A distributed algorithm for minimum weight spanning trees

R. G. Gallager, P. A. Humblet and P. M. Spira

Prepared by:
Guy Flysher and Amir Rubinshtein

Preface

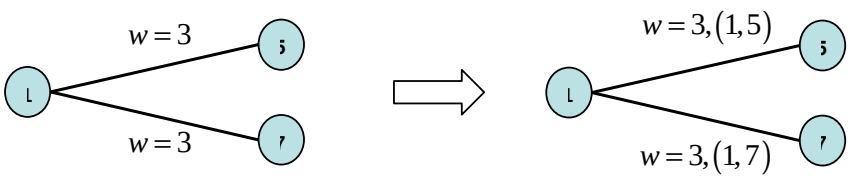
In this document we will review Gallager, Humblet and Spira's distributed algorithm for minimum weight spanning trees.
In centralized algorithms (as opposed to distributed ones) we know of many simple algorithms that find a MST. In the distributed model however, things are more complex since there is no "entity" that knows the topography of the entire graph. In this algorithm we will assume that each node knows only the weights of its adjacent edges. The algorithm displayed will be uniform (all nodes run the exact same code) and will require up to $5N\log_2 N + 2E$ messages where each message takes not more than $\log_2 w_{\max} + \log_2 8N$ bits ($w_{\max}$ being the maximum weight in the graph).

1. Problem definition

In the problem of the minimum spanning tree, we are given an undirected graph with *N* nodes and *E* edges. Each node represents a processor (on which the algorithms code is executed) and each edge represents a bidirectional communication channel (messages can be transmitted independently in both directions of an edge). We will assume that all messages arrive after a finite but unpredictable delay, without errors and in sequence (FIFO). We will also associate a weight which is finite and distinct with each edge.
The question arises, why is the demand for distinct weights needed? This question will be answered in section 2.2. Another question that might arise is how can one "create" distinct weights when the original weights are not distinct.

1.1 *Making the weights distinct*

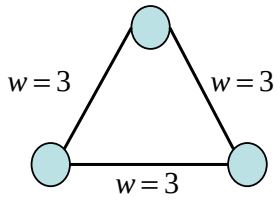
If all the nodes have distinct identities then we can use these identities to make the edges' weights distinct. We do this by concatenating the identities of the neighboring nodes of each edge to its weight.



$$3,(1,5) < 3,(1,7)$$

We arbitrarily choose if throughout the algorithm the smaller identity appears first or the larger one in order to assure both nodes are in agreement over the edge's weight.
A natural approach is to try and have the nodes select random identities and send a message with these identities to their neighbors notifying them of the selection. Once all nodes have chosen distinct identities we can apply the method shown above. While the expected number of tries needed to reach a situation where all the identities are distinct can be as small as we want (By having the identities chosen from a larger set), the probability of failure after any amount of attempts is not zero. Therefore, this approach will leave us with a random algorithm where we can only bound the expected complexity.

If we do not have distinct weights nor distinct identities, no deterministic uniform algorithm exists for the MST problem.
We can easily be persuaded of this fact by the following example:



Every two edges make up a MST. Since the algorithm is uniform and each node has exactly the same information as the others, clearly there is no way to choose specific two of the three.

2. Review of spanning trees

We define a *fragment* of a MST as a sub-tree (that is, a connected set of nodes and edges of the MST). Given a graph and a fragment of a MST, we define an *outgoing edge* as an edge with exactly one end in the fragment.

2.1 - *Property 1*: Given a fragment of a MST, let *e* be a minimum weight outgoing edge of the fragment. Joining *e* and its adjacent non-fragment node to the fragment yields another fragment of **a** MST (not necessarily the same MST the original fragment was part of).

