Optimal Distributed Algorithm for Minimum Spanning Trees Revisited

Michalis Faloutsos* and Mart Molle†

Abstract

In an earlier paper, Awerbuch presented an innovative distributed algorithm for solving minimum spanning tree (MST) problems that achieved optimal time and message complexity through the introduction of several advanced features. In this paper, we show that there are some cases where his algorithm can create cycles or fail to achieve optimal time complexity. We then show how to modify the algorithm to avoid these problems, and demonstrate both the correctness and optimality of the revised algorithm.

1 Introduction

Given an undirected graph G with N nodes and E edges, with weights assigned to each edge, we want to find a spanning tree for which the combined weight of all its edges is minimized, denoted an **MST** in the sequel. Furthermore, we want to use a distributed algorithm to find that MST by placing a processor at each node and treating each edge as a bidirectional and error-free communication channel, over which the nodes can exchange messages among themselves. We assume that initially, all nodes of the graph are "awake" but none of them has any special status (nor are any of them aware of the network topology except for their adjacent edges) so we cannot simply send all the information about G to a distinguished node that solves the problem and broadcasts the answer to the rest of the graph. Fortunately, our task is made easier because it is well known that the MST problem can be solved by a "greedy" algorithm, which can generate an optimal solution without backtracking.

For the general graph, the distributed MST problem requires at least $\Omega^1(E+N\cdot\log(N))$ messages, where we count the transmission of one message across one edge as our unit of "cost". In addition, there are graphs where the time complexity is at least $\Omega(N)$ assuming that each message delivery takes one time unit. Recent research in the area suggests that the diameter of a network is a more accurate parameter for describing the time complexity [GKP93]. However, it was also proven [SB95] that a tighter bound for the termination time of the algorithm [GHS83], presented below, is $O((D+d)\cdot\log(N))$, where d is the diameter of the resulting MST and D is the maximum degree of the nodes. Since the algorithm we present is based on this algorithm, the time complexity of our algorithm has to be of the same order, if not better. For the rest of this paper, *we will consider $O(N)$ to be the optimal time bound* and leave the resolution of this argument to future research, since we have reasons to believe that our algorithm will be of competitive complexity independently of the way optimality will be defined.

*University of Toronto, Dpt. of Computer Science, Toronto M5S 1A4, Ontario, Canada. **mfalou@cs.toronto.edu**. Supported by the University of Toronto through the UofT Open and the Connnaught scholarship programs.

†University of Toronto (formerly) and University of California at Riverside (currently), Dept. of Computer Science, Riverside, CA 92521. **mart@cs.ucr.edu**. Supported by the Natural Sciences and Engineering Research Council of Canada under grant #A5517.

[1] With Ω we denote the lower bound of the asymptotic complexity.

1.1 Basic Algorithm of Gallager, Humblet and Spira

In their pioneering paper [GHS83], Gallager, Humblet and Spira introduced the distributed MST problem and presented an algorithm that has formed the basis of subsequent work in the area, for example [CT85], [Gaf85], [Awe87] and [Fal95]. In their algorithm, each node is initially the **root** of its own **fragment** (a trivial connected subgraph of the MST) and all the edges are **Unlabeled**.

Thereafter, adjacent fragments join to form larger fragments by labeling their intermediate edge as a **Branch** of the MST. The new branch is chosen by the root of one (or possibly both) of the fragments, as the **minimum outgoing edge** (or **MOE**) for the entire fragment. This fragment MOE is determined by broadcasting an *initiate* message to all nodes in the fragment, asking them to send a *report* message with their local MOE to the root (**Finding procedure**). Each node determines its local MOE by testing its Unlabeled edges, minimum weight first, until it finds one that leads to another fragment (**Testing procedure**). Any edges that are found to connect to nodes in the same fragment are labeled as **Rejected**, and are subsequently ignored. Each node gathers the reports of its children and reports the minimum MOE found by itself or its children (**Reporting procedure**). Finally the root sends a *changeRoot* message to the node adjacent to the MOE, appointing it as the new **leader** of the fragment. The leader sends a *connect* message along that edge and joins with the other fragment.

