1985

A Distributed Shortest Path Algorithm for a Planar Network

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

Report Number:
85-527

A DISTRIBUTED SHORTEST PATH ALGORITHM FOR A PLANAR NETWORK*

Greg N. Frederickson
CSD TR 527 Revised July 1988

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

* This research was supported in part by the National Science Foundation under Grants MCS-8201083, DCR-8320124 and CCR-8620271, and by the Office of Naval Research under contract N00014-86-K-0689.

Abstract.

An algorithm is presented for finding a single source shortest path tree in a planar undirected distributed network with nonnegative edge costs. The number of messages used by the algorithm is $O(n^{5/3})$ on an n-node network. Distributed algorithms are also presented for finding a breadth-first spanning tree in a general network, for finding a shortest path tree in a general network, for finding a separator of a planar network, and for finding a division of a planar network.

Key words and phrases. breadth-first search, distributed network, message complexity, planar network, planar separator, single source shortest paths.

1. Introduction

Consider a problem in which the information necessary for its solution is distributed among the nodes of a network. A fundamental question in distributed computation is how to solve the problem, using a minimum number of messages to route the information. In particular, suppose that the problem is a graph problem about the network itself, in which initially each node has knowledge only about its neighbors. An algorithm could always route all information to a particular node and then solve the problem directly. But this approach would use $O(mn)$ messages, where n is the number of nodes and m the number of edges. Certain problems can be handled more efficiently, as for example that of finding a minimum spanning tree of the network, which can be done with only $O(m + n \log n)$ messages [GHS]. In this paper we present communication-efficient algorithms for several other basic graph problems, including finding a shortest path tree in a distributed network.

Several papers [AR, Fm, T] have investigated the message complexity for the all pairs shortest paths problem, with the best solution requiring $O(mn)$ messages. As far as the message complexity, this is no better than the straightforward approach mentioned above. We concentrate on the single source problem in an undirected network with nonnegative edge weights, and present two efficient algorithms for this problem. The first generates a single source shortest path tree in a general network, using $O(n^2)$ messages. Our main, and more interesting, result concerns the case in which the network is planar, for which we give an algorithm that uses $O(n^{5/3})$ messages. In achieving this bound we also solve three problems of independent interest. First we present a simple algorithm to find a breadth-first spanning tree of a general network, using $O(n\, m^{1/2})$ messages. This algorithm uses $O(n^{3/2})$ messages on a planar network,

since m is $O(n)$ for such a network. A previous algorithm in [G] uses $O(n^{8/5} + n^{2/3}m^{2/3})$ messages, and more recent algorithms use $O(n^{8/5}\log n + m)$ and $O(m2^{\sqrt{\log n \log\log n}})$ messages [AG1, AG2]. Second, we present a distributed algorithm that finds a separator of a planar network using $O(n \log n)$ messages if a breadth-first tree is already given. Third, we present a distributed algorithm that finds a division of a planar network into regions satisfying a size bound on each region and a size bound on the total number of nodes shared by regions.

We make the following assumptions about our model. A message will carry a constant number of "words" along one link of the network. In particular, a message contains the name of one node and/or one number representing the sum of the costs of edges of some simple path in the network. Computation time at a node will be assumed to be small in comparison with message transmission time, and thus will be ignored. Each processor will have a sufficiently large memory so that message buffering will not cause problems. Arbitrarily long delays can be encountered in the processing of a message by a node. However, no messages are lost, communication is error-free, and messages are handled in a first-in first-out fashion.

Our algorithms were designed with the goal of reducing the number of messages. However we also analyze the time performance of the algorithms. We define time as the length of the longest sequence of messages, where each message in the sequence cannot be sent until the predecessor in the sequence has been received. Here we assume that messages can be simultaneously received and sent from different input/output ports at the same time. Thus this measure of time will correspond to the time used by the algorithm if every message transmission is completed in unit time. In all of our algorithms except the one for finding a divison of the network into regions,

the time complexity is the same as the message complexity.

A preliminary version of this paper appeared in [Fs2].

2. Simple algorithm for finding a breadth-first search tree

We first sketch a natural way to generate a breadth-first search tree using $O(n^2)$ messages and time, and then modify it to give an algorithm which uses $O(n\,m^{1/2})$ messages and time. The simplest way to generate a breadth-first tree is one level at a time, so that every node on level i must be identified before attempting to identify any node on level $i+1$. Initially $level(s)$ is 0, where s is the root, and $level(v)$ is n for every other vertex v. The *current frontier* will be the set of all nodes with highest level number less than n. Initially the current frontier will contain just the root. The search is synchronized by the root, using edges in the current portion of the breadth-first tree. The computation consists of phases, each involving three activities: 1) a *broadcast* from the root to the nodes at the current frontier, 2) the *exploration* carried out from nodes at the frontier, and 3) the *echo*, which notifies the root that the exploration is complete.

Let f be the level number of nodes at the current frontier. The root initiates the broadcast by sending a *forward*(f) message to each of its children in the current portion of the breadth-first tree. When a node at level less than f in the tree receives a *forward*(f) message, it sends a *forward*(f) message to each of its children.

Exploration is performed as follows. When a node v at level f receives a *forward*(f) message, it sends an *explore*($f+1$) message to each adjacent node w, except its parent in the breadth-first tree. Node v assumes that each such w is its child in the breadth-first tree. The first *explore* message received by a node w determines its parent. In this case a *reverse* message is sent by w back to its parent v. For each

additional *explore* message received by w, it sends back a *negative* message to the sender. A node receiving a *negative* message removes the sender from its list of children.

The echo is handled as follows. Each node that receives an *explore* $(f+1)$ message will have sent either a *reverse* or a *negative* message to each node from which it received the *explore* message. Each node at a level less than $f+1$ waits until it has received a *reverse* message for each *forward* or *explore* message that it sent. If it is not the root, it then sends a *reverse* message to its parent. Termination for the algorithm can be achieved by attaching a bit to each *reverse* message, indicating if any nodes were discovered at level $f+1$.

