A Distributed Algorithm for Minimum-Weight Spanning Trees

R. G. GALLAGER, P. A. HUMBLET, and P. M. SPIRA
Massachusetts Institute of Technology

A distributed algorithm is presented that constructs the minimum-weight spanning tree in a connected undirected graph with distinct edge weights. A processor exists at each node of the graph, knowing initially only the weights of the adjacent edges. The processors obey the same algorithm and exchange messages with neighbors until the tree is constructed. The total number of messages required for a graph of N nodes and E edges is at most $5N \log_2 N + 2E$, and a message contains at most one edge weight plus $\log_2 8N$ bits. The algorithm can be initiated spontaneously at any node or at any subset of nodes.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—Distributed networks; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*trees*

General Terms: Algorithms

Additional Key Words and Phrases: Distributed algorithms, communication complexity, shortest spanning trees

1. INTRODUCTION

In this paper we consider a connected undirected graph with N nodes and E edges, with a distinct finite weight assigned to each edge. We describe an asynchronous distributed algorithm which determines the minimum-weight spanning tree (MST) of the graph. We assume that each node initially knows the weight of each edge adjacent to that node.

Each node performs the same local algorithm, which consists of sending messages over adjoining links, waiting for incoming messages, and processing. Messages can be transmitted independently in both directions on an edge and arrive after an unpredictable but finite delay, without error and in sequence.

After each node completes its local algorithm, it knows which adjoining edges are in the tree and also knows which edge leads to a particular edge designated as the core of the tree.

We view the nodes in the graph as being initially asleep. One or more of the nodes then wake up, spontaneously or upon receiving messages from awakened neighbors, and then proceed with their local algorithms.

Under these assumptions, we shall see that the total number of messages exchanged by the nodes to find the MST is less than $2E + 5N \log_2 N$. Each message consists of at most one edge weight, one integer between zero and $\log_2 N$, and three additional bits.

Our algorithm is similar to an earlier algorithm described by Spira [8], which followed an earlier algorithm given by Dalal [1]. Spira did not analyze in detail the number of messages required by his algorithm, but he gave a heuristic argument that the expected number of messages (over randomly selected graphs) would grow with E and N as $E + N \log N$.

If the nodes of the network have distinct identities that can be ordered, then it is easy to extend the algorithm to the case where the edge weights are not distinct. One simply appends to the edge weight the identities of the two nodes joined by the edge, listing, say, the lower ordered node first. These appended weights have the same ordering as before, with ties broken by the node identities. If the nodes do not know the identities of their neighbors, then each node can send its identity over each adjoining edge, thus requiring a total of $2E$ extra messages. We have also developed another algorithm, briefly described later, that does not require distinct weights and thus does not require these additional $2E$ messages.

If the network has neither distinct edge weights nor distinct node identities, then no distributed algorithm (of the type described above) exists for finding an MST with a bounded number of messages. This can be seen most easily for a three-node fully connected graph with equal-weight edges. Any two edges form an MST, but since the nodes obey the same algorithms, they have no way to choose. If nodes chose random identities, then the algorithm could be made to work as soon as the identities were all different, but there is no way to guarantee this in a finite number of choices. Naturally, the *expected* number of required choices is small, but any bounded number of messages will fail with some positive probability.

Distributed MST algorithms are useful in communication networks when one wishes to broadcast information from one node to all other nodes and there is a cost associated with each channel of the network. If the cost of using a channel in one direction is different from that in the opposite direction, then the MST does not provide the desired solution, but a companion paper [3] treats this more general problem. In addition to the broadcast application, there are many potential control problems for networks whose communication complexities are reduced by having a known spanning tree. With topology changes caused by possible failures in the network, it is desirable to be able to generate a spanning tree starting from any node or subset of nodes, and the algorithm here is as efficient as any we have been able to find for generating an arbitrary spanning tree. Finally, there are a number of applications for distributed algorithms that can

select the node in the network with the highest identity number. An efficient distributed algorithm for this problem starts with the MST algorithm and then uses the resulting tree to find the highest numbered node.

2. REVIEW OF SPANNING TREES

We assume the reader is familiar with the elementary definitions and properties of graphs, paths, cycles, trees, etc., which can be found, for example, in [5, 6]. Suppose that each edge e of a graph has a weight $w(e)$ associated with it. The weight of a tree in the graph is defined as the sum of the weights of the edges in the tree, and our objective is to find a spanning tree of minimum weight, that is, an MST. A fragment of an MST is defined as a subtree of the MST, that is, a connected set of nodes and edges of the MST. The algorithm starts with each individual node as a fragment and ends with the MST as a fragment. Define an edge as an *outgoing edge* of a fragment if one adjacent node is in the fragment and the other is not.