Even the basic algorithm presented in [GHS83] contains several subtleties. First, each fragment has a **level**, L, in addition to its unique fragment identifier, F. The fragment levels are used to make fragment joining less symmetric, so that certain types of "one-sided" joins can be permitted without the risk of forming cycles. If two adjacent fragments discover that they share a common MOE and wish to label that edge as a branch, then it is clear that the resulting "two-sided" join can be permitted because the combined fragment will still be a subgraph of the MST. However, since fragments operate asynchronously (and edge testing, MOE selection and joining are neither instantaneous nor atomic), "one-sided" joins create the risk of forming a cycle as one fragment tries to label its MOE as a Branch while the adjacent fragment tries to label another edge as a Branch, and so on.

Rather than reducing parallelism by delaying each join until it becomes "two-sided" (a situation we refer to as an **equi-join**), they permit a fragment F at level L to do a "one-sided" join along its MOE as long as the level of the adjacent fragment is *greater than* L (a situation we refer to as a **submission**). All fragment levels are initialized to zero, and thereafter at each join the higher level replaces the lower one in a submission while both sides increase their level by one for an equi-join. Thus, the level can reach at most $\log(N)$ when the algorithm terminates.

Another noteworthy feature involves reducing the required number of messages by having a node *delay its response* to any test message arriving from a fragment with a higher level than its own, since that other fragment will not be allowed to initiate a join with our fragment until our level increases.

It can be shown (e.g., [GHS83]) that the message complexity of this algorithm is $O(E + N \cdot \log(N))$, and hence optimal. However, its time complexity is $O(N \cdot \log(N))$ and hence not optimal.

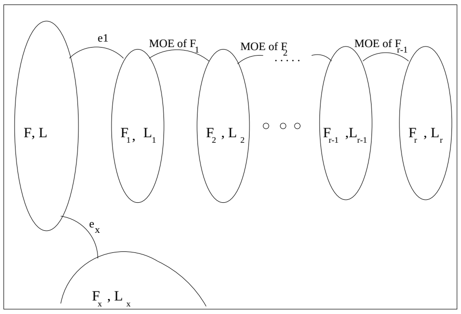

Figure 1: Bad case example.

We now present a pathological test case that we refer to as the **bad case example**, which for this basic algorithm (and all the others based on it) exhibit their worst performance. There is a chain of fragments $F_1, ..., F_r$ (as shown in Fig.1) with respective levels $L_1 \leq L_2 \leq ... \leq L_r$, and $r >> 1$. Furthermore, assume that these fragments are connected with their MOE edges in such a way that F_1 submits to F_2, F_2 submits to F_3 and so on without the previous (in line) fragment participating in the Finding procedure of the following one. We can assume that F_r is of greater level and absorbs all these fragments, or that it equi-joins with F_{r-1}. We can notice that it will take a long time for an *initiate* message from F_r to arrive at F_1. (In particular, we will see in the complexity section that the MOE reporting procedure for the whole fragment will take a long time to complete, in comparison to the level of the fragment.)

Now suppose that an outgoing edge of a fragment (F, L) connects to the fragment (F_1, L_1) such that $L > L_1$. Because of this level difference, F_1 delays its answer to the *test* message from F. Thus F waits for a "very long" time (i.e., until all nodes of the chain of fragments are traversed) before it can complete its Finding procedure. If F joined finally with the chain, then it would get a great level increase and it would be compensated for the delay. However, this is not guaranteed, since F could end up joining with some other fragment F_x and get a very small level increase.

1.2 Awerbuch's Optimal Algorithm

In [Awe87], Awerbuch proposed an innovative three-phase distributed MST algorithm, which achieves optimal performance in terms of both message and time complexity. The different phases represent a tradeoff between the demands of the initial part of the problem (involving large numbers of small fragments, where limiting the number of messages is most critical) and the final part of the problem (involving small numbers of large fragments, where limiting the execution time is most critical).

In the first **Counting** phase, Awerbuch's algorithm determines the total number of nodes, N, after building an unweighted spanning tree using a simple $O(E + N \cdot \log(N))$ message and $O(N)$ time algorithm. The second phase starts to build the MST by following the basic [GHS83] algorithm, described above. However, as the fragments grow larger, they switch to a more complicated phase III algorithm as soon as their sizes reach $\frac{N}{\log(N)}$. The estimation of the size is done trivially in the reporting procedure; all nodes that report are counted.