The total number of messages due to all exploration is $O(m)$, since at most two *explore* messages, plus matching *reverse* messages, are sent along each edge. There are $O(n^2)$ messages due to synchronization, since there are $O(n)$ phases, with each of $O(n)$ edges in the current breadth-first tree carrying one message in each of the broadcast and echo. Thus the total number of messages is $O(n^2)$. The time is bounded as follows. A broadcast of *forward* (f) will take time $f-1$, an exploration will take constant time, and the echo time $f-1$, which is $O(n)$ time per phase. Since there are $O(n)$ phases, each following the preceding one, the time is $O(n^2)$.

If the network is sparse, there is a more efficient approach. The idea is to have fewer synchronization phases by extending the breadth-first tree l levels at a time between synchronization phases, where l is a parameter to be specified later. This basic idea has also appeared in [G], but was not taken full advantage of in that paper. As before, the activities in a phase are broadcast, exploration, and echo.

Messages used in exploration will be of the form *explore* (j, k), where k

indicates the number of levels that can be explored from the current node, and j indicates the index of the next level. Nodes at the current frontier, level f, will send out *explore* ($f+1$, l) messages. Note that the first message to reach a node will not necessarily determine the node's parent in the final breadth-first tree, since an explore message could come along later on a shorter path from some node on the frontier. Assume that $level(w) = \infty$ and $parent(w) = nil$ initially for each node w except the root. Suppose an *explore* (j, k) message is received at a node w. If $j \geq level(w)$, then the *explore* (j, k) message did not identify a shorter path to w than that previously known, and a *negative* (j) message is returned to indicate this fact. If a node v receives a *negative* (j) message from node w, and $level(v)$ is still $j-1$, then w should be removed from the list of children of v.

If node w receives an *explore* (j, k) message from v, where $j < level(w)$, then a shorter path to w has been found. If $parent(w) \neq nil$, a *negative* ($level(w)$) message is sent to this parent. In any case, $level(w)$ is reset to j, $parent(w)$ is reset to v, and the list of children of w is reset to be the adjacency list of w, with v removed. If $k=1$, then a *reverse* (j) message is sent to v. If $k>1$, then an *explore* ($j+1$, $k-1$) message is sent to each node on this list.

Let j be the current value of $level(w)$. Node w will ignore any *reverse* (j') or *negative* (j') message with $j' > j+1$. If w has received *negative* ($j+1$) or *reverse* ($j+1$) messages from each node to which it sent an *explore* ($j+1$, $k-1$) message, it sends a *reverse* (j) message to $parent(w)$.

The echo is handled as before.

Theorem 1. A breadth-first tree can be found in a distributed network of n nodes and

m edges using $O(n\, m^{1/2})$ messages and time.

Proof. The above algorithm will correctly find a breadth-first search tree. Suppose that at the beginning of phase i that the first il levels of the tree have been correctly constructed. Nodes whose correct level number should be $il+1$ will eventually receive an *explore*$(il+1, l)$ message from some node on the frontier. If a node w receives an *explore*(j, k) message, where j is its correct level number and $k > 1$, then an *explore*$(j+1, k-1)$ message will be sent to every neighbor of w except its parent, and thus the level number of any neighbor will be at most $j+1$. Then it follows by induction on the level number from the frontier that all nodes at levels $il+1$ through $(i+1)l$ will be correctly added to the tree. Whenever an incorrectly labeled node receives its correct level number, the node is removed from the list of children of its previous parent. Thus it follows that the list of children at each node will be correct. By induction on k, each correctly labeled node w will receive a *negative* or *reverse* message from each node that it had included initially on its list of children at the time that w was correctly labeled. Thus one can conclude that each phase will terminate.

The number of messages that are used is bounded as follows. Since at most two *explore*(j, k) messages are sent along each edge, for $k=1, 2, \cdots, l$, the total number of messages due to exploration is $O(lm)$. Since there are $O(n/l)$ synchronization phases, there are $O(n^2/l)$ messages due to synchronization. With l chosen to be $n/m^{1/2}$, we achieve the desired result for messages. A longest sequence of messages during one phase will contain $f-1$ *forward* messages, l *explore* messages, and a corresponding number of *reverse* messages. Thus $O(n)$ time will be used per phase, over $O(n/l)$ phases, or $O(nm^{1/2})$ in total. □

3. Distributed algorithm for finding a planar separator

We describe an algorithm for finding a separator in a planar distributed network, given a breadth-first search tree of the network. Our algorithm is an adaptation of the method of Lipton and Tarjan [LT1] for finding a separator in a planar graph. To make their algorithm communication efficient, at crucial points in the algorithm we use several variants of binary search that are suitable for distributed computation. Following [LT1], the vertices have nonnegative vertex costs summing to no more than 1. The algorithm must partition the vertices of the graph into three sets A, B, and C such that no edge joins a vertex in A with a vertex in B, neither A nor B has total cost exceeding 2/3, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices. We assume that each vertex has a list of the edges incident on it in clockwise order around the node.

For convenience we call the algorithm in [LT1] algorithm PS. If some vertex v has cost at least 1/3, then take $C = \{v\}$ and $B = \emptyset$. If the total cost of all vertices is less than 1/2, then take $B = C = \emptyset$. Otherwise, PS does the following on a connected graph. Given a breadth-first search tree, algorithm PS first finds the largest level l_1 such that the total cost of all vertices on levels 0 through $l_1 - 1$ is at most 1/2. In a distributed setting we can accomplish this by performing a binary search for 1/2, probing at level numbers. Each test of a level number l involves broadcasting a message out along the breadth-first search tree up through level l, and accumulating the cost of nodes at level l or lower on the return sweep. Since each broadcast uses $O(n)$ messages and $O(n)$ time, determining l_1 uses $O(n \log n)$ messages and time. Let k be the number of nodes in levels 0 through l_1. The value of k can be computed by a broadcast and echo in the breadth-first tree.