Proof: Suppose e is not in any MST containing the fragment. By definition adding any edge to a MST will create a cycle, in particular adding *e* will create a cycle. *e* is an outgoing edge therefore it leads out of the fragment. Since the cycle passes through the fragment (it has a part

which is outside the fragment and a part which is inside the fragment) there must be another outgoing edge with respect to this fragment which is part of the cycle, we shall mark it as *a*. Since *e* is the fragment's minimum weight outgoing edge we can deduce $w(e) \leq w(a)$. If we remove *a*, we are left with a connected graph without cycles therefore a tree. We did not increase the total weight of the tree (by adding *e* and removing *a*) therefore this new tree must also be a MST. ■

2.2 - *Property 2*: If all edges of a connected graph have different weights, then the MST is unique.

Proof: Suppose there are 2 different MSTs.
Let *e* be the minimum weight edge such that *e* belongs to one MST (*T'*) and not the other (*T*). Since *T* is a tree, adding *e* will create a cycle. At least one of the edges along the cycle does not appear in *T'*. This is true since if all the edges on this cycle appear in *T'*, then *T'* contains a cycle which is a contradiction to *T'* being a tree. Removing one of the edges along the cycle which do not appear in *T'* will leave us with a tree whose total weight is lower than that of *T* (since *e* is the minimum weight edge that appears in only one of the trees and all weights are distinct) in contradiction to *T* being a MST. ■

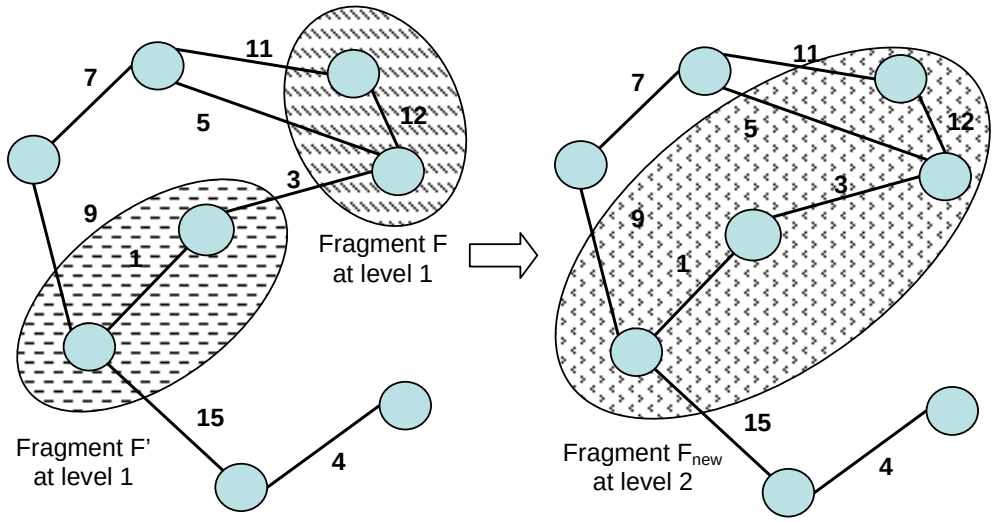
3. The algorithm

3.1 - *The idea*

In this section we will talk about the general idea of the algorithm. The details about how the actions described below are accomplished appear in section 3.3.