Awerbuch's third phase differs from the basic [GHS83] algorithm through the addition of two new procedures designed to limit the time between level increases. Recalling the bad case example, above, both procedures aim to increase the level in a long chain, with one of them working outwards from the root of a large but low level fragment, and the other working inwards towards the root from a distance subfragment.

First, the **Root Distance** procedure ensures that the root of a level L fragment will never be blocked for longer than $O(2^{L+1})$, waiting for local MOE responses from all nodes in its fragment. The Root Distance procedure solves the problem by adding a hop counter initialized to 2^{L+1} to the outgoing *initiate* message, which is decremented by one at each hop. Should it ever reach zero, we say that the message has *expired* and require that node to send an *expInit* message to the root which restarts the MOE search from the next level.

The second new procedure in phase three of Awerbuch's algorithm is the **Leader Distance** procedure.[2] This procedure limits the time that a fragment that has submitted to its neighbor (and hence is no longer independent) can be blocked at its old level if the root of the other fragment is very far away (recall the bad case example). In particular, the Leader Distance procedure allows a submitted fragment currently at level L to increase its level within time $O(2^{L+1})$, no matter how far it is from the new root. Similar to the Root Distance procedure, a *testDistance* message, containing a hop count initialized to 2^{L+1} is sent towards the new root. Each node along the path decrements the hop count.

[2] Awerbuch used the name **Test Distance**, but we feel that our name makes the description of the revised algorithm clearer.

If the count hits zero before the message reaches the new root, an acknowledgment is sent back to the fragment, triggering a level increase. Otherwise, the *testDistance* message is discarded when it reaches the new root and the procedure stops. The symmetry of the two procedures is apparent.

We can now go back to the bad case example (Fig.1) and see that F_1 will be updated using the Leader Distance procedure, and will reach L "faster", and thus F will have an answer to its *test* message sooner and thus avoid getting stuck at an edge that may not be its MOE, say e_1 in Fig.1.

1.3 Some Flaws in Awerbuch's Algorithm

In studying Awerbuch's algorithm, we found several issues where the descriptions were either incomplete or incorrectly specified. Consequently, there are some cases where this algorithm, as it is described in [Awe87], can fail by creating a cycle, or at least fail to achieve optimal time complexity.

First, in the proof of optimal time complexity, it is assumed that after each minimum-level fragment has found its MOE, they can submit or equi-join in constant time. Since [Awe87] does not specify a new policy, we must assume that the algorithm has inherited the fragment joining policy from the basic [GHS83] algorithm. Unfortunately, the joining policy in [GHS83] does not satisfy this assumption, even if the fragment(s) are of minimum level. As a counterexample, observe that the last fragment in a long chain of minimum level fragments, each submitting to its neighbor, could wait an arbitrarily long time before receiving an answer to its *connect* message.

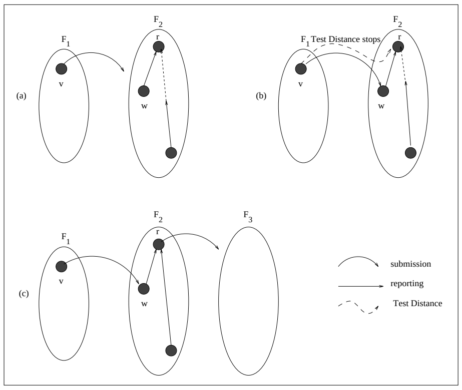

Figure 2: Leader Distance failure

A second issue related to time complexity involves a boundary case where the Leader Distance procedure fails. Recall that after the fragment F_1 has submitted to a node, w, from the adjacent fragment F_2, F_1 uses Leader Distance to force timely level increases while it is waiting for an *initiate* message from the new root. But suppose that node w has already reported its local MOE to the root, r, of F_2 before the F_1 submits, so F_1 will not simply be absorbed into the ongoing MOE search in F_2. Furthermore, suppose w happens to be close to r, so the *testDistance* message from F_1 reaches r with a positive counter, before F_2 has completed its reporting procedure. In this case, r discards the *testDistance* message, which terminates the Leader Distance procedure for F_1. Consequently, if r later decides to submit to fragment F_3, F_3 submits to F_4, and so on, the distance from F_1 to the new root may be huge – leaving F_1 without a level increase for a very long time. Since Awerbuch's proof of time optimality requires level increases to occur on schedule, the Leader Distance procedure obviously needs some refinement.