Let $L(l)$ be the number of vertices on level l. Algorithm PS determines a level

l_0 where $l_1 - \lfloor\sqrt{k}\rfloor \le l_0 \le l_1$ and $L(l_0) + 2(l_1 - l_0) \le 2\sqrt{k}$. In a distributed setting, we can do the following. If $l_1 - \lfloor\sqrt{k}\rfloor < 0$, choose $l_0 = 0$. Otherwise, perform a search of the (closed) interval $[l_1 - \lfloor\sqrt{k}\rfloor, l_1]$ similar to binary search. Let $[a, b]$ be the current interval. If $a = b$, then choose $l_0 = a$. If $a < b$, then consider level $l = \lceil (a+b)/2 \rceil$. Level l can be tested by sending out one broadcast in the breadth-first tree, and accumulating on the return the number of nodes $L([a, l-1])$ and $L([l, b])$ in the intervals $[a, l-1]$ and $[l, b]$, resp. Compute the following two averages, and determine which of the two is no larger than the other (they could be equal): $(L([a, l-1]) + \sum_{i=a}^{l-1} 2(l_1 - i)) / (l-a)$ and $(L([l, b]) + \sum_{i=l}^{b} 2(l_1 - i)) / (b-l+1)$. Continue searching recursively within the corresponding interval.

It is easy to see that the above procedure finds a level l_0 such that $L(l_0) + 2(l_1 - l_0)$ is at most $2\sqrt{k}$. If $l_1 - \lfloor\sqrt{k}\rfloor < 0$, then $l_1 \le \lfloor\sqrt{k}\rfloor - 1$, and thus $L(l_0) + 2(l_1 - l_0) \le 2\lfloor\sqrt{k}\rfloor - 1$. Otherwise the average of $L(l) + 2(l_1 - l)$ over levels $l_1 - \sqrt{k} \le l \le l_1$ is initially less than $2\sqrt{k}$, and the average of the quantity $L(l) + 2(l_1 - l)$ for those levels excluded on any one step is no smaller than the average of those levels retained. $O(\log n)$ levels are tested, for a total of $O(n \log n)$ messages and time. A similar approach finds a level l_2 where $l_1 + 1 \le l_2 \le l_1 + \lceil\sqrt{n-k}\rceil$ and $L(l_2) + 2(l_2 - l_1 - 1) \le 2\sqrt{n-k}$.

Algorithm *PS* next deletes vertices at levels l_2 and larger, and contracts vertices at levels 0 through l_0 to a single vertex. Since the network topology cannot be changed, we instead reassign the cost of each of these nodes to be 0. Then for each node v, we record the parent of v in the tree, and the total cost of all descendants of v, including v itself. This can be accomplished within the framework of a broadcast-echo in the breadth-first tree.

Algorithm *PS* then triangulates the faces of the embedding of the graph. Again we cannot modify the network explicitly, but instead will traverse the network in a fashion that is consistent with a particular triangulation. (The triangulation, or more properly, a subset of the edges of a triangulation, will be induced as the traversal proceeds.) Given the triangulation, algorithm *PS* chooses a nontree edge which induces a cycle with respect to tree edges. We similarly choose some nontree edge in the network. (We are assuming that the network contains at least one cycle. Otherwise there is a simpler, and more message-efficient, method to find a separator.) Algorithm *PS* then determines which side of the cycle contains vertices of greater cost, and denotes this side as the *inside* of the cycle. Again, we can perform this task using a broadcast-echo in the breadth-first tree. (If the root is inside the cycle, reroot the tree at some cycle vertex.) Each node in the cycle can be labeled as being on the cycle by this broadcast.

If the cost inside the cycle exceeds 2/3, algorithm *PS* shrinks the cycle iteratively as follows. Let (v_i, w_i) be the nontree edge that induces the current cycle. Algorithm *PS* identifies the triangle inside the cycle that has edge (v_i, w_i). Call the third vertex of the triangle y. If either (v_i, y) or (y, w_i) is a tree edge, then (v_{i+1}, w_{i+1}) is set to the nontree edge among the two. If v_{i+1} is a child of v_i in the tree, then the cost of v_{i+1} is subtracted from the cost inside the cycle, and similarly with w_{i+1} and w_i.

If neither (v_i, y) nor (y, w_i) is a tree edge, then algorithm *PS* determines the tree path from y to the (v_i, w_i) cycle by following the parent pointers from y. Let z be the vertex on the (v_i, w_i) cycle reached by this search. The cost of this path, excluding vertex z, is computed. Then algorithm *PS* computes the cost inside the (v_i, y) and (y, w_i) cycles as follows. Each tree edge incident on, and inside of a cycle, is incident on a

vertex that contains the total cost of a subtree inside the cycle. The algorithm interleaves the operations involved in scanning edges inside the (v_i, y) cycle, with those in scanning edges inside the (y, w_i) cycle, until it has scanned all tree edges incident on, and inside of, one of these cycles. Once the cost inside one cycle is known, the cost inside the other cycle can be determined by subtracting the cost inside one cycle and the cost of the path from the cost of the (v_i, w_i) cycle. The edge inducing the cycle whose inside has larger cost then becomes (v_{i+1}, w_{i+1}). This approach guarantees the linear time performance claimed for algorithm *PS*.

We handle the shrinking of the cycle as follows. Let $p(v_i)$ be the node preceding v_i on the cycle. Let the direction around vertex v_i from (v_i, w_i) to $(v_i, p(v_i))$, on the inside of the cycle be called *insidewise*. Assume that a search process is at node v_i. The process will carry as data the names of the nodes v_i and w_i, the position of edge (v_i, w_i) in the adjacency list of v_i (or the position (v_i, w_i) would occupy if there were such an edge), and the current cost inside of the cycle. The process should choose the next edge (v_i, y) in an insidewise direction around v_i from (v_i, w_i). If (v_i, y) is a tree edge, then we choose (y, w_i) as (v_{i+1}, w_{i+1}). In this case we move the search process to v_{i+1}. Note that the edge (y, w_i) may not exist in the network, but can be viewed as part of the partial triangulation generated so far. If (v_i, y) is not a tree edge, then send a message from v_i to y to determine if there is a tree edge from y to w_i. If so, choose (v_i, y) as (v_{i+1}, w_{i+1}).