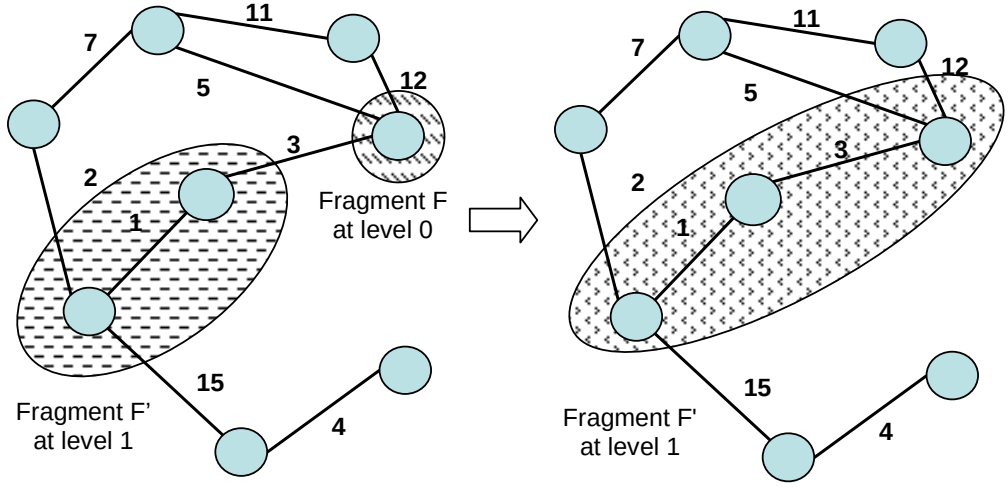
The idea of the algorithm will be based on the two properties above. We start with *N* fragments containing a single node. Using property 1 we can enlarge the fragments. Property 2 assures us that when two fragments have a common node; their union is also a fragment. This follows from the fact that both original fragments were part of a MST and the "new" fragment is also part of a MST. Since there is only one possible MST they both belong to the same MST and so does the new combined fragment.
In the algorithm to follow each fragment finds its minimum weight outgoing edge (asynchronously) and tries to combine with the fragment on the other side of the edge. The way this is done depends on the fragments' *level*.
A fragment containing a single node is at *level* 0.

- Case 1: If two fragments have the same *level L* and the same minimum weight outgoing edge, they combine to form a new fragment at level (*L*+1).



The shared minimum weight outgoing edge is called the new fragment's *core* and its adjacent nodes are called the *core nodes*.

- Case 2: If fragment *F* is at level *L* and fragment *F'* at level *L'*>*L* is at the other end of *F*'s minimum outgoing edge, then fragment *F* is absorbed by *F'* (the level remains *L'*).



The rule of thumb (The logic behind it will be explained in section 3.7) is: A low level component is never kept waiting.

Comment: If fragment *F* is at level *L* and fragment *F'* at level *L'<L* is at the other end of *F*'s minimum outgoing edge then no absorption occurs.

3.2 - *Preliminaries*

The nodes: Each node can be in one of three states: *Sleeping*, *Find* and *Found* where *Sleeping* is the initial state of all nodes. The node will be in state *Find* when looking for the minimum weight outgoing edge of the fragment it belongs to and in state *Found* at all other times. The algorithm starts when one or more nodes spontaneously awaken. A node may also be woken up by the receiving of a message.

The edges: For each edge *e*=(*u*,*v*) nodes *u* and *v* will have a variable with the edge's state which can be one of the following three:

1. *Rejected* – The edge is not part of the MST.
2. *Branch* – The edge is a part of the MST.
3. *Basic* – The algorithm has not yet decided whether the edge is part of the MST or not.

3.3 - Finding the minimum weight outgoing edge

A zero level fragment (which always consists of a single node) performs the following algorithm upon awakening:

1. It chooses its minimum weight adjacent edge.
2. It marks it as a branch of the MST.
3. It sends a "*Connect*(0)" message on this edge (we will explain this in greater detail later).
4. It goes into state Found (waiting for a reply from the node on the other side).

According to property 1 and property 2 we know that the minimum weight outgoing edge of a fragment consisting of a single node belongs to the unique MST.