The third problem is a correctness issue. The level increases for blocked fragments obtained through the Leader Distance procedure can sometimes cause Awerbuch's algorithm to fail. Consider the situation that results from executing the following sequence of four steps (see Fig.??).

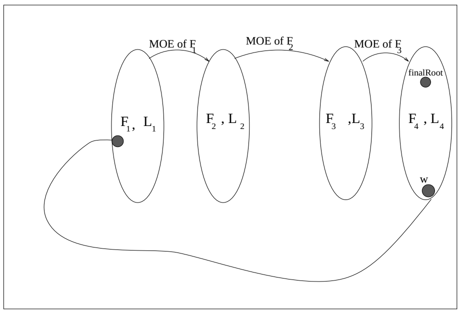

Figure 3: The creation of a cycle.

2 Revised Algorithm

In this section, we present a revised version of Awerbuch's three-phase distributed MST algorithm, which achieves the optimal bounds for both communication and time complexity. The algorithm incorporates several changes in order to eliminate the three problems we identified in the previous section. We have also tried to clarify certain fine details that were not obvious from reading [Awe87] but were needed for an implementation of the algorithm.

2.1 Node Counting Phase

Any optimal Spanning Tree algorithm can be used. In the original paper [Awe87], an $O(E + N \log N)$ messages and $O(N)$ time algorithm is proposed. The purpose of this phase is to determine N so that a fragment size threshold of $\frac{N}{\log(N)}$ can be used to trigger the switch from phase II to phase III. A Spanning Tree (with no minimum weight requirement) is formed by ignoring the edge weights and allowing each fragment to join along the edge leading to the largest fragment. Therefore, we achieve fast level increases and the communication and time complexity of this phase are $O(E + N \cdot \log(N))$ and $O(N)$ respectively (see [Awe87] for details). Given the Spanning Tree, the number of nodes in the network can easily be counted.

In [FM95], a simple counting algorithm is presented. Its complexity is $O(E)$ messages and $O(N)$ time. It assumes that one node can be authorized to initiate the algorithm.

2.2 Small Fragment MST Phase

This second phase is unchanged from [Awe87]. It begins with each node acting as the root of a trivial one-node MST fragment executing the basic [GHS83] algorithm. Each fragment switches to the more complicated phase III algorithm as soon as its size reaches $\frac{N}{\log(N)}$.

2.3 Large Fragment MST Phase

This final phase keeps the same general structure as described in [Awe87], i.e., it is the basic algorithm with the addition of the **Root Distance** and **Leader Distance** procedures. However, there are numerous detail changes to address the issues we identified above. Thus, we will give a detailed walk through of phase III, highlighting the new features in the revised algorithm.

Root Initiates MOE Finding. The first task for a node after it becomes the root of a fragment, with i.d. F and level L, is to initiate the search for the fragment MOE. The root broadcasts an $initiate(F, L)$ message with hop count 0 to all of its children, which increment the hop count and broadcast copies to each of their children, and so on.

If the hop count exceeds 2^{L+1}, we say that the Root Distance procedure succeeds, which stops the broadcast and sends an $expInit(L)$ message back to the root. Each node forwards the first $expInit(L)$ message (if any) it sees back to the root, where it triggers a level increase to $L+1$ and restarts the MOE finding procedure. Additional $expInit(L)$ messages are ignored.

Once the *initiate* messages have been distributed, each node enters the **Testing** state and goes on to the next step. If any smaller-level neighboring fragments try to submit before the node leaves the Testing state, it accepts the submission and then sends a copy of the *initiate* message.

Edge Testing. The Testing policy is the same as in [GHS83]. Nodes query their Unvisited edges, one at a time in increasing order, by sending a $test(F, L)$ message. Edges connecting nodes belonging to the same fragment are rejected using two messages, either a *test* answered by *reject* or two *test* messages sent concurrently. The only other response is *accept*, indicating that the edge is indeed outgoing, and the level of the neighboring fragment is at least L. Thus, if the neighboring node belongs to a smaller level fragment, the Testing policy demands that the answer is delayed until its level increases.

