dn^2}{t^3}. \tag{5.5}$$

Remembering that $n \leqq e$ and $s < n/t$ eliminates the second and fifth terms in favor of the first. Again reasoning that $t < n/t$ whenever $t < \sqrt{n}$ eliminates the third term in favor of the first. In Case 4, we have $d/t \leqq t$, so we can eliminate the seventh term by noting that $dn^2/t^3 \leqq n^2/t$. Then we use $n \leqq e$ to eliminate the last term in favor of the first. Finally, we use the fact that $s < n/t$ to eliminate the fourth term in favor of the sixth. We are thus left with only the first and sixth terms, $en/t + sn^2/t^2$.

In Case 5, we have $d > t^2$, so we use R2 with $s_1 = n/t^2$ on the middle subgoal. The first and last subproblems when we initially expanded $S(s, d, n, e)$ by R3 yield the first five terms of (5.5), which by the reasoning above simplifies to $en/t + s^2n/t$. The middle term $D(s, d/t, n/t)$, which stands for $S(s, d/t, n/t, n^2/t^2)$, when expanded by R2 yields the subproblems $S(n/t^2, t, n/t, n^2/t^2)$, $S(n/t^2, d/t^2, n/t^2, n^2/t^4)$, and $S(s, t, n/t, n^2/t^2)$. Note that we can neglect the "$+s_1^2$" term in the number of edges of the last subproblem, because the graph is already dense. If we apply B2 to the first, we get work $n^3/t^4 + n^2/t$; B1 applied to the second uses work n^3/t^6; and B2 on the third uses $sn^2/t^2 + n^2/t$. Thus, all the terms in the expression for work are

$$\frac{en}{t}+\frac{s^2n}{t}+\frac{n^3}{t^4}+\frac{n^2}{t}+\frac{n^3}{t^6}+\frac{sn^2}{t^2}.$$

Evidently, the fifth term can be eliminated in favor of the third, and the fourth in favor of the first. The second can be eliminated in favor of the sixth, because $s < n/t$ is assumed in Case 5. Thus, the remaining terms are $en/t + sn^2/t^2 + n^3/t^4$, as stated in Fig. 5.3. □

A Las Vegas algorithm. We can modify Algorithm 5.2 to check the validity of its answer, using $O(es)$ work and $\tilde{O}(1)$ time. As we shall see in the next section, the work required by Algorithm 5.2 is $\tilde{\Omega}(es)$ in any of the five cases of Fig. 5.3; thus the additional work and time used by this modification can be neglected. Suppose we have a set of nodes X 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 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 some source v, would have an arc to a node not in X.

We can thus modify Algorithm 5.2 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 as given in Fig. 5.3.

6. Optimality of the general algorithm.

We shall now show that Algorithm 5.2 is the best we can do, given the tools at hand—recursive rules R1 through R4 and basis rules B1 and B2. First, define an *instance* of the transitive-closure problem to be a 4-tuple $I = (s, d, n, e)$, representing the arguments of the problem $S(s, d, n, e)$. Then, define a *strategy* for solving instance I with time limit t to be a tree with the following properties.

(1) Every node is labeled by an instance.

(2) The root is labeled I.

(3) Each interior node is also labeled by a recursive rule, R1, R2, R3, or R4. Optionally, we attach the value of the key parameter, s_1 or k, to help the reader follow what is going on.

(4) Each leaf is additionally labeled by a basis rule, B1 or B2.

(5) The interior nodes make sense, in that if interior node v is labeled by instance J and rule Ri, then the children of v are labeled by the instance(s) found on the right of the arrow in the definition of Ri, as given in § 4.

(6) The leaves make sense, in that if a leaf is labeled by instance (s, d, n, e) and rule B2, then $d \leqq t$. There is no similar constraint for leaves labeled B1, since that algorithm can be applied in polylog time.

Example 6.1. Suppose $I = (1, n, n, e)$, that is, single-source, sparse transitive-closure. Let $t = n^{1/3}$. The solution provided by Algorithm 5.2 is the tree of Fig. 6.2, since Case 5, as enumerated in Fig. 5.3, applies. Note that we have simplified expressions in some of the instances by dropping terms that are dominated by other terms. For example, at the rightmost child of the root, the fourth argument, $e + n^{2/3}$, has been simplified to e.