In the case of nonzero fragment let us assume that two *L*-1 fragments have just merged to form a new fragment at level *L* where the weight of the new component's *core*, *w*(*e*) is the new component's identity. An *initiate* message containing the fragments identity, its level and the argument *find* is sent from the *core nodes* along the fragment's *branch* edges (and only these edges). The message is passed on to all the fragments nodes through flooding (similar to the PI protocol). Upon

receiving such a message a node becomes aware of the merging which has occurred and of the new identity and level of the fragment.
An important observation is that if there are components of level *L*-1 waiting to connect to the fragment, the nodes floods the *initiate* message to them as well (and in the same way if there are other *L*-1 fragments waiting to connect to it, it floods the message to them as well). The logic behind this is the rule of thumb we mentioned earlier. This *L*-1 fragment asked to be connected to one of the two *L*-1 level fragments which have just merged and was not yet answered, which is allowed. Now however it has become a lower level fragment since the new fragments level is *L*. Therefore it must be absorbed immediately. The initiate message does exactly that.

Each node, after receiving the *initiate* message with the argument find starts looking for its minimum weight outgoing edge. It does so by sending out a message called *test* on the minimum weight *basic* edge adjacent to it. The *test* message contains the fragments id and level. A node receiving a *test* message does the following:

- If the id in the message is the same as the id of the fragment the node belongs to – a *reject* message is sent back (*)
- If the id in the message differs from the id of the fragment the node belongs to and the level in the message is lower or equal to that of the node's fragment – an *accept* message is sent back.
- If the id in the message differs from the id of the fragment the node belongs to and the level in the message is higher than that of the node's fragment – no reply is sent until the situation has changed.

(*) an exception to this rule is when the node that sent the *test* message receives a *test* message along the same edge with the same id. In such a case a reject message is not sent (the other side would get the message sent by this node and know that this edge is not an outgoing edge). This is done in order to obtain a small decrease in message complexity.

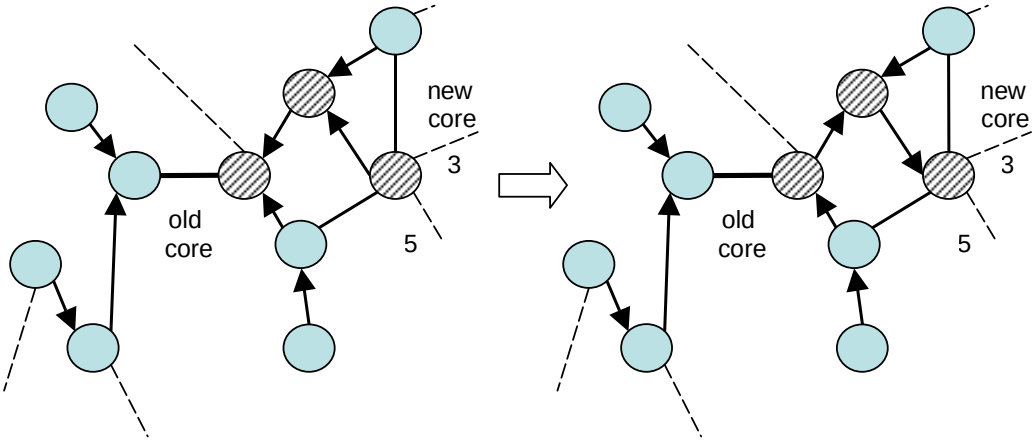
If the node which sent the *test* message received a *reject* message, it marks the edge as *rejected* and moves on to try the next *basic* edge with the minimum weight adjacent to it.

After a node has found its minimum weight outgoing edge (or found that it has no adjacent *basic* edges left) it waits for a *report* message from all the nodes it flooded the *initiate* message to (similar to the acknowledgment part in PIF). After receiving all of these messages it sends back a *report* message back to the node from which it received the

initiate message. The *report* message contains the minimum between the weight of the outgoing edge found by the node itself and the minimum found by all the nodes which sent the *report* messages back. This process continues until the *report* messages reach the *core nodes*. The two *core nodes* exchange *report* message and at this point, both *core nodes* know through which of their edges the path to the minimum outgoing edge passes.