It is interesting to examine the case where node w of fragment (F_w, L_w) tests node v of (F_v, L_v) assuming $L_w > L_v$, so a delayed answer is required. The interesting case occurs if the fragment F_v later decides to submit to F_w. Since node w is still in the Testing state (recall that the edge testing procedure at w is blocked, waiting for the reply from v), w will accept the submission and send a copy of the latest *initiate* message to v. Thus, since node v reaches level L_w with the same fragment i.d. as node w, its (delayed) answer to the original *test* message from node w will be a *reject*, and node w will go on to test its next Unlabeled edge. We see that this delay in answer is crucial for the correctness of the algorithm.

MOE Reporting. While a node is searching for its local MOE, it is also gathering *report* messages from each of its children, which identify the best MOE found in their respective subtrees. Once the local MOE has been determined, and all children (if any) have reported, the node identifies the best MOE among all these choices (and remembers the path that leads there), and sends the result to its parent in another *report* message.

Leader Selection. When all the *report* messages have reached the root, the fragment MOE is selected as the best choice reported by any of node. If none of the nodes reported an MOE, then the algorithm terminates, having built the entire MST. Otherwise, the root directs a *changeRoot* message to the node adjacent to the fragment MOE, appointing it as the leader. The *changeRoot* message reverses the parent relations of the nodes along the path between the old root and the new leader, re-orienting the tree hierarchy towards the new leader.

NEW – After forwarding the *changeRoot* message, each node on the path (including the old root and new leader) broadcasts an *MOEfound* message to all remaining nodes, to inform everyone that the search for the fragment MOE is over and that the tree has been re-oriented towards the new leader. We will say that a node is in the **Decided** state after it has received either a *changeRoot* or *MOEfound* message and before it receives the next *initiate* message. It is easy to see that the path from any decided node towards the leader (or root) will not change unless a new Finding procedure takes place. The significance of this fact will become apparent below.

Leader Joining Protocol. Consider a node, v, which has become the leader of fragment (F_v, L_v) and wants to join with node w of fragment (F_w, L_w). Node v sends a *connect* message to node w and waits for a reply.

If $L_w > L_v$ then node w will reply immediately, allowing F_v to adopt the new level and even participate in the MOE finding procedure for F_w if node w has not yet reported its MOE.

NEW – Conversely,[3] if $L_w = L_v$ then node w waits for a *changeRoot* or a *MOEfound* message and then proceeds to do an equi-join or accepts the submission of F_w respectively.

The addition of the *MOEfound* message is significant, because it means that the maximum time that node v can be blocked by (the absence of) node w's response to its connect message will be proportional to its own fragment size. As we already said, such a characteristic doesn't exist in the leader join protocol that was described in [GHS83], and seems to have been inherited by [Awe87].

Thus, unlike the joining protocol in [GHS83], ours offers a guarantee that the delay until the leader receives some answer from the adjacent fragment, is proportional to the other fragment's size. It is also very important to notice that with the use of *MOEfound* message, we can guarantee that the path towards the root that the leaders will try to measure (Leader Distance procedure) will not change in length. This will be used also in the proof of correctness.

Leader Distance Testing. Now we will see how the Leader Distance procedure works. The basic idea is that a leader sends a *testDistance* message towards the root and if that message visits enough nodes, the leader is allowed to increase its level accordingly.

In more detail, a leader submits and waits for an answer (we will call such a leader **Blocked**). In a submission and after the other node becomes Decided, the leader invokes the Leader Distance procedure. The *testDistance* message carries the level of the leader, say L_{msg}, (we will call it level of the message) and has a counter which is initially set to 0 and measures the distance. Each node increases the counter by one before forwarding the message to its parent.

NEW – There are two ways the procedure can increase the level of the leader that started it: a) the message *expires* when the counter reaches $2^{L_{msg}+2}$ (this hop count trigger is twice as large as in [Awe87] and will be explained below); or b) the message arrives at a node whose level is greater than L_{msg} (the second condition is entirely NEW). In both cases, the node where the test succeeds sends an acknowledgment

[3] Recall that the case $L_w < L_v$ can't happen at this stage, since node w would have been required to delay its reply to v's test message.

message is sent back to the leader informing it of the level it can adopt. This new level can be: the old level increased by one If the $testDistance$ message expired, the new level will be one greater than the old one. However, larger increases are possible if it was triggered by a higher level node. In addition, a leader that receives a $testDistance$ message of greater level than its own, will increase its level before forwarding the message (which is also NEW).