PROPERTY 1. *Given a fragment of an MST, let e be a minimum-weight outgoing edge of the fragment. Then joining e and its adjacent nonfragment node to the fragment yields another fragment of an MST.*

PROOF. Suppose the added edge e is not in the MST containing the original fragment. Then there is a cycle formed by e and some subset of the MST edges. At least one edge $x \neq e$ of this cycle is also an outgoing edge of the fragment, so that $w(x) \geq w(e)$. Thus, deleting x from the MST and adding e forms a new spanning tree which must be minimal if the original tree was minimal. The original fragment with e added is a fragment of the new MST.   □

PROPERTY 2. *If all the edges of a connected graph have different weights, then the MST is unique.*

PROOF. Suppose, to the contrary, that there are two different MSTs. Let e be the minimum-weight edge that is in one but not both of the trees, and let T be the set of edges of the MST containing e and T' be the edge set of the other MST. The edge set $\{e\} \cup T'$ must contain a cycle, and at least one edge of this cycle, say e', is not in T (since T contains no cycles). Since the edge weights are all different and e' is in one but not both of the trees, $w(e) < w(e')$. Thus $\{e\} \cup T' - \{e'\}$ is the edge set of a spanning tree of smaller weight than T', yielding a contradiction.   □

These properties immediately suggest a general type of algorithm for finding the MST for a graph with different edge weights. One starts with one or more fragments consisting of single nodes. Using Property 1, one can enlarge these fragments in any order. Whenever two fragments have a common node, Property 2 assures us that the union of these fragments is also a fragment, allowing fragments to be combined into larger fragments. The standard algorithms for generating MSTs correspond to different orders in which the above fragments are enlarged and combined. For example, the Prim–Dijkstra algorithm [2, 7] starts with a single node and successively enlarges the fragment until it spans the graph. The Kruskal algorithm [4] starts with all nodes as fragments and successively extends the fragment with the smallest-weight outgoing edge, combining

fragments where possible. Other algorithms [1, 8, 9] start with all nodes as fragments, extend each fragment, then combine, then extend each of the new enlarged fragments, then combine again, and so forth.

The Prim–Dijkstra and Kruskal algorithms work equally well if some of the edge weights are the same. To see this, simply impose an arbitrary ordering on the equal-weight edges consistent with the choices made in the execution of the algorithms. Algorithms such as [1], [8], and [9] that extend several fragments without intermediate combining do not necessarily work correctly with equal edge weights. For example, in a three-node fully connected network with equal-weight edges, each node could extend with a different edge, giving rise to a cycle when the fragments are combined.

In the algorithm to follow, each fragment finds its minimum-weight outgoing edge asynchronously with regard to other fragments, and, when this edge is found, the fragment attempts to combine with the fragment at the other end of the edge. How and when this combination takes place depends on the *levels* of the two fragments, which depend in turn on previous fragment combinations. Specifically, a fragment containing only a single node is defined to be at level 0. Suppose a given fragment F is at level $L \geq 0$ and the fragment F' at the other end of F's minimum-weight outgoing edge is at level L'. If $L < L'$, then fragment F is immediately absorbed as part of fragment F', and the expanded fragment is at level L'. If $L = L'$ and fragments F and F' have the same minimum-weight outgoing edge, then the fragments combine immediately into a new fragment at level $L + 1$; the combining edge is then called the core of the new fragment. In all other cases, fragment F simply waits until fragment F' reaches a high enough level for combination under the above rules.

Figure 1 illustrates these rules. Fragment F is a level 1 fragment formed when nodes 1 and 2 combine on their common minimum-weight edge, and node 3 and its minimum-weight edge are then absorbed. Fragments F and F' later combine on their minimum-weight edge to form a level 2 fragment, and node 4 is later absorbed. Depending on the timing, it would also be possible for node 4 to be absorbed into fragment F before the formation of the level 2 fragment.

We show later, after describing more of the algorithm, that the waiting in the above rules cannot cause a deadlock. The reason for the waiting is that the communication required for a fragment to find its minimum-weight edge is proportional to the fragment size, and thus communication is reduced by small fragments joining into large ones rather than vice versa.