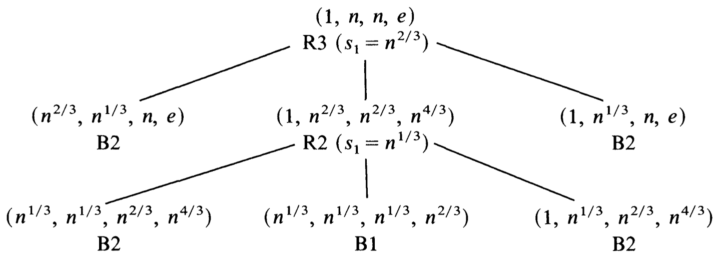
$(1, n, n, e)$
R3 $(s_1 = n^{2/3})$
$(n^{2/3}, n^{1/3}, n, e)$
B2
$(1, n^{2/3}, n^{2/3}, n^{4/3})$
R2 $(s_1 = n^{1/3})$
$(1, n^{1/3}, n, e)$
B2
$(n^{1/3}, n^{1/3}, n^{2/3}, n^{4/3})$
B2
$(n^{1/3}, n^{1/3}, n^{1/3}, n^{2/3})$
B1
$(1, n^{1/3}, n^{2/3}, n^{4/3})$
B2

FIG. 6.2. *Solution for single-source transitive-closure with* $t = n^{1/3}$.

The *cost* of a solution is defined recursively, up the tree. As in previous sections, costs elide factors of log n. For a leaf, we compute the work for the associated basis rule, as given in § 4: n^3 for B1 and $es + ed$ for B2. The cost for an interior node is the sum of the costs of its children, if R1, R2, or R3 is used, and k times the cost of its lone child if R4 is used with parameter k.[7]

The following function $f(I, t)$ summarizes the cost of the strategy generated by Algorithm 5.2, as we shall prove. 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. We shall show that $f(I, t)$ is a lower bound on the cost of any strategy for $I = (s, d, n, e)$ with time limit t. First, we show that $f(I, t)$ accurately reflects the cost of the strategy generated by Algorithm 5.2.

[7] We shall see that R4 can never be used to advantage, but since it is a plausible strategy, we need to consider its use.

LEMMA 6.3. *The work of Algorithm* 5.2'*s strategy, as given in Fig.* 5.3, *is equal to that of the function $f(I, t)$ above.*

Proof. We divide the proof according to the relationship between d and t.

Case A. $d \leqq t$. Then $\alpha_1 = \alpha_2 = 0$, and $f(I, t)$ is $es + e \min(d, \sqrt{n})$. Also, only Cases 1 and 2 of Fig. 5.3 can apply. In Case 1, where $d \leqq \sqrt{n}$, $\min(d, \sqrt{n}) = d$, so $f(I, t) = es + ed$; this formula is exactly that given by Fig. 5.3. In Case 2, $d > \sqrt{n}$, so $f(I, t) = es + e\sqrt{n}$, also as in Fig. 5.3.

Case B. $t < d \leqq t^2$. Then $\alpha_1 = 1$, but $\alpha_2 = 0$. Only Cases 2–4 of Fig. 5.3 can apply. In Case 2, $d > \sqrt{n}$, so $f(I, t)$ is

$$es + e\sqrt{n} + en/t + sn^2/t^2.$$

Since $t \geqq \sqrt{n}$ in Case 2, $en/t \leqq e\sqrt{n}$, so the second term dominates the third. Also, $sn^2/t^2 \leqq sn$, so the first term dominates the fourth. Thus, the above formula for $f(I, t)$ reduces to $es + e\sqrt{n}$, as given for Case 2 in Fig. 5.3.

In Cases 3 and 4, we do not know how d and $\sqrt{n}$ compare, so we write $f(I, t)$ as

$$es + e \min(d, \sqrt{n}) + en/t + sn^2/t^2.$$

In Case 3, we have $s \geqq n/t$, so the first term dominates the third. Also, $t < \sqrt{n}$, so $s \geqq n/t > \sqrt{n}$. Hence, $es > e\sqrt{n}$, and the first term dominates the second. Thus, $f(I, t)$ simplifies to the formula for Case 3 in Fig. 5.3.

Finally, consider Case 4. There, $s < n/t$, so the third term dominates the first. Also, $t < \sqrt{n}$, so $en/t > e\sqrt{n}$, and the third term dominates the second. Thus, $f(I, t)$ reduces to the formula given by Case 4 of Fig. 5.3.

Case C. $d > t^2$. Then $\alpha_1 = \alpha_2 = 1$, and $f(I, t)$ is

$$es + e \min(d, \sqrt{n}) + en/t + sn^2/t^2 + n^3/t^4.$$

Here, only Cases 3 or 5 of Fig. 5.3 can apply. If Case 3 of Fig. 5.3 applies, we can argue as in Case B above that the first term dominates the second and third. Also, since $s \geqq n/t$, the fourth term is at least n^3/t^3, and so dominates the fifth term. Thus, $f(I, t)$ simplifies to the formula of Case 3 in Fig. 5.3, that is, the first and fourth terms, above.