The leader broadcasts its level increase to the nodes of its fragment (by broadcasting a $MOEfound$ message) and restarts the procedure at its new level. When a $testDistance$ message "crosses" with an *initiate* message at an edge it is discarded (we say that it dies) and the procedure stops.

Avoiding Leader Distance failure. (NEW) – We can avoid the failure of the procedure by making sure that when a $testDistance$ message arrives at a Blocked leader, the message is "kept in memory" and forwarded towards the new root, when the Leader Distance procedure is invoked in that leader. Notice that Blocked leaders can't actually store the incoming $testDistance$ messages, since the number of such messages could be equal to the number of distinct fragments which is $O(\log(N))$ for phase III. Therefore, it is necessary and more efficient to store only the $testDistance$ message with the greatest counter. This "concentration" of the messages into one requires trivial memory and reduces the communication complexity.

Notice that because of the NEW features of the Leader Distance procedure, a leader and all the messages it must store are of the same level; smaller level messages that may arrive succeed immediately (no need to store them), while greater level messages increase the level of the receiving leader (which also informs the other low level leaders as well). Thus the selection of the greater distance can only speed up the level increase of some of the leaders.

Avoiding Cycles (NEW) – The factor of 2 that we saw when checking the distance counter of the $testDistance$ message, is a handicap that we apply to the Leader Distance procedure compared to the Root Distance procedure. In brief, in order to trigger an increase to the same level, Leader Distance must detect twice the distance compared to Root Distance. This way, it is guaranteed that whenever Leader Distance can trigger a level increase from L to $L+1$, Root Distance will trigger a level increase from $L+1$ to $L+2$

We will discuss this in more detail in the correctness section.

3 Correctness

3.1 General Results

In order to prove that the algorithm is correct, we have to prove that it terminates and finds the MST.

For termination, the following theorem holds.

Theorem 1 *The revised algorithm is deadlock free.*

A proof can be found in [GHS83].

To prove that the algorithm finds the minimum of the spanning trees we can recall the fact that the MST problem can be solved by a "greedy" algorithm. In other words it is sufficient to verify that the algorithm makes fragments to try and join only along their MOE. The previous description must have left no doubt that the edge along which a *connect* message is sent, is indeed of minimum weight for all the nodes that received the *initiate* message and participated in the Finding procedure.

In addition, we must prove that it actually finds a tree, i.e., that it does not create a cycle. The following theorem holds.

Theorem 2 *The revised algorithm does not create cycles.*

PROOF. Initially, let us ignore the Leader Distance procedure. We can see that decision making is centralized within a fragment, so it is not possible to have a cycle, i.e., a fragment decides to join to one fragment at a time and then participates in a new Finding procedure and reexamine their previous outgoing edges. This centralization, the use of levels in the joining policy and the distinct weights of the edges guarantee the avoidance of cycles. Schematically, we can say that joining decisions and level increases take place at the root and thus errors are avoided.

Let us consider the Leader Distance procedure. and recall the example that made [Awe87] create a cycle.

As already discussed, our modified algorithm breaks the symmetry of the two procedures. We demand that the Leader Distance increases the level of a submitted fragment to $L+1$ only when it detects a distance to the root to be at least $2 \cdot 2^L = 2^{L+2}$. This way it will be guaranteed that if the Leader Distance procedure succeeded and made it to level $L+1$ then Root Distance will ultimately increase the level of the final fragment to $L+2$. This applies for all the levels and thus we guarantee that eventually **the final fragment**, fragment F_4 in our example, **will submit to a fragment of greater level than the level of any of its subfragments ever was.•**

Notice here that the modified Joining policy guarantees also that the two procedures explore the same path. As we said, nodes start forwarding $testDistance$ messages after they are decided guaranteeing that the final orientation of the fragment is established (final until the next Finding procedure). In [Awe87], this problem of "path change" is not being discussed and lack of detailed description does not allow us to know whether it was taken under consideration.