3. DESCRIPTION OF THE DISTRIBUTED ALGORITHM

We start by describing how a fragment finds its minimum-weight outgoing edge. First consider the trivial special case of a zero-level fragment (i.e., a single node). Initially, each node is in a quiescent state called *Sleeping*. There are three possible node states: the initial state *Sleeping*, the state *Find* while participating in a fragment's search for the minimum-weight outgoing edge, and the state *Found* at other times. When a sleeping node either spontaneously awakens to initiate the overall algorithm or is awakened by the receipt of any algorithm message from another node, the node first chooses its minimum-weight adjacent edge, marks this edge as a *branch* of the MST, sends a message called *Connect*
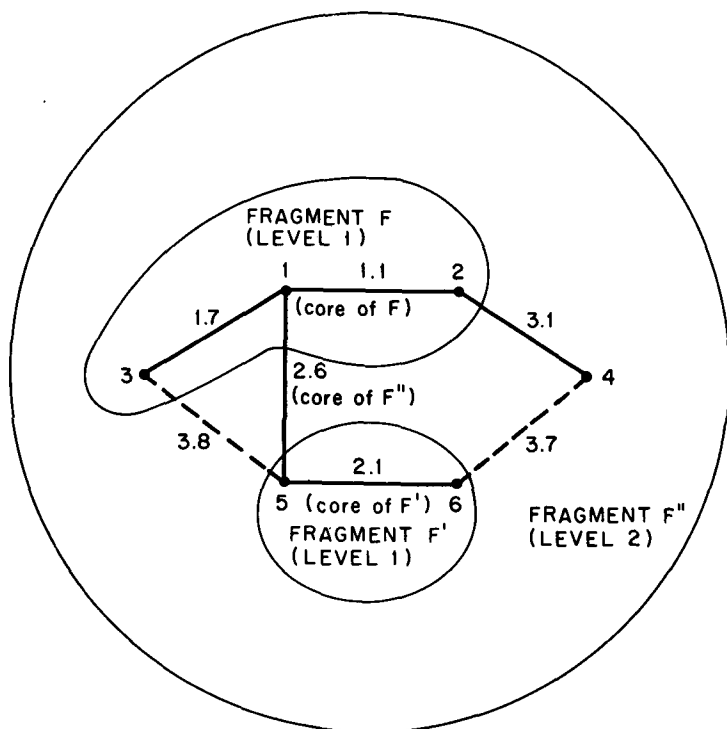
Fig. 1.   Fragments and cores.

over this edge, and goes into the state *Found*, waiting for a response from the fragment at the other end of the selected edge.

Now consider the problem of how the nodes in a nonzero-level fragment cooperate to find the minimum-weight outgoing edge. Suppose a new fragment at level L has just been formed by the combination of two level $(L - 1)$ fragments with the same minimum-weight outgoing edge, which becomes the *core* of the new fragment. The weight of this core edge is used as the identity of the fragment.

The two nodes adjacent to the core start the new cycle by broadcasting an *Initiate* message to the other nodes of the fragment. This message is sent outward on the fragment branches and is relayed outward by the intermediate nodes on the fragment. The initiate message carries the new fragment level and identity as arguments, providing all nodes in the fragment with this information. The initiate message also contains the argument *Find*, which places the node in the *Find* state as discussed later. If other fragments at level $(L - 1)$ are waiting to connect into the nodes of the new level L fragment, the initiate message is passed on to them also, putting them into the new fragment. The initiate message is also passed on to level $(L - 1)$ fragments waiting to connect into these new nodes, and so forth.

When a node receives this initiate message, it starts to find its minimum-weight outgoing edge. The difficulty here is that a node does not know which edges are outgoing. This difficulty is resolved as follows: Each node classifies each of its adjacent edges into one of three possible states: *Branch*, if the edge is a branch

in the current fragment; *Rejected*, if the edge is not a branch but has been discovered to join two nodes of the fragment; and *Basic* if the edge is neither a branch nor rejected.

In order to find its minimum-weight outgoing edge, a node picks the minimum-weight *Basic* edge and sends a *Test* message on it. The test message carries the fragment identity and level as arguments. When a node receives such a test message, it checks whether or not its own fragment identity agrees with that of the test message. If the identities agree, then (subject to a slight exception) the node sends the message *Reject* back to the sender of the test message, and both nodes put the edge in the *Rejected* state. The node sending the test message then continues by testing its next-best edge. The exception above is that, if a node sends and then receives a test message with the same identity on the same edge, it simply rejects the edge without the reject message; this reduces the communication complexity slightly.