In the case that neither (v_i, y) nor (y, w_i) is a tree edge, we find the path from y to z as above, by sending a process up the tree from y until it encounters a node z on the cycle. Nodes on the path from y to z will be labeled as cycle nodes. However, to find the cost inside the (v_i, y) and (y, w_i) cycles, we cannot perform efficiently the par-

ticular type of interleaving discussed above because of the cost of synchronization. We economize on communication by performing half of a one-sided binary search, as follows. We use a bound on the number of operations performed in examining each cycle, which is initially set to some small constant. Starting at z, we check tree edges inside one cycle, until the bound on operations is exhausted, and then return to z and do the same in the other cycle. Checking tree edges corresponds to summing the weights of the children inside the cycle. If neither cycle is completed, double the bound and repeat. This approach can be seen to require messages proportional to the smaller of the number of messages used to handle either of the two cycles alone. By an argument similar to that giving the linear time for algorithm *PS*, this portion of our algorithm can be seen to use a linear number of messages altogether. When (v_{i+1}, w_{i+1}) has been determined, shift the search process to y if $v_{i+1} = y$.

Upon completion, the separating set will consist of the nodes on the cycle between levels l_0 and l_2, plus all nodes on levels l_0 and l_2.

Theorem 2. Let G be a planar distributed network of n nodes. A separator for G of size at most $2\sqrt{2}\sqrt{n}$ can be found using $O(n \log n + B_1(n))$ messages and $O(n \log n + B_2(n))$ time, where $B_1(n)$ and $B_2(n)$ are the number of messages and the time necessary to find a breadth-first search tree in a planar graph.

Proof. Correctness of our algorithm is based in large part on the correctness of the Lipton and Tarjan procedure, which we have been calling PS. We concentrate our discussion on those parts of our algorithm that are not just a straightforward translation of *PS*. As argued previously, levels l_0 and l_2 satisfied the required bounds on level number and number of nodes between levels. As pointed out above, nodes that would

have been pruned or contracted together in *PS* are assigned weight 0 in our algorithm. As discussed, a triangulation sufficient for the search process can be inferred as the search process progresses. Thus the movement of the search process in our algorithm will mimic the movement of the search process in *PS*. Correctness then follows.

We next discuss the performance bounds. There will be at most 2 nodes on every level in the cycle. Thus the number of nodes on levels l_0 through l_1 will be at most $2\sqrt{k}$, and the number of all other nodes in the separator will be at most $2\sqrt{n-k}$. Thus the total number of nodes in the separator is at most $2(\sqrt{k}+\sqrt{n-k}) \leq 2\sqrt{2}\sqrt{n}$. The bound on the time and message complexity follows from the previous discussion. □

4. Regions and boundary nodes

Our shortest path algorithm in the planar network makes use of a *division* of the planar network into regions [Fs1]. A *region* consists of two types of nodes, boundary nodes and interior nodes. An *interior node* is contained in exactly one region and is adjacent to nodes only in its own region. A *boundary node* is shared among at least two regions and is adjacent to interior nodes of each of these regions. To generate appropriate regions, we make use of our distributed version of the planar separator algorithm.

To be able to use the regions efficiently in our shortest paths application, it is convenient to have the degree of every node bounded by some small constant. While many networks may satisfy this constraint, it is possible that there are nodes of rather large degree in some networks. We solve this problem by having any node of degree greater than 3 split logically (not physically) into a subgraph of nodes and edges of degree 3. A well-known transformation in graph theory [H, p. 132] may be used to do

this. Consider a planar embedding of the network. For each node v of degree $d > 3$, where $w_0, \cdots, w_{d-1}$ is a cyclic ordering of the nodes adjacent to v in the planar embedding, replace v with new nodes $v_0, \cdots, v_{d-1}$. Add edges $\{(v_i, v_{(i+1) \bmod d}) \mid i=0, \cdots, d-1\}$, each of distance 0, and replace the edges $\{(w_i, v) \mid i=0, \cdots, d-1\}$ with $\{(w_i, v_i) \mid i=0, \cdots, d-1\}$, of corresponding distances. From a corollary of Euler's formula [H], the number of nodes in the resulting network will be less than six times the number of vertices in the original network. Note that in any distributed algorithm the processor at node v will perform an emulation of the algorithm on the logical nodes $v_0, \cdots, v_{d-1}$.

Given a parameter r, we show how to generate connected regions with $O(r)$ nodes each, and $O(n/\sqrt{r})$ boundary nodes in total. Each region will have as an index an integer between 1 and n. Initially, label all nodes as being in region n. Then apply the following recursive procedure with arguments n as the number of nodes, n as the number of interior nodes, and n as the label.

We now describe the recursive procedure, with parameters size n' of the region, and label L of the region. If $n' \leq r$, then return. Otherwise, do the following. Apply the separator algorithm to the network with all node weights equal to $1/n'$, yielding sets A, B and C. Let C' be the set of vertices in C not adjacent to any vertex in $A \bigcup B$, and let $C'' = C - C'$.