4 Complexity

This section will offer a brief description of the complexity issues for the revised algorithm. Recall that communication complexity is defined as the total number of messages exchanged between adjacent nodes, i.e., broadcasting an *initiate* message in a fragment of k nodes, counts as $k-1$ message exchanges. Similarly, time complexity corresponds to the elapsed time required for the termination of the algorithm, assuming the transmission delay at each edge is one time unit. Processing time for each message within a node and queuing delays are considered negligible.

The length of messages is $O(\log(N))$, i.e., capable of representing N node identifiers. It is optimal and it won't be discussed further (see [GHS83]).

We will calculate the complexity of the algorithm by summing the complexities of its three phases.

4.1 Communication Complexity

Phase I: The communication complexity of this phase was proven optimal [Awe87].

Phase II, III: We can see that each edge is rejected only once (if at all) and only two messages (two $test$ messages or a *test* and a *reject* message) are required. Thus, edge rejection uses $O(E)$ messages. Notice that *test* messages not leading to rejection will be counted in the sequel.

For the other messages, we can see that if we partition the messages generated at a single level among the nodes, then the number of messages assigned to a node is bounded by a constant. In more detail, a node can receive at most one *initiate*, one $MOEfound$ and one *accept* message. It can transmit at most one successful *test* message, one *report* message and one *changeRoot* or *connect* message. As we already said, the maximum level is $\log(N)$ and thus the total number of the other messages is $O(N \cdot \log(N))$.

We are now left with the $testDistance$ messages. Notice that the number of acknowledgment messages sent when the Leader Distance procedure succeeds is at most equal to number of $testDistance$ messages. Recall that the Leader Distance procedure is activated after the number of fragments is reduced to $\log(N)$. We can't have more than $\log(N)$ submissions and thus $O(N \cdot \log(N))$ $testDistance$ messages. It is really interesting to notice that if the the Test Distance procedure was active from phase II, it could increase the message complexity above the optimal, i.e., we can think of $N/2$ single nodes submitting to a fragment of $N/2$ size in such a way that causes $O(N^2)$ $testDistance$ messages. This is the reason why the Leader Distance procedure is activated only in phase III.

Therefore, the communication complexity of the algorithm is $O(E + N \cdot \log(N))$ and optimal.

4.2 Time Complexity

It is comparatively easy and considerably less exciting to see that the Counting phase and the first part of the MST phase are optimal with respect to time (see [Awe87]) and we will provide only short explanations.

Phase I: In the Counting phase adjacent fragments join in a way similar to that of [GHS83], but since it is trying to find just a Spanning Tree it is relatively easy to guarantee that fragments almost never wait for other trees (for a detailed proof see [Awe87]).

Phase II: In the first part of the MST phase, the size of the fragments has at most $\frac{N}{\log(N)}$ nodes and the maximum level is bounded by $\log(N) - \log(\log(N)) \leq \log(N)$. We can assume that for each of these subfragments, we run independently a [GHS83] algorithm for a graph of size $\frac{N}{\log(N)}$ and thus the termination time, which is the product of the maximum level and the size of the graph, is $O(N)$.

Phase III: we will provide an overview of the proof.

Theorem 3 *For the second part of the MST phase, the length of the time period during which the minimum level is L is bounded by $O(2^L)$ time units.*

PROOF (sketch). Let $T(L)$ denote the first time at which L is the minimum level in the network. We can distinguish two parts. In each one of them, one of the new procedures guarantees that the level is increased "on time". Note that considering $Size(F) = O(2^L)$ can facilitate understanding.

Part 1. Root Distance Procedure : Consider the minimum level fragments after $T(L)$. In some of them, an *initiate* message may visit 2^{L+1} nodes corresponding to 2^{L+1} time units and the Root Distance procedure will increase their level in $O(2^L)$ time.