3.4 – Changing the core

After both *core nodes* know through which edge the minimum outgoing edge can be reached, the *core node* which is "closer" to the minimum outgoing edge sends a message called *change-core*. The *change-core* message is sent to all the nodes on the path from the *core node* to the minimum outgoing edge (each node knows through which edge it should pass the message according to the report messages). When the *change-core* reaches the node which found the minimum outgoing edge, this node sends a *connect* message along this edge which contains the fragment's *level*. This message notifies the fragment on the other side of the edge that this fragment whishes to merge/be absorbed by it.
We only pass the *change-core* message through the nodes on the path from the *core-nodes* to the minimum outgoing edge and the question arises, do the other nodes not need to know of this change. The answer is no as can be seen in the image below.



As we can see changing the "direction" of the edges along the path will cause all messages being sent to the old *core* to be diverted to the new *core* (in reality a change in direction is simply the nodes knowing of the change, sending messages along the edge in this direction).

3.5 - Connecting fragments

The rules for joining fragments in the algorithm are:

If two fragments at *level L* have the same minimum outgoing edge, each sends the message *connect(L)* over this edge, one in each direction (this was mentioned earlier in section 3.4, when we discussed what happens after change-core reaches the nodes adjacent to the minimum outgoing edge). This edge now becomes the core for the component with level (L+1). New initiate messages are sent by the new core's nodes and the process of finding the minimum weight outgoing edge starts over.

What happens when a *connect* message from node *n* in a low level fragment with level *L* and identity *F* reaches a node *n'* in a higher level *L'* with identity *F'*? As we said before a lower level fragment never waits. Node *n'* immediately sends back an *initiate* message with parameters *F'* and *L'* to *n* (which is flooded to all nodes in *F*, effectively causing fragment *F* to be absorbed in *F'* by updating the information in its nodes to that of the new fragment).

The message *initiate* contains, in addition to the mentioned parameters (*level* and fragment identity) a third parameter: the state of the node sending it (i.e. find or found).
If the node sending the initiate has not yet sent its report, then its state must be *Find*, and the parameter it sends is therefore *Find*. This causes fragment *F* to join the search (The node will not send report until it also receives a *report* message from the joining fragment).
If the node sending the initiate had already sent its report, then its state and the parameter it sends must be *Found*. Since it had already sent its report, the edge on which the *initiate* was sent was not its minimum outgoing edge. Therefore, fragment *F* does not need to join the search. Indeed, notice that it's not possible that fragment *F* has an outgoing edge with a lower weight than the minimum outgoing edge of *F'* (the edge connecting *F* to *F'*), since then *F* would have sent *connect* over this edge.

3.6 - Correctness

To prove correctness of the algorithm, we need to show the following:

- Fragments indeed find their minimum-weight outgoing edges
- No deadlocks

The paper does not prove the first claim formally. Indeed, proving correctness in distributed algorithms sometimes lacks the formality we are used to in centralized algorithms.

As for deadlocks, notice that some nodes may wait for responses after they sent *test* or *connect*. This is the reason that the issue of deadlocks is brought up here. Deadlock could happen if all nodes are waiting for some action to take place.
Assume that the algorithm has started but not yet finished. This means that there exist fragments, which are not sleeping nodes. Consider the fragment with the lowest *level* and with the smallest minimum-weight outgoing edge, excluding zero-*level* sleeping nodes (there exists such fragment since the algorithm has started). The following cases are possible:

When sending a *test* message:
- case 1: a sleeping node awakens, and that node may now start its algorithm
- case 2: the *test* is responded to without waiting (since it reaches a higher or equal level fragment – remember we chose the fragment with the lowest level)

When sending *connect* :
- case 1: a sleeping node awakens, and that node may now start its algorithm
- case 2: the *connect* reaches a higher level fragment
- case 3: the *connect* reaches a fragment with the same level

In the last two cases, a higher level fragment is formed, and therefore there exists some fragment which is now the "new" fragment with the lowest level and with the smallest minimum-weight outgoing edge (in case 3 this occurs since its outgoing edge is the one with the lowest weight among outgoing edges of all of the fragments at this level. Therefore it is also a minimum weight outgoing edge of the fragment on the other side).
We went over all the cases in which a deadlock could happen, and showed that in each one either there is no waiting, or a new node awakens.