If Case 5 holds, then $s < n/t$ and $t < \sqrt{n}$ tell us that the third term dominates the first and second, which reduces the above formula to that of Case 5 in Fig. 5.3. □

LEMMA 6.4. *Without loss of generality, we may assume that the optimal strategy does not use* R4.

Proof. Note that $f(I, t)$ never grows more than linearly with s. Each term is either independent of s or linear in s. In particular, α_1 and α_2 are independent of s, and so cannot, by their discontinuity, make $\partial f/\partial s$ exceed one. Thus, if a strategy uses R4 with parameter k at an interior node v, and that node has child with label $I = (s, d, n, e)$, we can eliminate that child and multiply the number of sources by k in each descendant whose number of sources depends on s. Note that the validity of a strategy can only depend on the relationship between the distance and the time limit, so the strategy cannot be rendered invalid by this transformation. We have multiplied the cost of each descendant of v by at most k, and thus the cost of v itself has not increased, since we no longer have to multiply the cost of v's child by k to get the cost of v. □

THEOREM 6.5. *The cost of any strategy for solving $I(s, d, n, e)$ in time t is $\tilde{\Omega}(f(I, t))$.*

Proof. By Lemma 6.4 we may assume that the strategy does not use R4. We proceed by induction on the height of the tree T for the strategy in question.

Basis. If the tree T is a leaf labeled B1, then the cost of the strategy is n^3, and we note that no term of $f(I, t)$ exceeds n^3. If the leaf is labeled B2, then we must have

$d \leqq t$. Thus, $\alpha_1 = \alpha_2 = 0$, and $f(I, t) = es + e \min(d, \sqrt{n})$. The cost of T is $es + ed$, which cannot be less than $f(I, t)$.

Induction. The root of T is labeled by R1, R2, or R3. We shall consider each of the five possible terms of $f(I, t)$: es, $e \min(d, \sqrt{n})$, en/t, sn^2/t^2, and n^3/t^4, and show that, when that term is present in the formula for f (i.e., the appropriate α is one, if necessary), then one of the subproblems for the rule used also has at least that term.

(1) Term es. Each of the rules R1, R2, and R3 contains on the right an instance with s sources and at least e arcs. By the inductive hypothesis, the cost of that subproblem is bounded below by f, and so has a term $e's$ for some $e' \geqq e$.

(2) Term $e \min(d, \sqrt{n})$. On the right-hand side of each recursive rule, the first subproblem has s_1 sources, distance n/s_1, n nodes, and e arcs. By the induction hypothesis, the cost of this subproblem is at least $es_1 + e \min(n/s_1, \sqrt{n})$, which is at least $e\sqrt{n}$, because either $s_1 \geqq \sqrt{n}$ or $n/s_1 \geqq \sqrt{n}$.

(3) Term en/t. Here, we may assume $\alpha_1 = 1$, that is, $d > t$. We consider the first subproblem of each rule, which is $(s_1, n/s_1, n, e)$, and branch depending on whether $n/s_1 \leqq t$.

  (a) $n/s_1 \leqq t$. Then the cost of the first subproblem includes, by the inductive hypothesis, a term es_1. Since $n/s_1 \leqq t$, this term is at least en/t.

  (b) $n/s_1 > t$. Since n/s_1 is the distance in the first subproblem, the value of f for that subproblem has $\alpha_1 = 1$, and thus includes term en/t.

(4) Term sn^2/t^2. We distinguish the same two subcases as in (3). In Case (a), where $n/s_1 \leqq t$, note that each rule has a subproblem with s sources and at least s_1^2 arcs; this subproblem is the last in R1 and R2, and the second in R3. The cost for this subinstance includes a term $s_1^2 s$, which is at least sn^2/t^2, if Case (a) holds. In Case (b), the last subproblem of each rule has s sources, n nodes, and distance at least n/s_1, which exceeds t in Case (b). Thus, $\alpha_1 = 1$ for this subproblem, and its cost includes a term sn^2/t^2 by the inductive hypothesis.

(5) Term n^3/t^4. Here, we may assume $\alpha_1 = \alpha_2 = 1$. There are three subcases, depending on how n/s_1 compares with t and t^2.

  (a) If $n/s_1 > t^2$, then for the first subproblem of each rule, the value of α_2 is one. Thus, the cost of this subproblem includes term n^3/t^4 by the inductive hypothesis.