If the node receiving a test message has a *different* identity from that of the test message, and if the receiving node's fragment level is greater than or equal to that of the test message, then the message *Accept* is sent back to the sending node, certifying that the edge is an outgoing edge from the sending node's fragment. If, on the other hand, the receiving node's fragment level is less than that of the test message, then the receiving node delays making any response until its own level increases sufficiently. The major reason for this delay feature is that, after a lower level fragment combines into a higher level fragment, the outgoing nodes of the fragment do not find out about the change for an uncertain period (in fact, until they receive a new *Initiate* message). An important property of the algorithm is that the fragment identity of a node changes when and only when the level increases; furthermore, a given fragment identity can occur at only one level. These properties are intuitively clear from the preceding discussion of how fragments combine and can be established rigorously by induction on any allowable time ordering of events in the algorithm. From these properties of fragments, we see that when a node A sends an *Accept* message in response to B's *Test* message, then the fragment identity of A differs, and will continue to differ, from B's *current* fragment identity.

We have just described how each node in a fragment eventually finds its minimum-weight outgoing edge, if any. The nodes must now cooperate, by sending *Report* messages, to find the minimum-weight outgoing edge from the entire fragment; if no node has outgoing edges, the algorithm is complete, and the fragment is the MST. In particular, each leaf node of the fragment, that is, each node adjacent to only one fragment branch, sends the message *Report*(W) on its inbound branch; W is the weight of the minimum-weight outgoing edge from the node, and W is infinity if there are no outgoing edges. Similarly, each interior node of the fragment waits until it has both found its own minimum-weight outgoing edge and received messages on all outbound fragment branches. The node then denotes the edge (either outgoing edge or outbound fragment branch) on which the smallest of these weights, W, was found as *best-edge*, and the node sends *Report*(W) on its inbound branch. When a node sends the *Report* message, it also goes to the state *Found*, indicating the completion of its role in finding the fragment's minimum-weight outgoing edge. Eventually, the two nodes adjacent

to the core send report messages on the core branch itself, allowing each of these nodes to determine both the weight of the minimum outgoing edge and the side of the core on which this edge lies.

After the two core nodes have exchanged *Report* messages, the best edges saved by the fragment nodes make it possible to trace the path from the core to the node having the minimum-weight outgoing edge. The message *Change-core* is then sent over each branch of this path, and the inbound edge for each of these nodes is changed to correspond to *best-edge*. When this message reaches the node with the minimum-weight outgoing edge, the inbound edges form a rooted tree, rooted at this node. Finally, this node sends the message *Connect*(L) over the minimum-weight outgoing edge; L is the level of the fragment.

If two fragments at level L have the same minimum-weight outgoing edge, then each sends the message *Connect*(L) over the edge, one in each direction, and this causes the edge to become the core of a level $(L + 1)$ fragment and causes new initiate messages with the new level and fragment identity to be sent out. This rule for forming new fragments ensures that a level $(L + 1)$ fragment always contains at least two level L fragments $(L \geq 0)$; it follows that level L fragments contain at least 2^L nodes, and thus that $\log_2 N$ is an upper bound on fragment levels.

Finally, consider what happens when a connect message from a node n, in a low-level fragment with level L and identity F, reaches a node n' in a higher level fragment with level L' and identity F'.

Due to our strategy of never making a low-level fragment wait, node n' immediately sends an initiate message with identity and level parameters F' and L' to n. If node n' has not yet sent its report message at the given level, fragment F simply joins fragment F' and participates in finding the minimum-weight outgoing edge from the enlarged fragment. If, on the other hand, node n' has already sent its report message, then we can deduce that an outgoing edge from node n' has a lower weight than the minimum-weight outgoing edge from F. This eliminates the necessity for F to join the search for the minimum-weight outgoing edge. These two cases are distinguished by sending the node state, either *Find* or *Found*, in the initiate message. The nodes in fragment F go into state *Find* or *Found* depending on this parameter of the initiate message, and they send *Test* messages only in the *Find* state.