Identify the connected components $A_1, A_2, \cdots, A_q$ in $A \bigcup B \bigcup C'$. This can be done as follows. Initially give each vertex in $A \bigcup B \bigcup C'$ a *null* label, and set i to 0. Perform an inorder traversal of the breadth-first search tree. Whenever a node is encountered with a *null* label, increment i, reset its label to i, and make it the leader of component A_i. Perform a broadcast within the set of all nodes with *null* label that can

be reached along a path of nodes with *null* label. Label each such node with i. On the echo of this broadcast, compute the number of nodes in this component, and store this in the leader. When this broadcast is complete, continue the inorder traversal. Once the traversal is complete, interior nodes in all components have been identified, except that some nodes in C'' may not be adjacent to interior nodes in two different regions. Any node v in C'' adjacent to a node in A_i and not adjacent to a node in A_j for $j \neq i$, can be removed from C'' and included in A_i. This test can be performed on the nodes of C'' one at a time in any order, so that we perform an inorder traversal of the breadth-first tree, and handle such nodes in inorder. Region i will have as interior nodes the nodes in A_i that were interior in the input network. The boundary nodes will be the remaining nodes in A_i and those nodes in C'' that are adjacent to some node in A_i. Note that by our construction of regions, the subnetwork induced on the interior nodes of any region is connected. Since each resulting region will be a proper subset of $A \cup C'$ or $B \cup C'$, each region will contain fewer than $2n'/3 + 2\sqrt{2}\sqrt{n'}$ nodes.

Two additional tasks need to be done. First, each node in a component should query its neighbors to find out which neighbors are in the same component. Once each node has this information, the component can be handled logically as a whole network in any recursive calls of the procedure on the component.

Second, the labels of the new regions must be formed. Initialize *count* to L minus the number of interior nodes in the input network, and start an inorder traversal. When the leader of a component A_i is encountered, perform a broadcast within the component to obtain an updated count of the number n_i of nodes in the component, and the number n'_i of interior nodes in the component. Reset *count* to be *count* + n'_i, and broadcast this value as the label of the component. The component is thus set up itself

as a region. Once a component has been handled, resume the inorder traversal. The time and messages for generating the labels can be seen to be $O(n')$.

Once the new regions have been identified and labeled, the procedure is applied concurrently to each new region. We note that our approach shares some similarities with an approach for the nondistributed case that is described in [LT2].

Lemma 1. An n-node planar distributed network can be divided into connected regions with no more than r nodes each, and $O(n/\sqrt{r})$ boundary nodes in total, using $O(n(\log n)^2 + B_1(n)\log n)$ messages, and $O(n \log n + B_2(n))$ time, where $B_1(n)$ and $B_2(n)$ are the number of messages and the time to find a breadth-first search tree in a planar network.

Proof. The number of boundary nodes follows from the results in [Fs1]. By Theorem 2, the number of messages and the time to find a planar separator will be $O(n \log n + B_1(n))$ and $O(n \log n + B_2(n))$, respectively. Given the planar separator, the messages and time to find and augment and label the regions will be $O(n)$. The recurrence for the number of messages will thus be, for sufficiently large n, $M(n) \leq n \log n + B_1(n) + \sum_i (M(n_i))$, where $\max_i \{n_i\} \leq 2n/3 + 2\sqrt{2}\sqrt{n}$, $\sum_i n_i \leq n + 4\sqrt{2}\sqrt{n}$, and n_i is the number of nodes in A_i. The additive term of $4\sqrt{2}\sqrt{n}$ results from the fact that every node is of degree at most 3, and thus each boundary node can be counted as a member of at most 3 regions. The claimed bound on messages is the solution to the recurrence. Since the procedure is applied concurrently to the new regions, the recurrence for the time will be, for sufficiently large n, $T(n) \leq n \log n + B_2(n) + T(\max_i \{n_i\})$, where $\max_i \{n_i\} \leq 2n/3 + 2\sqrt{2}\sqrt{n}$. The claimed bound on time is the solution to the recurrence. □

5. Finding a single source shortest path tree

Our algorithms are based on Dijkstra's single source shortest paths algorithm [D]. Dijkstra's algorithm performs a search of a graph that proceeds in phases. Each vertex v whose shortest distance $d(s, v)$ from the source s is not known, is said to be *open*, and the currently known shortest distance $\rho(v)$ from the source to v is maintained. Initially, s is closed and all other vertices are open, $\rho(s) = 0$, $\rho(v) = c(s, v)$ for every neighbor v of s, and $\rho(v) = \infty$ for every other vertex v. In each phase, the open vertex v with minimum $\rho(v)$ is closed, and the shortest distances $\rho(w)$ are updated for all w such that there is an edge (v, w). For any vertex $v \neq s$ with $\rho(v) < \infty$, the name of the vertex $parent(v)$ is maintained, where $\rho(v) = \rho(parent(v)) + c(parent(v), v)$. Note that $parent(v)$ is closed. When all vertices have been closed, the ρ values represent shortest distances from s, and the parent pointers encode a shortest path tree rooted at s. A natural implementation [J] of this algorithm maintains the distances $\rho(v)$ in a heap.

We first discuss a straightforward implementation of Dijkstra's algorithm on a network. We define the *current shortest path tree* as the set of nodes v with $\rho(v) < \infty$, plus the edges $(parent(v), v)$ for each $v \neq s$ in this set. Note that any nonleaf node in this tree is closed. For each open node v, the value $\rho(v)$ is maintained, along with $parent(v)$. For each closed node v, $\rho(v)$, $parent(v)$, and the children of v are maintained. The heap will be maintained in the current shortest path tree. Let $minval(v)$ be the minimum $\rho(u)$ of any open node u that is a descendant of v in the current shortest path tree. Let $minnode(v)$ hold the corresponding node u. Node v will maintain $minval(v)$, $minnode(v)$. At the conclusion of the computation, each node will know its parent and its children in the shortest path tree.

The initialization is performed as follows. Every node v not adjacent to the

source s sets $\rho(v)$ to ∞. Source s sets $\rho(s)$ to 0, and copies its adjacency list to be its list of children. Source s notifies each child w to set $\rho(w)$ to $c(s, w)$, *parent*(w) to s, and *minval*(w) to $\rho(w)$. Source s sets *minval*(s) to $\rho(w')$ and *minnode*(s) to w', where w' is a child of s with smallest ρ value. Once the initialization is complete, the computation proceeds in phases.