  (b) If $t < n/s_1 \leqq t^2$, then α_1 for the first subproblem is one, and the cost of that subproblem includes a term $s_1 n^2/t^2$. Since $n/s_1 \leqq t^2$ is assumed, we have $s_1 \geqq n/t^2$, and therefore $s_1 n^2/t^2 \geqq n^3/t^4$.

  (c) If $n/s_1 \leqq t$, then $ds_1/n \geqq d/t > t$; the latter inequality follows because $\alpha_2 = 1$ for the instance at hand, or else we would not have to deal with the term n^3/t^4 at all. For each of rules R1, R2, and R3, the second subproblem, call it $S(s', d', n', e')$, satisfies $n' \geqq s_1$, $e' \geqq s_1^2$, and $d' \geqq ds_1/n > t$. By the latter, the α_1 bit for this subproblem is one, and its cost includes the term $e'n'/t \geqq s_1^3/t \geqq n^3/t^4$. □

COROLLARY 6.6. *The strategy of Algorithm* 5.2 *is optimal.*

Proof. The cost of that strategy is exactly that given by $f(I, t)$, as we learned from Theorem 5.4 and Lemma 6.3. □

7. Important special cases.

The most important instances of the general transitive closure are:

(1) $S(1, n, n, e)$, or sparse, single-source.

(2) $S(n, n, n, e)$, or sparse, all-pairs.

(3) $S(1, n, n, n^2) = D(1, n, n)$, or dense, single-source.

(4) $S(n, n, n, n^2) = D(n, n, n)$, or dense, all pairs.

All these have d, the distance, equal to n, the number of nodes, and have s, the number of sources, equal to one or n.

All-pairs problems. When we let $s = d = n$ in the formula $f(I, t)$ of the previous section, we are left with the expression

$$en + e\sqrt{n} + \alpha_1(en/t + n^3/t^2) + \alpha_2 n^3/t^4.$$

Also, we note that since $d = n$, $\alpha_1 = 1$ as long as $t < n$, which is the interesting case, and $\alpha_2 = 1$ whenever $t < \sqrt{n}$. We may therefore drop the last term, $\alpha_2 n^3/t^4$, since whenever it is nonzero, it is dominated by the term $\alpha_1 n^3/t^2$. Moreover, the second and third terms are clearly dominated by the first, so we can write

$$f(n, n, n, e, t) = en + n^3/t^2. \tag{7.1}$$

Note that we do not need the factor α_1 on the second term, since whenever $\alpha_1 = 0$, n^3/t^2 is no larger than n, and therefore is dominated by the first term anyway. Figure 7.2(a) gives the strategy for the all-pairs case, when $t \geqq \sqrt{n}$; here, Case 2 of Fig. 5.3 applies, and the cost is en. Figure 7.2(b) shows the strategy when $t < \sqrt{n}$; here Case 3 of Fig. 5.3 pertains.

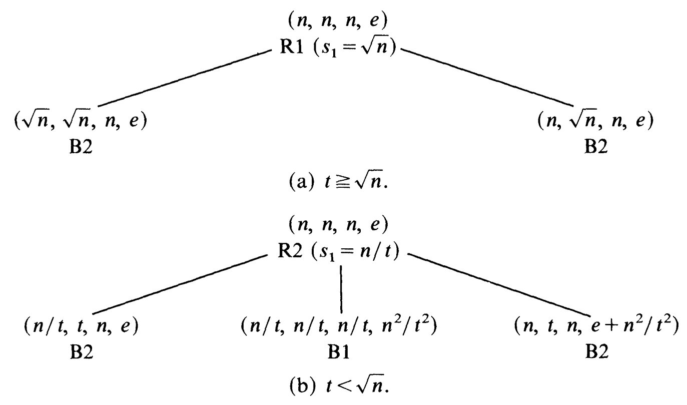

FIG. 7.2. *Strategies for the all-pairs problem.*

A useful simplification is to assume that $t = n^\varepsilon$ for some small ε, and $e = n^\alpha$, for some α between one and two. In (7.1), the en term dominates as long as $\alpha \geqq 2 - 2\varepsilon$. We can thus solve the all-pairs problem optimally, as long as the graph is not too sparse. For time $\sqrt{n}$, that is, $\varepsilon = \frac{1}{2}$, 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.

The dense, all-pairs problem. In the dense case, where $e = n^2$, (7.1) reduces to n^3. That is, our algorithms offer no help for the dense, all-pairs problem. In that case, the basis rule B1 is as good a strategy as we know.