We now outline a proof that the algorithm is correct. In view of Properties 1 and 2 it is sufficient to verify that the algorithm does indeed find minimum-weight outgoing edges from fragments and that the waiting does not lead to deadlocks. The previous description of the algorithm should convince the reader that the edge on which a message *Connect*(L) is sent *is* the minimum-weight outgoing edge for the level L fragment consisting of all nodes that received the initiate message with the given identity (note that the fragment corresponding to a given fragment identity can grow as lower level fragments join the given fragment).

To show that deadlocks do not exist, consider the set of fragments in existence at any given time, excluding zero-level fragments consisting of sleeping nodes. Assume the algorithm has started but has not finished, so that the above set of fragments is nonempty and each fragment has a minimum-weight outgoing edge. Out of the lowest level fragments in the set, consider one with the smallest

minimum-weight outgoing edge. Any test message from that fragment either will wake up a sleeping zero-level fragment or will be responded to without waiting. Similarly, a connect message from that fragment either will wake up a sleeping zero-level fragment, or will go to a higher level fragment (with an immediate initiate response), or will go to a fragment of the same level with the same minimum-weight outgoing edge, leading to a new higher level fragment. Since the assumed system state was arbitrary, we see that deadlocks do not exist.

The program in the appendix has also been tested (using a FORTRAN implementation) on a variety of network topologies.

We now briefly describe a modification of the algorithm that can be used for nondistinct edge weights and that does not require $2E$ extra messages for appending the adjacent node identities to the edge weight. In the modification, fragments are identified by node identities, which are ordered and distinct. A minimum-weight outgoing edge from a fragment is found as before, and a connect message is sent over that edge as before. The new feature is that a connect message on edge e from fragment F to F' is later canceled if (1) both fragments are at the same level and $F > F'$; (2) some fragment F'' at the same level has sent a connect message to F and $F'' < F$; (3) an initiate message has not already been sent back on edge e. When a connect message is canceled, the node that sent it increases its level and sends out a new initiate message, in this case joining fragments F and F''.

4. COMMUNICATION COST ANALYSIS

We determine here an upper bound on the number of messages exchanged during the execution of the algorithm. Note that the most complex message contains one edge weight, one level between zero and log N, and a few bits to indicate message type.

Since an edge can be rejected only once, and each rejection requires two messages, there are at most $2E$ test or reject messages leading to rejections.

Next, while a node is at a given level except the zeroth and the last, it can receive at most one initiate and one accept message. It can transmit at most one successful test message, one report message, and one change-root or connect message. Since $\log_2 N$ is an upper bound on the highest level, a node can go through at most $(-1 + \log N)$ levels not counting the zeroth and last, and this accounts for at most $5N(-1 + \log N)$ messages.

At level zero, each node can receive at most one initiate message and can transmit at most one connect message. At the last level, each node can send at most one report message. This adds less than $5N$ messages to our grand total, which becomes $5N \log N + 2E$.

Note that, if the number of nodes in the graph is initially unknown, as we have implicitly assumed, then no distributed algorithm can find the MST with fewer than E messages; if there is an edge over which no message is sent, then there might have been a node at the center of that edge, causing the algorithm to fail.

5. TIMING ANALYSIS

Although it appears that the algorithm typically allows a large amount of parallelism in messages, it is not difficult to find examples such as Figure 2 in
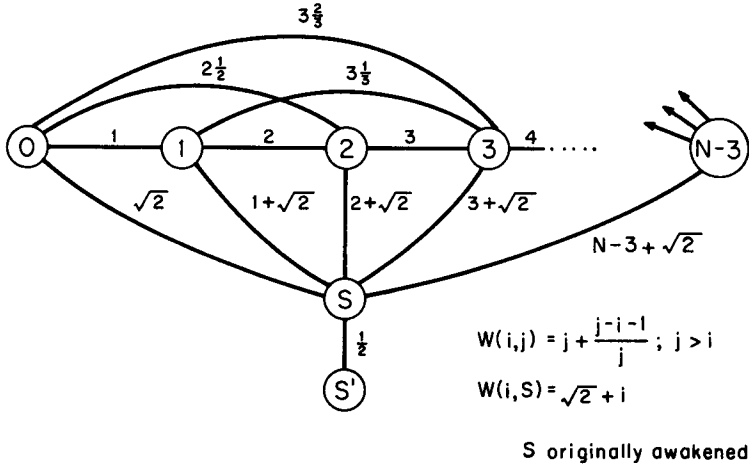
Fig. 2. Example when algorithm requires $N(N - 1)$ message time units.

which most of the messages are sent sequentially. In particular, if node S originally wakes up, each node j sequentially both sends a test message to each $i \leq j - 2$ and receives a reject before any $j' > j$ is awakened. This leads to $N(N - 1)$ sequential messages.