3.7 - Communication Cost
Notice that in a level L component there are at least 2^L nodes (easy to prove by induction). If this is the case, the maximum level possible for any fragment is $\log_2 N$.

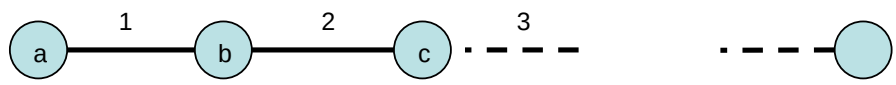
We now determine an upper bound for the number of messages.
Recall that the types of messages used by the algorithm are:
- reject
- accept

- test (partitioned here for analysis purposes to successful test, which are replied by an *accept* and failed test, which are replied by *reject*).
- initiate
- connect
- report
- change-core

First we give an upper bound on the number of messages of type *reject* and failed *test* throughout the whole execution of the algorithm. Then, we consider a single node at a specific level, and count the outgoing messages of type *connect*, *report*, *change-core*, and successful *test*, and the incoming messages of type *accept* and *initiate*. This will give us an upper bound on the number of messages transmitted from or to that node at that level.

An edge can be rejected only once. Rejection requires two messages: failed *test* and *reject* (or possibly two failed *tests*). So we have $2E$ messages so far counted.
A node in any level beside 0 and the last 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-core* or *connect* message. A node can go through at most $\log_2 N - 1$ levels (not counting level 0 and the last). These sum up to another $5N(\log_2 N - 1)$ messages.
At level 0, a node can receive at most one *initiate* message, and transmit at most one *connect* message ($2N$ messages). At the last level, each node can send at most one *report* message (N messages).
This counting brings us to $2E + 5N(\log_2 N - 1) + 3N < 2E + N(\log_2 N)$
Finally, notice that the most complex message in the algorithm contains one edge weight, one level between 0 and $\log_2 N$, and 3 bits to indicate the message type. The total number of bits a message requires is therefore $\log_2 w_{\max} + \log_2 8N$.

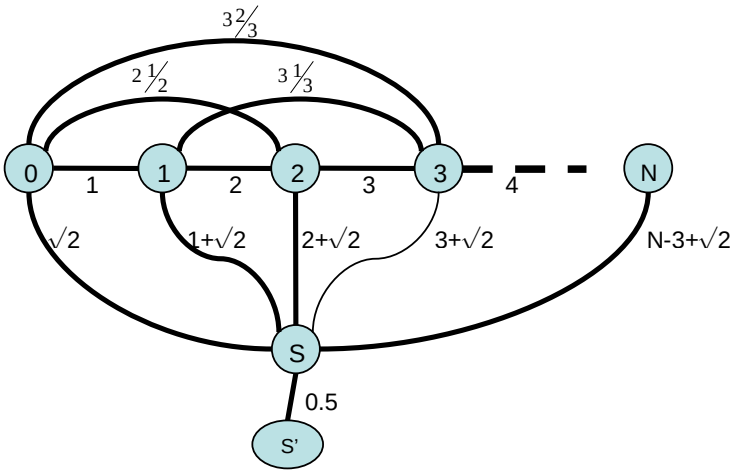
We said before that a lower level fragment never waits on a higher level fragment and that a higher level fragment is made to wait by a lower level fragment. What if we reversed things? What if a higher *level* fragment at level *L'* will not wait on a lower level fragment at level *L*?
To answer this, consider the following example:



If the node *a* wakes up first, it sends connect to *b*, then *a* and *b* form a new fragment in *level* 1. From now on, this fragment will send *test* from its rightmost node to the node on its right. That node will wake up, and sent connect back to the node which sent the test. Later it will also send *accept* since they share the minimum weight outgoing edge and the high *level* fragments are not made to wait. But since we are letting the small level fragment wait, the 1st *level* fragment will forward the *report* to the left, until it reaches the core (which remains the edge with weight 1 at all times). Now, a *change-core* message will be forwarded from the *core node* on the right, and will result in an *initiate* message when it reaches the "front" of the fragment. In other words a *change-core* message is transmitted throughout the fragment in one direction, and an initiate is transmitted back to the other direction. This happens in "cycles", each in which the fragment expands one node to the right. So the number of messages sent is the sum of an arithmetic series and therefore is $\Theta\left(N^2\right)$.

3.8 - Timing Cost

In the general case we can easily find examples where the time complexity is $\Theta\left(N^2\right)$. An example for such a case as appears in the paper:



Node S originally awakens, and forms a 1 level fragment with node S'. Node 0 is now awakened, and forms a separate 1st *level* fragment with node 1. The last fragment is now expanded each time in one node to the right; it first absorbs node 2, then node 3, and so on. In this example, each node sequentially sends test and receives reject from every i≤ j-2 before j'>j is awakened, and therefore the time complexity is $\Theta\left(N^2\right)$.

However, if all nodes awaken originally, time complexity is O(*N*log*N*).

Awakening all nodes can be done at the beginning in at most N-1 time units. By time N – each node will wake up and send *connect*, and by time $2N$ each node will be at level 1.

Claim: By time $5lN - 3N$ each node is at *level* l.
Proof by induction:

True for l=1.
Assume all nodes are at level l by $5lN - 3N$.
- each node can send up to N *test* messages (N time units)
- these messages are answered. (N time units)
- all nodes send report to the core. (N time units)
- the core nodes spread out change-core. (N time units)
- initiate may be sent as a result (N time units)

The longest time (if all the above happens sequentially) it takes for the above messages to be sent is 5N time steps.
$(5lN - 3N) + 5N = 5(l+1)N - 3N$.

At the last level only *test, reject* and *report* messages are sent (total $3N$).
Since $l \le \log_2 N$, the algorithm is complete by time:

$$5lN - 3N + 3N \le 5N \log_2 N$$

This concludes the review of the distributed minimum spanning tree algorithm.
Attached in the next 2 pages is the actual code which is executed at each of the nodes.

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) ← *Branch;*
LN ← 0;
SN ← *Found;*
Find-count ← 0;
send *Connect(O)* 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)* ← *Branch;*
send *Initiate(LN, FN, SN)* on edge j;
if *SN = Find* **then**
find-count ← *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* ← *L; FN* ← *F; SN* ← *S; in-branch* ← *j;*
best-edge ← *nil; best-wt* ← ∞ *;*
for all i ≠ j such that *SE(i) = Branch*
do begin send *Initiate(L, F, S)* on edge i;
if *S = Find* then *find-count* ← *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* ← the minimum-weight adjacent edge in state *Basic;*
send *Test(LN, FN)* on *test-edge*
end
else begin *test-edge* ← *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* ≠ *FN* then send *Accept* on edge j
else begin if *SE (j)* = *Basic* then *SE (j)* ← *Rejected;*
if *test-edge* ≠ *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* ← *nil;*
if w(j) < *best-wt*
then begin *best-edge* ← *j; best-wt* ← *w(j)* end;
execute procedure *report*
end

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

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

(9) procedure *report*

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

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

if j ≠ *in-branch*
then begin *find-count* ← *find-count - 1*
if w < *best-wt* then begin *best-wt* ← *w; best-edge* ← *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-core*
else if w = *best-wt* = ∞ then halt

(11) procedure *change-core*

if *SE (best-edge) = Branch*
then send *Change-core* on *best-edge*
else begin send *Connect(LN)* on *best-edge;*
SE (best-edge) ← *Branch*
End

(12) Response to receipt of *Change-core*

execute procedure *change-core*