Single-source problems. Now, let $s = 1$ and $d = n$ in the formula $f(I, t)$. The resulting formula is

$$e + e\sqrt{n} + \alpha_1(en/t + n^2/t^2) + \alpha_2 n^3/t^4.$$

Again, $\alpha_1 = 1$ as long as $t < n$, and $\alpha_2 = 1$ whenever $t < \sqrt{n}$. Now, note that n^2/t^2 can never exceed en/t, so we may eliminate the fourth term. The first term is clearly dominated by the second. We can thus write

$$f(1, n, n, e, t) = e\sqrt{n} + en/t + n^3/t^4. \tag{7.3}$$

We do not need multipliers α_1 or α_2 on the last two terms, because when one of the α's is zero, its term is always dominated by the first term.

Figure 7.4(a) shows the strategy of Algorithm 5.2 for the single-source problem when $t \geqq \sqrt{n}$. It represents an alternative to Algorithm 2.3, which was defined only for the case $t = \sqrt{n}$, but evidently works for larger t.

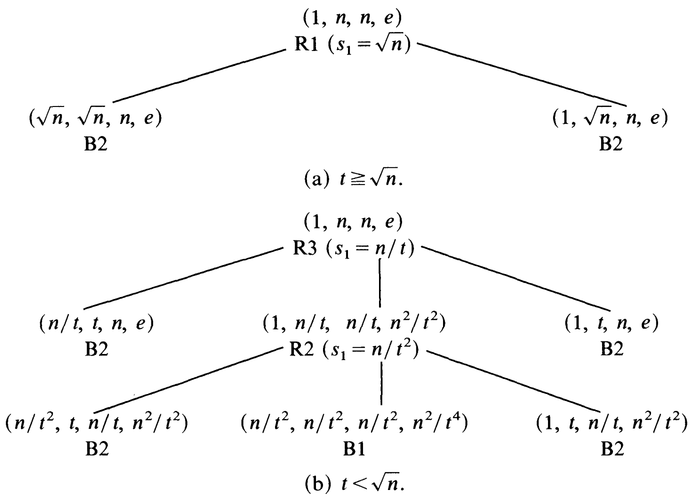

FIG. 7.4. *Strategies for the single-source problem.*

If we assume $t = n^\varepsilon$ for $\varepsilon \leqq \frac{1}{2}$, then the $e\sqrt{n}$ term in (7.3) can be dropped. The term en/t dominates as long as $e = n^\alpha$, and $\alpha \geqq 2 - 3\varepsilon$. Put another way, we can solve the problem with time $t = \tilde{O}(n^\varepsilon)$ and work $\tilde{O}(en^{1-\varepsilon})$ (i.e., $en^{1-2\varepsilon}$ processors), as long as $\varepsilon \geqq (2-\alpha)/3$. 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.

The dense, single-source problem. For the dense case, (7.3) reduces to $n^{2.5} + n^3/t$, or just n^3/t if we assume that only $t \leqq \sqrt{n}$ 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 are willing to take, up to $t = \sqrt{n}$, the more improvement in the total work we achieve. Note, 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.

8. 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. Recall that the single-source BFS (breadth-first-search) problem of Gazit and Miller [1988] is to find, for each node v, the least number of arcs on the paths from the source node to v. Alternatively, we can view BFS as the special case of the shortest-path problem when the arcs all have unit length. We can solve the single-source BFS problem in the time given by (7.3); it is not known whether the same holds for the all-sources BFS problem and formula (7.1).

We shall discuss only the hard (and interesting) case where $t \leqq \sqrt{n}$. The strategy we use is similar to that illustrated in Fig. 7.4(b) for single-source transitive closure. First, we select $\tilde{O}(n/t)$ "distinguished nodes" and explore distance t from them. Then, we select from among the distinguished nodes $\tilde{O}(n/t^2)$ "superdistinguished nodes," and solve a restricted search problem from these. There are sufficiently few superdistinguished nodes that we can compute all shortest paths among them with work $(n/t^2)^3 = n^3/t^6$, by conventional means.[8]

As we remarked in § 2, the technique for step (2) of Algorithm 2.3 is not essential, except to save a factor of log n. We can instead use e/t processors to search forward from any one distinguished node for distance t in time $\tilde{O}(t)$, as follows. The algorithm is essentially a breadth-first-search, in which we divide the graph into layers according to the length of the shortest path from the given source. Suppose we have constructed the first i layers and marked all nodes that have been reached.

(1) The arcs out of the nodes on layer i are divided evenly among the e/t processors.

(2) Each processor finds the unmarked nodes reached by these arcs, and marks them as belonging to layer $i+1$. The reached nodes become layer $i+1$.

(3) A list of the nodes on layer $i+1$ is made, eliminating duplicates, and a count of the arcs out of this layer is made, to facilitate step (1) for layer $i+1$.