If time is important, it is preferable to awaken all nodes originally; this can be done in at most $N - 1$ time units assuming each message transmission requires at most one time unit. We now show that, with this assumption of initial awakening, the algorithm requires at most $5N \log_2 N$ time units. Note first that, by time N, each node will be awakened and will have sent a connect message. By time $2N$, each node must be at level one through the propagation of initiate signals.

We proceed by induction on the level numbers, showing that it takes at most $5lN - 3N$ time units until all nodes are at level l. This is true for $l = 1$; assume it true for l. At level l, each node can send at most N test messages which will be answered before time $5lN - N$. The propagation of the report, change-root and connect, and initiate messages can take at most $3N$ units, so that by time $5(l + 1)N - 3N$ all nodes are at level $l + 1$. At the highest level, $l \leq \log_2 N$, only test, reject, and report messages are used; so the algorithm is complete by time $5N \log_2 N$.

A worst-case $O(N \log N)$ is possible, as is shown by the example in Figure 3, where the handle and the head both contain $N/2$ nodes. If the edge weights in the handle increase as one gets away from the head, then all nodes in the handle will be in the same fragment at level one. If processing the head requires log $N/2$ levels and if at each level a fragment joins the handle, which can happen, then the time until completion will be $O(N \log N)$.

APPENDIX

The algorithm to follow is obeyed by each node and consists of a list of the responses to each type of message that can be generated. In addition, the response to a spontaneous awakening of the node is given. Each node is assumed to queue
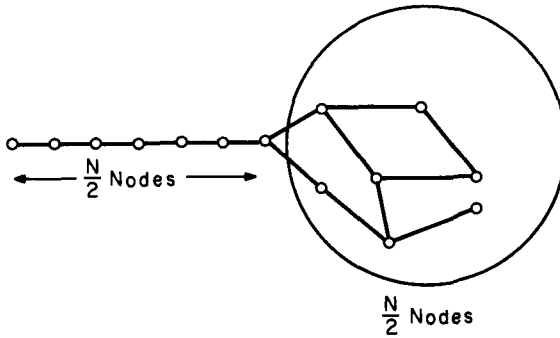
Fig. 3. Example when $O(N \log_2 N)$ message time units are required with initial general awakening.

$\dfrac{N}{2}$ Nodes

$\dfrac{N}{2}$ Nodes

the incoming messages and to respond to them in first-come, first-served order. One particular response is to place the message back on the end of the queue for delayed servicing, but, aside from this, each response is completed before the next is started. Each node, of course, maintains its own set of variables, consisting of its state (denoted by SN and assuming possible values *Sleeping*, *Find*, and *Found*) and the state of the adjacent edges. The state of edge j is denoted by $SE(j)$ and can assume the possible values *Basic*, *Branch*, and *Rejected*. It is possible for the edge states at the two nodes adjacent to the edge to be temporarily inconsistent. Initially for each node, $SN = Sleeping$ and $SE(j) = Basic$ for each adjacent edge j. Each node also maintains a fragment identity FN, a level LN, and variables *best-edge*, *best-wt*, *test-edge*, *in-branch*, and *find-count*, all of whose initial values are immaterial. There is also an initially empty first-come first-served queue for incoming messages. Finally, the weight of each adjacent edge j is denoted $w(j)$.

The Algorithm (As Executed at Each Node)

(1) Response to spontaneous awakening (can occur only at a node in the sleeping state)
    execute procedure *wakeup*

(2) **procedure** *wakeup*
    begin let m be adjacent edge of minimum weight;
        $SE(m) \leftarrow Branch$;
        $LN \leftarrow 0$;
        $SN \leftarrow Found$;
        Find-count $\leftarrow 0$;
        send *Connect*(0) on edge m
    end

(3) Response to receipt of *Connect*(L) on edge j
    begin if $SN = Sleeping$ **then execute procedure** *wakeup*;
        if $L < LN$
          then begin $SE(j) \leftarrow Branch$;
                 send *Initiate*(LN, FN, SN) on edge j;
                 if $SN = Find$ **then**
                    find-count $\leftarrow$ *find-count* $+ 1$
             end
        else if $SE(j) = Basic$
             then place received message on end of queue
             else send *Initiate*$(LN + 1, w(j), Find)$ on edge j
    end