In the rest of the fragments, after $c_1 \cdot 2^L$ time where c_1 is some constant, they will have found their MOE and all the nodes have been informed (they have received a *changeRoot* or a $MOEfound$ message). Since the fragments are of minimum level, they will be able to join in constant time (because, by now, the node answering their test message must either be in the Decided state or belong to a higher level fragment) and either increase their level or invoke the Leader Distance procedure. (Note that this claim for immediate joining doesn't hold the way things are described in [Awe 85]) Therefore, $c_1 \cdot 2^L$ time units after $T(L)$, all minimum level fragments will have increased their level or have submitted, and nodes of L level are either leaders that have invoked the Leader Distance procedure or Decided nodes. In the next part we follow the level behavior of such leader.

Part 2. Leader Distance: if a fragment F has submitted, and received an *initiate* message from the greater fragment it is an acknowledged part of the latter and the previous phase takes care of its level upgrades.

If F is far away from the root that will send an *initiate* message, the Leader Distance procedure guarantees that a level increase will take place in $c_2 \cdot 2^L$ where c_2 is a constant. It is easy to see that in the worst case a $testDistance$ will expire after $2 \cdot 2^{L+1}$ time units, and the acknowledgment message will return to the leader in the same time. Notice that after $T(L) + c_1 \cdot 2^L$, all L level leaders are not Blocked and there is no delay in the propagation of the message.

Notice that in both phases, the time required for the level increase to propagate to all nodes of the fragment is bounded by $c \cdot 2^L$ time units where $c = c_1 + c_2$. Consequently, after time $O(2^L)$ all nodes will have level greater than L.•

It is not difficult now to prove that the time complexity of phase III is optimal, by adding the time intervals during which each level is the minimum level.

$$Time = \sum_{i=\frac{N}{\log(N)}}^{\log(N)} c \cdot 2^i \leq 2 \cdot c \cdot 2^{\log(N)} = 2 \cdot c \cdot N$$

Finally, we can see the logical steps that the creation of the algorithm seems to follow. In order to achieve $O(N)$ time, the Leader Distance procedure was needed. This increased the communication cost and in order to keep it optimal we had to demand fewer than $\log(N)$ submissions of fragments entering phase III (enabling Leader Distance). For this, fragment should switch to phase III after obtaining size $\frac{N}{\log(N)}$. Therefore, each fragment should be aware of N, and the Counting phase (I) is necessary.

5 Epilogue

In this paper, we identified some problems with Awerbuch's distributed MST algorithm [Awe87], involving both correctness and optimality issues. We then gave a revised algorithm, which introduces several new features to solve these problems. We also show that with these changes, our revised algorithm satisfies the desired correctness and optimality conditions.

This work arose from our pragmatic efforts to find a good MST algorithm for real applications reported in [Fal95]. In that work, apart from discussing theoretical issues, we tested the performance of several distributed MST algorithms after constructing detailed implementations of each one in a simulated communication network environment. Our results indicate that there is still a lot of room for improvement in this problem domain.

Acknowledgments – The authors would like to thank Baruch Awerbuch for his useful comments and encouragement concerning the modifications and Vassos Hadzilacos for his valuable advice. We would also like to thank Petros Faloutsos for his keen comments and support.

References

[Awe87] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. *Proc. 19th Symp. on Theory of Computing*, pages 230–240, May 1987.

[CT85] F. Chin and H.F. Ting. An almost linear time and $o(vlogv + e)$ messages distributed algorithm for minimum weight spanning trees. *Proceedings of Foundations Of Computer Science (FOCS) Conference Portland, Oregon*, October 1985.

[Fal95] Michalis Faloutsos. Corrections, improvements, simulations and optimstic algorithms for the distributed minimum spanning tree problem. *Technical Report CSRI-316*, 1995.

[FM95] Michalis Faloutsos and Mart Molle. Creating optimal distributed algorithms for minimum spanning trees. *Technical Report CSRI-327 (also submitted in WDAG '95)*, 1995.

[Gaf85] Eli Gafni. Improvements in the time complexity of two message-optimal election algorithms. *Proceedings of 1985 Principles Of Distributed Computing (PODC)*, Conference, Minacki, Ontario, August, 1985.

[GHS83] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5(1):66–77, January 1983.

[GKP93] J.A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *Proceedings of Foundations Of Computer Science (FOCS)*, 34, 1993.

[SB95] Gurdip Singh and Arthur J. Bernstein. A highly asynchronous minimum spanning tree protocol. *to appear in Distributed Computing*, Spinger Verlag 8(3), 1995.