A phase starts when the source s selects *minval*(s), which equals $\rho(v)$ for some v = *minnode*(s). The source then initiates a broadcast in the current shortest path tree by sending a *close*(v) message to each of its children. When a closed node u receives the *close*(v) message, it will set *minval*(u) to ∞ and *minnode*(u) to 0, and send a *close*(v) message to each of its children. When node v receives a *close*(v) message, it marks itself as closed, sets its list of children to its adjacency list minus its parent, and sets *minval*(v) to ∞ and *minnode*(v) to 0. It then computes *dist*$(w) = \rho(v)+c(v, w)$ for each child w. It then sends an *explore*(*dist*(w)) message to each child w. When a node w receives an *explore*(x) message, it compares x with $\rho(w)$. If $x \geq \rho(w)$, then a *update*$(\infty, 0)$ message is returned to v, and an *update*$(\rho(w), w)$ message is sent to *parent*(w) in response to a *close*(v) message that w received from *parent*(w). Otherwise, the value $\rho(w)$ is updated, *parent*(w) is set to v, and an *update*(x, w) message is returned to v. If w had a parent previously, then a *update*$(\infty, 0)$ message is sent to this parent in response to a *close*(v) message.

The echo, along with the adjustment of the heap, is handled as follows. When an open node $w \neq v$ not adjacent to v receives a *close*(v) message, it returns an *update*$(\rho(w), w)$ message. A node u that receives an *update*(x, t) message from w will do the following. If $x <$ *minval*(u), it will then reset *minval*(u) to x and *minnode*(u) to t. Otherwise, if $x = \infty$ and w is open, then w is removed from the list of children.

(Thus the message *update* $(\infty, 0)$ from an open node plays the same role as the *negative* message in our algorithms for finding a breadth-first search tree.) Node u will wait until it has received an *update* (x, t) message from each node w to which it sent a *close* (v) or *explore* message. If $u \neq s$, it will then return an *update* (*minval* (u), *minnode* (u)) message to its parent. When s has received messages from of all of its children in the tree, it will begin the next phase if *minnode* $(s) \neq 0$, and will terminate the algorithm otherwise. Note that termination occurs when all nodes have been closed.

Theorem 3. A shortest path tree can be found in a distributed network of n nodes and nonnegative edge costs using $O(n^2)$ messages and time.

Proof. We argue by induction on the number of phases that the above adaptation of Dijkstra's algorithm correctly computes the current shortest path tree and the heap embedded within it. Clearly the current shortest path tree and the heap are set up correctly prior to the first phase. Assume that the tree and the heap are correct prior to phase i. We shall argue that they are correct after completion of phase i. By definition of *minval* (s) and $v =$ *minnode* (s), the algorithm chooses the correct node to close. The broadcast ensures that each open node receives the *close* (v) message. Node v notifies each neighbor w other than *parent* (v), allowing for w to update $\rho(w)$. It is clear that each neighbor w of v updates $\rho(w)$ correctly. Also, w sends a response to the message it received from v, and to its parent if it was already in the current shortest path tree. Thus w will have the correct parent in the current shortest path tree at the end of phase i.

The values in the heap are adjusted correctly by the following argument. When

open node $w \neq v$ not adjacent to v responds to a *close*(v) message, it sends its $\rho(w)$ value to its parent. When v receives the *close*(v) message during the broadcast, it sets *minval*(v) to ∞. On the echo, node v determines the smallest $\rho(w)$ among the neighbors w that actually become children of v. When a closed node u received the *close*(v) message during the broadcast, it set *minval*(u) to ∞. On the echo, node u determines the smallest ρ value forwarded to it by its surviving children. It forwards this ρ value, along with the corresponding node name, to its parent. Thus the tree and the heap are correct at the end of phase i.

When all nodes in the current shortest path tree are closed, then *minnode*$(s) = 0$, and the algorithm will terminate.

The time and the number of messages used can be seen to be $O(n^2)$ by arguments similar to those for the simple breadth-first search strategy discussed earlier. □

We next consider a more involved implementation of Dijkstra's algorithm, which will use $o(n^2)$ messages for a planar network. The idea, following [Fs1], is to conduct the iterative search on a carefully selected subset of nodes. The subset of nodes will be the boundary nodes of a division. Let a *constrained shortest path* from u to v be a path of shortest length from u to v constrained to contain no boundary nodes as intermediate nodes in the path. Let $d'(u, v)$ be the length of such a path. Initially, the source s is closed, and all other nodes are open. In addition, $\rho(s) = 0$, and $\rho(v) = d'(s, v)$. The search proceeds by constructing a current shortest path tree, and maintaining a heap within it, using *minval* and *minnode* fields at each node. However, only boundary nodes will be chosen to be closed, and thus all leaves in the current shortest path tree will be boundary nodes. (Interior nodes on shortest paths to closed

boundary nodes will also be marked as closed.)

Preprocessing is needed to find a division, and to identify the boundary nodes. Additional preprocessing will then determine constrained shortest paths between all pairs of boundary nodes. During the search, when a boundary node v is closed, $\rho(w)$ must be updated for all boundary nodes w such that a constrained shortest path from v to w exists. At the end of the search, the current shortest path tree includes each boundary node. Postprocessing then determines the location of each remaining node in a shortest path tree.

We now present the distributed version of this algorithm. We do the following preprocessing. Find a division of a planar network, with $r = n^{2/3}$. Within each region, route a description of the region to each node. For each region, once a node within the region possesses a description of the region, the node performs the following computations. Let a *constrained shortest path tree* in a region be a shortest path tree constrained so that no boundary node other than the root can have children in the tree. (This can be enforced by performing the shortest path computation on a directed graph, with no outgoing arcs from any of the boundary nodes other than the designated root.) Such a constrained tree exists, since the subnetwork induced on the set of interior nodes of the region is connected. A boundary node computes a constrained shortest path tree rooted at it. An interior node computes for every boundary node of its region a constrained shortest path tree rooted at that boundary node. A standard single source shortest path algorithm can be used for these computations. Obviously, no messages are used in these latter computations, once each node has a description of the region. The result of this preprocessing is that each node knows the following information. A boundary node for the region will know the length of a constrained shortest path to

each other boundary node of the region, along with the first edge on a constrained shortest path to any other node in the region. An interior node will know its set of children in the constrained shortest path tree rooted at any boundary node of the region.