(4) Response to receipt of *Initiate*(L, F, S) on edge j

  begin $LN \leftarrow L$; $FN \leftarrow F$; $SN \leftarrow S$; *in-branch* $\leftarrow j$;
    best-edge $\leftarrow$ *nil*; *best-wt* $\leftarrow \infty$;
    for all $i \neq j$ such that $SE(i) = Branch$
     do begin send *Initiate*(L, F, S) on edge i;
         if $S = Find$ **then** *find-count* $\leftarrow$ *find-count* $+ 1$
      end;
    if $S = Find$ **then execute procedure** *test*
  end

(5) **procedure** *test*

  if there are adjacent edges in the state *Basic*
   then begin *test-edge* $\leftarrow$ the minimum-weight adjacent edge in state *Basic*;
      send *Test*(LN, FN) on *test-edge*
    end
   else begin *test-edge* $\leftarrow$ *nil*; **execute procedure** *report* **end**

(6) Response to receipt of *Test*(L, F) on edge j

  begin if $SN = Sleeping$ **then execute procedure** *wakeup*;
    if $L > LN$ **then** place received message on end of queue
     else if $F \neq FN$ **then send** *Accept* on edge j
      else begin if $SE(j) = Basic$ **then** $SE(j) \leftarrow Rejected$;
          if *test-edge* $\neq j$ **then send** *Reject* on edge j
           else execute procedure *test*
       end
       end

(7) Response to receipt of *Accept* on edge j

  begin *test-edge* $\leftarrow$ *nil*;
    if $w(j) < $ *best-wt*
     then begin *best-edge* $\leftarrow j$; *best-wt* $\leftarrow w(j)$ **end**;
    execute procedure *report*
  end

(8) Response to receipt of *Reject* on edge j

  begin if $SE(j) = Basic$ **then** $SE(j) \leftarrow Rejected$;
    execute procedure *test*
  end

(9) **procedure** *report*

  if *find-count* $= 0$ **and** *test-edge* $=$ *nil*
   then begin $SN \leftarrow Found$;
      send *Report*(*best-wt*) on *in-branch*
    end

(10) Response to receipt of *Report*(w) on edge j

  if $j \neq$ *in-branch*
   then begin *find-count* $\leftarrow$ *find-count* $- 1$
      if $w < $ *best-wt* **then begin** *best-wt* $\leftarrow w$; *best-edge* $\leftarrow j$ **end**;
      execute procedure *report*
    end
   else if $SN = Find$ **then** place received message on end of queue
     else if $w > $ *best-wt*
       then execute procedure *change-root*
        else if $w = $ *best-wt* $= \infty$ **then** halt

(11) **procedure** *change-root*

  if $SE(best\text{-}edge) = Branch$
   then send *Change-root* on *best-edge*

else begin send *Connect*(*LN*) on *best-edge*;
                    SE(*best-edge*) ← *Branch*
       end
(12) Response to receipt of *Change-root*
         execute procedure *change-root*

REFERENCES

1. DALAL, Y.   Broadcast protocols in packet switched computer networks. Tech. Rep. 128, Dep. of Electrical Engineering, Stanford Univ., Apr. 1977 (revised version for publication in preparation).
2. DIJKSTRA, E.   Two problems in connection with graphs. *Numer. Math. 1* (1959), 269–271.
3. HUMBLET, P.A.   A distributed algorithm for minimum weight directed spanning trees. Rep. LIDS-P-1149, Laboratory for Information and Decision Systems, Massachusetts Inst. of Technology, Cambridge, Mass., Sept. 1981.
4. KRUSKAL, J.B.   On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.* 7 (1956), 48–50.
5. LAWLER, E.   *Combinatorial Optimization-Networks and Matroids.* Holt, Rinehart & Winston, New York, 1976.
6. LIU, C.L.   *Introduction to Combinatorial Mathematics.* McGraw Hill, New York, 1968.
7. PRIM, R.C.   Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36* (1957), 1389–1401.
8. SPIRA, P.   Communication complexity of distributed minimum spanning tree algorithms. In Proceedings, 2nd Berkeley Conference on Distributed Data Management and Computer Networks, Berkeley, Calif., June 1977.
9. YAO, A.C.C.   An $O(E \log \log V)$ algorithm for finding minimum spanning trees. *Inf. Process. Lett. 4* (1975), 21–23.