Starting with only the given distinguished node in layer zero, we can construct layers up to t.

LEMMA 8.1. *The above construction correctly computes the length of the shortest path from the given source w to each node that is at distance at most t from w. It takes parallel time $\tilde{O}(t)$.*

Proof. The correctness should be evident. Subtlety is in the running time. Intuitively, each arc is explored only once, so while we may spend a lot of time processing one layer, the total time spent on all the layers is $\tilde{O}(t)$. Let e_i be the number of arcs out of the nodes at layer i, and note that $\sum_{i=0}^{t} e_i \leqq e$. The time spent processing layer i is easily seen to be $\tilde{O}(e_i t/e)$; the principal costs are exploring the arcs in step (2) and sorting in step (3). But

$$\sum_{i=0}^{t} e_i t/e \leqq (t/e) \sum_{i=0}^{t} e_i \leqq t.$$

Thus, the total time over all layers is $\tilde{O}(t)$. □

Note that the total work performed by the algorithm outlined above is $\tilde{O}(en/t)$. That is, there are $\tilde{O}(n/t)$ distinguished nodes, with e/t processors assigned to each. The total number of processors, en/t^2, is multiplied by the time, $\tilde{O}(t)$, to get the total work.

Search from superdistinguished nodes. If we pick $\tilde{O}(n/t)$ distinguished nodes at random, and include the single source among them, we can use the above algorithm to search from each distinguished node with work $\tilde{O}(en/t)$ and time $\tilde{O}(t)$. The result

[8] Note that we cannot do so for the graph of the distinguished nodes, because that would take $(n/t)^3$ work, which is larger than the n^3/t^4 term of (7.3). However, one more level, to the superdistinguished nodes, gets us down to work n^3/t^6, which can be neglected when compared with the n^3/t^4 term.

will tell us all the paths from one distinguished node to another, provided that path is of length t or less. More formally, if we have discovered that searching from distinguished node w_1, we can reach distinguished node w_2, and the layer of w_2 with respect to w_1 is i, then $i \leqq t$, and i is truly the length of a shortest path from w_1 to w_2. Conversely, if we have not discovered a path from w_1 to w_2, then the length of a shortest path from w_1 to w_2, if it exists, is greater than t.

We could use a conventional path-doubling-by-matrix-multiplication algorithm to find the lengths of the shortest paths between each pair of distinguished nodes, in the graph of distinguished nodes whose arcs are weighted by the lengths of the paths found between nearby distinguished nodes. However, that would take work $\tilde{O}(n^3/t^3)$, and we need to do better, specifically $\tilde{O}(n^3/t^4)$.

Thus, we construct the graph H of distinguished nodes with weighted arcs representing shortest paths up to length t, but before proceeding, we select at random $\tilde{O}(n/t^2)$ superdistinguished nodes for this graph. We also include the original source node among the superdistinguished nodes. We shall compute the lengths of shortest paths between every pair of superdistinguished nodes first. Note that arcs in H have weights up to t. "Shortest" refers to weighted path length, not to the number of arcs. However, paths in H represent paths in the original graph, and the weighted path length in H equals the length (number of arcs) on the corresponding paths of the original graph.

The idea is illustrated in Fig. 8.2. We start from some superdistinguished node x, and look at a shortest path in H from x to some (distinguished) node y. We divide the nodes along the path, all of which are distinguished, into layers according to the length of their shortest path from x; the layers each have width t; that is, the first layer consists of nodes with shortest paths from x of length up to t, the next of nodes with shortest paths from $t+1$ to $2t$, and so on. There are n/t layers, since each path among the distinguished nodes represents a path in the original graph of n nodes, and therefore has length at most n. However, we shall only want to consider the first t layers, and thus discover paths of length up to t^2 in the original graph; as we shall see, these paths are represented by paths with at most $2t$ arcs in the graph H.