Given this preprocessing, the search portion of the algorithm proceeds by building a current shortest path tree. The initialization for the search is as follows. Every boundary node v not contained in a region containing the source s sets $\rho(v)$ to ∞. Source s sets $\rho(s)$ to 0. If s is an interior node, then the current shortest path tree is initialized to be the constrained shortest path tree for s. If s is a boundary node, then the current shortest path tree is initialized to be the union of the constrained shortest path trees for s in each of its regions, with any boundary node that is in more than one of these regions informing its parent in all but one of its trees to delete it as a child. In either case, any interior node should be deleted if it does not have a boundary node as a descendant, and the ρ, *minval* and *minnode* values should be set appropriately. Each node also has an *ancestor* field, that gives the name of the lowest proper ancestor that is a boundary node. Once the initialization is complete, the search proceeds in phases.

A search phase starts when the source s selects $minval(s)$, which equals $\rho(v)$ for some $v = minnode(s)$. The source then initiates a broadcast in the current shortest path tree by sending a $close(v)$ message to each of its children. When a node u that is closed or is an interior node receives a $close(v)$ message, it sets $minval(u)$ to ∞ and $minnode(u)$ to 0, and sends $close(v)$ to each of its children. In addition, if u is an interior node that is open and $minnode(u) = v$, then u should mark itself as closed. When node v receives a $close(v)$ message, it marks itself as closed, and sets its list of children as follows. Node v concatenates the lists of children of v in the constrained shortest path trees rooted at v in each region in which v is a boundary node. It deletes

parent(v) from this list. Node v then sets *minval*(v) to ∞ and *minnode*(v) to 0, and computes *dist*$(w) = \rho(v) + c(v, w)$ for each child w. An *explore*(*dist*(w), v) message is sent to each child w.

When an interior node u receives an *explore*(x, v) message from a node t, it does the following. (Note that node u may already be in the current shortest path tree, and thus already have a list of children.) If *parent*(u) is undefined, node u sets *parent*(u) to t, its *ancestor*(u) to v, and takes as its list of children its list of children in the constrained shortest path tree rooted at v within the region. For each node w in the list of children, it computes *dist*$(w) = x + c(u,w)$, and sends w an *explore*(*dist*(w), v) message. If *parent*(u) is defined, node u sets its tentative parent to be t, its tentative ancestor to be v, and takes as its tentative list of children its list of children in the constrained shortest path tree rooted at v within the region. For each node w in the tentative list of children, it computes *dist*$(w) = x + c(u,w)$, and sends w an *explore*(*dist*(w), v) message. (Note that we could determine at this point whether the tentative list of children should supplant the current list of children, by maintaining and comparing a ρ value for u with x. However, to aid in the synchronization of the algorithm, we allow *explore* messages and *close* messages to penetrate to all boundary nodes in the regions containing node v.)

When a boundary node u receives an *explore*(x, v) message from a node t, it does the following. It compares x with $\rho(u)$. If $x \geq \rho(u)$, then an *update*$(\infty, 0, v)$ message is returned to t. Otherwise, the value $\rho(u)$ is updated, and if u already has a parent, then an *update*$(\infty, 0,$ *ancestor*$(u))$ message is sent to this parent in response to a *close*(v) message. In the case in which x was less than $\rho(u)$, node u also then sets *parent*(u) to t, and *ancestor*(u) to v. In either case, node u may share more than one

region with node v, and thus u must wait until *explore* messages have been received through each of these regions. Once the necessary number of messages have been received, an *update* $(\rho(u), u, ancestor(u))$ message is sent to $parent(u)$ in response to either a *close* (v) message (if the $parent(u)$ has remained unchanged) or in response to an *explore* message (if the $parent(u)$ has changed).

The echo, along with the adjustment of the heap, is similar to the echo for the previous shortest path algorithm. When an open boundary node $w \neq v$ not in a region containing v receives a *close* (v) message, it returns an *update* $(\rho(w), w, ancestor(w))$ message. Consider a node $u \neq v$ that sent out *close* (v) messages but no *explore* messages. When it receives an *update* (x, t, v') message from node w it does the following. If $x < minval(u)$, it will reset $minval(u)$ to x and $minnode(u)$ to t. Otherwise, if x is ∞ and w is open, then w is deleted from the list of children. Once node u has received *update* messages in response to all *close* messages that it sent, if $u \neq s$, it will then return an *update* $(minval(u), minnode(u), ancestor(u))$ message to $parent(u)$.

Consider a node u that sent out *explore* messages. When u receives an *update* (x, t, v') message from a node w, it does the following. If $x < minval(u)$, then it resets $minval(u)$ to be x and $minnode(u)$ to be t, and if $u \neq v$ and $v' \neq ancestor(u)$ it reassigns its list of children to be its tentative list of children, sends an *update* $(\infty, 0, ancestor(u))$ to $parent(u)$, and then resets $parent(u)$ to be the tentative parent and $ancestor(u)$ to be the tentative ancestor. Otherwise, if x is ∞ and w is open, then if $u = v$ or $v' = ancestor(u)$ then w is deleted from the list of children, and if $u \neq v$ and $v' \neq ancestor(u)$ then w is deleted from the tentative list of children. Node u will wait until it has received *update* messages in response to all *explore* and *close* messages that it sent. If $u \neq s$, it will then return an

update (*minval* (*u*), *minnode* (*u*), *ancestor* (*u*)) message to *parent* (*u*). As before, when *s* has received messages from of all of its children in the tree, it will begin the next phase if *minnode* (*s*) ≠ 0, and will terminate the search otherwise. Note that termination of the search occurs when all boundary nodes have been closed.

At termination of the search, the current shortest path tree will contain all boundary nodes as closed nodes. Other nodes may be incorporated into the shortest path tree by performing postprocessing in each region concurrently. A modified version of our distributed version of Dijkstra's algorithm can be used in each region, described as follows. Shortest distances are known to the boundary nodes, but not in general to the interior nodes. An appropriate tree is needed to contain the heap, and to make efficient broadcast possible. We initialize this tree to be a spanning tree in which the root is an arbitrary interior node of the region and each boundary node is a leaf. Each boundary node u will have $\rho(u) = d(s, u)$, and the interior nodes in the initial tree will have no ρ value, since they are purely for communication. As Dijkstra's algorithm progresses, each interior node u will be added a second time to the tree, and this time it will be assigned a ρ value. Termination of the shortest path algorithm will occur when the source has been notified by each region that postprocessing within the region is completed.

Theorem 4. A shortest path tree can be found in a planar distributed network of n nodes and nonnegative edge costs using $O(n^{5/3})$ messages and time.

Proof. Correctness of the algorithm follows from the correctness of the sequential algorithm in [Fs1], and from establishing correctness of the distributed versions of the preprocessing, search, and postprocessing. The preprocessing correctly sets up a divi-

sion, and computes constrained shortest path trees within each region of the division. The correctness of the search is established in a fashion similar to that in the proof of Theorem 3. We note the following additional points. First, the only leaves in the current shortest path tree are boundary leaves. If any interior node has no boundary nodes as descendants as a result of the search, then it can be shown by induction that this node will have received *update* $(\infty, 0, v')$ messages from all of its children, and thus will send an *update* $(\infty, 0, v')$ message to its parent, which will cause it to be deleted from its parent's list of children. We also consider the case in which an interior node u receives both a *close* and an *explore* message. It follows from the manner in which *explore* messages are propagated that it can receive at most one *explore* message. From the algorithm it is clear that *close* and *explore* messages are sent to the children of u on the corresponding lists. We first argue that u cannot receive an *update* message back from the children to which it sent *explore* messages until after it has sent *close* messages to the appropriate children. This follows, since the echo proceeds only from boundary nodes, and these boundary nodes wait until they have received one *close* message and in addition one *explore* message for each region that they share with v. The same argument establishes that u cannot receive an *update* message back from the children to which it sent *close* messages until after it has sent *explore* messages to the appropriate children. We also argue that the reassignment of children to u on the echo is correct in the situation that u had received both a *close* and an *explore* message. Node u cannot receive an *update* message with noninfinite *minval* back from both a child to which it sent an *explore* message and a child to which it sent a *close* message. This follows since to claim any current descendant of u (or a potential descendant of u claimed by another node), the *explore* message to u must have identified a shorter path to u than any previously known. Thus node u is appropriately handled. It is also not

hard to establish the correctness of the postprocessing.

We next discuss the time and message complexity of the algorithm. By Theorem 1 and Lemma 1, finding a division of the planar graph will use $O(n^{3/2}\log n)$ messages and $O(n^{3/2})$ time. In the rest of the preprocessing, broadcasting a description of a region of size r_i will use $O(r_i^2)$ messages and time. This follows since there are $O(r_i)$ nodes and edges in the region, each such item must be broadcast throughout the region, at the cost of $O(r_i)$ messages per item. Since $r_i \leq r$, and the total size of all regions is $O(n)$, the total number of messages will be $O(nr)=O(n^{5/3})$ for broadcasting descriptions of regions. For each phase of the search, there will be $O(n)$ messages and time. Since there are $O(n/\sqrt{r})$ phases, the time and the number of messages used in the search will be $O(n^{5/3})$. In the postprocessing, the number of messages in a region of size r_i will be $O(r_i^2)$. The total number of messages for the postprocessing is thus $O(n^{5/3})$. □

Acknowledgment. The author would like to thank the referees for their helpful comments.

References

[AR] J. M. Abram and I. B. Rhodes, A decentralized shortest path algorithm, *Proc. 16th Allerton Conf. on Comm., Control and Computing* (1978) 271-277.

[AG1] B. Awerbuch and R. G. Gallager, A new distributed algorithm to find breadth first search trees, *IEEE Trans. on Inf. Theory 33* (1987) 315-322.

[AG2] B. Awerbuch and R. G. Gallager, Distributed BFS Algorithms, *Proc. 26th IEEE Symp. on Foundations of Computer Science* (October 1985) 250-256.

[D] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik 1* (1959) 269-271.

[Fs1] G. N. Frederickson, Fast algorithms for shortest paths in planar graphs, with applications, *SIAM J. on Computing 16* (1987) 1004-1022.

[Fs2] G. N. Frederickson, A single source shortest path algorithm for a planar distributed network, *Proc. 2nd Symp. on Theoretical Aspects of Computer Science,* Saarbruecken, Germany (January 1985) 143-150.

[Fm] D. U. Friedman, Communication complexity of distributed shortest path algorithms, LIDS-TH-886, Mass. Inst. Tech. (1979).

[G] R. G. Gallager, Distributed minimum hop algorithms, MIT technical report LIDS-P-1175 (1982).

[GHS] R. G. Gallager, P. Humblet, and P. Spira, A distributed algorithm for minimum weight spanning trees, *ACM Trans. Prog. Lang. Sys. 5,* 1 (January 1983) 66-77.

[H] F. Harary, *Graph Theory,* Addison-Wesley, Reading, Mass. (1969).

[J] D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM 24,* 1 (January 1977) 1-13.

[LT1] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math. 36,* 2 (April 1979) 177-189.

[LT2] R. J. Lipton and R. E. Tarjan, Applications of a planar separator theorem, *SIAM J. Comput. 9,* 3 (August 1980) 615-627.

[T] S. Toueg, An all-pairs shortest-path distributed algorithm, RC-8397, IBM T. J. Watson Res. Center (1980).