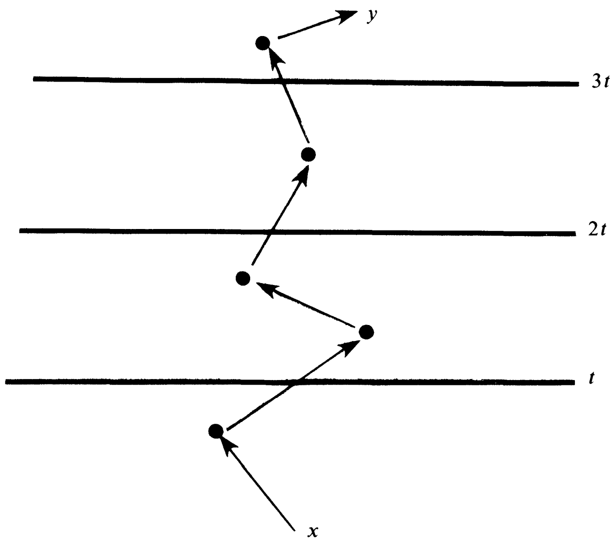

FIG. 8.2. *Shortest weighted path from x to y.*

When we performed a breadth-first-search in the original graph of n nodes, we could proceed layer by layer, since the arcs were unweighted, and each layer could be constructed directly from the one below. We are not so fortunate in the graph of distinguished nodes, since now arcs have weights up to t, and even if we regard layers as having width t, as suggested by Fig. 8.2, we cannot compute each layer directly from the one below. Fortunately, we can do so in two passes, as the next lemma argues.

LEMMA 8.3. *In the graph of distinguished nodes, a shortest path from x to y may be assumed not to have three consecutive nodes in one layer.*

Proof. Suppose w_1, w_2, and w_3 are three consecutive (distinguished) nodes belonging to the same layer, and lying on a shortest path from x to y. Then the path from w_1 to w_3 is not longer than t, and by Lemma 8.1, the search for distance t from distinguished node w_1 in the original graph reached w_3. Thus, H has an arc $w_1 \rightarrow w_3$, whose weight cannot be greater than the sum of the weights of the arcs $w_1 \rightarrow w_2$ and $w_2 \rightarrow w_3$, since all such arcs are weighted with true shortest path lengths. It follows that we could have eliminated w_2 from the shortest path from x to y. □

Thus, we can build the shortest-path information layer by layer, if we use two passes per layer, one to discover nodes of that layer whose predecessor on the shortest path from a given superdistinguished node is in the previous layer, and a second to discover nodes with one previous node on the shortest path that is in the same layer. Of course, no node can have a shortest path where the previous node is not in the same or next-lower layer, because arc weights are limited by t.

(1) For each node w in layer i, whose distance from superdistinguished node x has been calculated to be j (thus, $(i-1)t<j\leqq it$), consider each arc in the graph of distinguished nodes out of w. If j plus the weight d of that arc exceeds it, the arc enters node u, and u is not in a layer below $i+1$, then generate a fact that there is a path from x to u of length $j+d$. Note that $j+d\leqq(i+1)t$, so u belongs to layer $i+1$.

(2) Sort the generated facts by target node, and eliminate distances for each node other than the minimum. Note that the resulting distance for a node u in layer $i+1$ may not yet be minimum, since there may be another node in layer $i+1$ that the shortest path from x to u goes through. However, if u is a node of layer $i+1$ such that the shortest path from x to some other node v in layer $i+1$ goes through u, then the shortest path to u is already correct, or else the shortest path from x to v goes through three nodes in layer $i+1$, contradicting Lemma 8.3.

(3) Examine each node u already in layer $i+1$; say the current estimate of the length of the shortest path from x to u is k. For each arc out of u, of weight c and reaching v, generate the fact that there is a path from x to v of weight $k+c$, provided $k+c\leqq(i+1)t$.

(4) Again sort all the facts, including those remaining from step (2), according to their target node, and for each target node select the shortest path for that node. The result is that all nodes of layer $i+1$ are found and have the length of their shortest path from x calculated.

The complete algorithm for searching from each of the $\tilde{O}(n/t^2)$ superdistinguished nodes for weighted distance t^2 is to use n^2/t^3 processors for each of the $\tilde{O}(n/t^2)$ superdistinguished nodes, and use them to perform the above search for t layers. Thus, all paths between superdistinguished nodes of length at most t^2 are found. Note that the length t^2 can be thought of either as weighted path length in H, or as number of arcs in the original graph.

LEMMA 8.4. *The above steps correctly compute all shortest paths among superdistinguished nodes, provided those paths are of length at most t^2. Furthermore, the time taken is $\tilde{O}(t)$, and the total work is $\tilde{O}(n^3/t^4)$.*

Proof. Correctness follows from Lemma 8.3. For now, note that there are $\tilde{O}(n/t)$ nodes in the graph of distinguished nodes, and therefore at most $\tilde{O}(n^2/t^2)$ arcs. Each arc is considered at most twice, in steps (1) and (3). Thus, by an argument similar to that of Lemma 8.1, we can conclude that the time taken by the n^2/t^3 processors assigned to each superdistinguished node as a source, summed over each of the t layers, is at most $\tilde{O}(t)$. Since there are $\tilde{O}(n/t^2)$ superdistinguished nodes, the total number of processors is n^3/t^5, and when we multiply this figure by the time taken, we get total work of $\tilde{O}(n^3/t^4)$. □

The complete algorithm. We can now put together the ideas summarized in Lemmas 8.1 and 8.4 to get an algorithm that uses work $en/t+n^3/t^4$.[9]

ALGORITHM 8.5. 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 $t \leqq \sqrt{n}$.
OUTPUT: For each node v in G, the length of (number of arcs on) a 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, by the technique outlined prior to Lemma 8.1.
(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 a shortest path from u to v is of length $d \leqq t$.
(4) Select $\tilde{O}(n/t^2)$ superdistinguished nodes in H, and include v_0 among them, whether or not it was picked.
(5) Explore H from each superdistinguished node for t layers (weighted distance t^2), using the technique outlined prior to Lemma 8.4.
(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 a 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.

THEOREM 8.6. *Algorithm* 8.5 *is correct with high probability.*

Proof. If we pick $(cn \log n)/t$ distinguished nodes, for sufficiently high c, we can make the probability as high as we like that there is a distinguished node within

[9] As mentioned, we assume $t \leqq \sqrt{n}$. Thus, the term $e\sqrt{n}$ in (7.3) can be neglected.

every t nodes along a shortest path from v_0 to any node. Likewise, by picking $(cn \log n)/t^2$ superdistinguished nodes, we can assume with high probability that there is a superdistinguished node at least every t^2 nodes along every shortest path. Consequently, with high probability, the transitive-closure J computed at step (6) is correct, and at step (7), we correctly have the length of a shortest path from v_0 to every superdistinguished node. We may regard any shortest path as traveling from v_0 to some superdistinguished node (perhaps v_0 itself), then through a distance less than t^1, through at most $2t$ distinguished nodes, and finally through nondistinguished nodes for a distance of at most t. Step (7) takes us from the last superdistinguished node to the last distinguished node, and step (8) takes us the rest of the way. □

THEOREM 8.7. *Algorithm* 8.5 *takes time* $\tilde{O}(t)$ *and work* $\tilde{O}(en/t+n^3/t^4)$.

Proof. It is easy to check that each step can be done within $\tilde{O}(t)$ time. Steps (1) and (4) can be done with $\tilde{O}(n)$ work, as discussed in connection with Theorem 3.4. The work of step (2) is $\tilde{O}(en/t)$ by Lemma 8.1. Step (3) is easily seen to take $\tilde{O}(n^2/t^2)$ work. Step (5) requires $\tilde{O}(n^3/t^4)$ work by Lemma 8.4. Step (6) takes $\tilde{O}(n^3/t^6)$ work, step (7) uses $\tilde{O}(n^2/t^3)$, and step (8) requires $\tilde{O}(n^2/t)$, as discussed in Algorithm 8.5. It is easy to check that each of these terms is dominated by either en/t or n^3/t^4, assuming $t \leqq \sqrt{n}$. □

9. Open problems.

We may 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? Note that work $en+n^3/t$ is achievable, which is better than the obvious algorithm, but does not match the bound of (7.1).

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

Acknowledgments. We were helped by discussions concerning 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

A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN [1974], *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

P. A. BLONIARZ, M. J. FISCHER, AND A. R. MEYER [1976], *A note on the average time to compute transitive closure*, 3rd Internat. Colloquium on Automata, Languages, and Programming, Edinburgh, U.K., Springer-Verlag, Berlin, New York.

R. P. BRENT [1974], *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21, pp. 201-208.

A. Z. BRODER, A. R. KARLIN, P. RAGHAVAN, AND E. UPFAL [1989], *Trading space for time in undirected s-t connectivity*, in Proc. 21st annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York.

R. COLE AND U. VISHKIN [1986], *Approximate and exact parallel scheduling with applications to list, tree, and graph problems*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 478-491.

D. Coppersmith and S. Winograd [1987], *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, pp. 1-6.

H. Gazit [1986], *An optimal randomized parallel algorithm for finding connected components in a graph*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 492-501.

H. Gazit and G. L. Miller [1988], *An improved parallel algorithm that computes the* BFS *numbering of a directed graph*, Inform. Process. Lett., 28, pp. 61-65.

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

C. P. Schnorr [1978], *An algorithm for transitive closure with linear expected time*, SIAM J. Comput., 7, pp. 127-133.

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

K. Simon [1986], *An improved algorithm for transitive closure on acyclic digraphs*, 13th Internat. Colloquium on Automata, Languages, and Programming, Rennes, France, Springer-Verlag, Berlin, New York, pp. 376-387.