Distrib. Comput. (2004) 17: 151–170
Digital Object Identifier (DOI) 10.1007/s00446-004-0107-2

DISTRIBUTED COMPUTING

A linear-time optimal-message distributed algorithm for minimum spanning trees

Michalis Faloutsos, Mart Molle*

University of California Riverside, Department of Computer Science, Riverside, CA 92521, USA (e-mail: {mart,michalis}@cs.ucr.edu)

Received: February 9, 2003 / Accepted: April 2, 2004
Published online: 20 July 2004 –

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 a connected undirected graph G, with N nodes and E weighted edges, 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 none of the nodes has any special[1] status, nor are any of them aware of the network topology except for their adjacent edges. A centralized solution where all information is collected in one node would require at least a symmetry-breaking procedure to select the central node, and also lead to excessive communication overhead. 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. We assume an asynchronous[2] network, and consider messages of size $O(\log N)$ bits since the set of unique identifiers for N nodes requires $\Omega(\log N)$ bits.

The problem becomes more interesting when we attempt to minimize the number of messages and the execution time. For the message complexity bound, we count the transmission of one message across one edge as our unit of "cost". For the time complexity, we count the transmission of a packet as one unit of time. For the general graph, the distributed MST problem requires at least $\Omega(E + N \cdot \log(N))$ messages, and a time complexity bounded below by $\Omega(N)$ [7]. Intuitively, the factor $\Omega(E)$ corresponds to traversing all the edges at least once, while the factor $\Omega(N \cdot \log(N))$ corresponds to the "negotiations" among the nodes in deciding the MST. Regarding the time complexity lower bound, we obtain $\Omega(N)$ by assuming the worst-case of parallelism, where $O(N)$ messages must be exchanged sequentially to ensure network agreement. For example, think of a line of nodes. Therefore, we call optimal an algorithm that requires $O(E + N \cdot \log(N))$ messages and $O(N)$ time.

In [1], Awerbuch proposed an innovative three-phase distributed MST algorithm, which achieves optimal performance in terms of both message and time complexity. We refer to this algorithm as Awerbuch's algorithm or **AWE**. 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).

The contribution of this paper is two fold. First, we show that, in some cases, the AWE algorithm can create a cycle or fail to achieve optimal time complexity. Second, we present a modified version of Awerbuch's algorithm, that avoids the aforementioned problems. We call this algorithm the **Modified Awerbuch's algorithm (MA)**. We also clarify some subtle but critical functions of the algorithm, which for AWE were left unspecified in [1]. We then prove that the modified algorithm avoids these problems, and demonstrate its correctness and its optimality. This paper is based on our earlier work in this direction [4], but it has been significantly expanded to include more non-trivial details, more substantial proofs, and a pseudocode version of the algorithm in the Appendix.

* This material is based upon work supported by the National Science Foundation under CAREER Grant No. 9985195, and Nortel Networks and UC CoRe fund C99-14.

[1] For simplicity, we can consider that all nodes of the graph are "awake", but even if only one is, it can wake up the others through the execution of the algorithm.

[2] The algorithms that solve the problem in a synchronous setting would require synchronizers in an asynchronous network and that would make them suboptimal: the synchronizers introduce a polylogarithmic overhead.

Due to the nature of the work, the rest of this paper has been structured as follows. In Sect. 2, we present the seminal GHS algorithm [7] to establish a basic understanding of the problem. In Sect. 3, we present the original AWE algorithm in sufficient detail to allow us, in Sect. 4, to explain why AWE fails to perform as expected. In Sect. 5, we describe our MA algorithm in detail and explain how these modification solve the problems we found with AWE. In Sect. 6, we show that the message complexity of MA is optimal. In Sect. 7, we show that MA terminates in linear time. In Sect. 8, we prove the correctness of MA.

2 The GHS algorithm

In their pioneering paper [7], Gallager, Humblet and Spira introduced the distributed MST problem and established a number of basic assumptions that have been widely adopted in subsequent work. In particular, they assume that the graph is connected and all nodes have distinct identifiers. They also assume that the edge weights are distinct. Moreover, if the weights are not distinct they show how to use the node id's to break the ties in a deterministic way, and hence to avoid cycles [7].

More importantly, this first work [7] introduces an algorithm that we call **GHS**, which has formed the foundation for further work in the area, for example [1–3,6]. It can be shown [7] that the message complexity of GHS is $O(E + N \cdot \log(N))$, and hence optimal. However, it is well known that its time complexity is $O(N \cdot \log(N))$, and hence not optimal.

Following a slightly different angle, other research expresses the time complexity as a function of the diameter of the MST, d, the diameter of the network, δ, or the maximum node degree, D. Singh and Bernstein prove [11] that the time complexity of the GHS algorithm is bounded by $O((D + d) \cdot \log(N))$, which reduces to $O(N \cdot \log(N))$ when $d = O(N)$). Garay et al. deal with the problem of minimizing the time at the expense of the message complexity [8]. Their algorithm relies on a synchronized network, and speeds up the convergence of the decision of the nodes. Kutten and Peleg proposed a brilliant improved version of the previous algorithm [9]. The time complexity of their algorithm is $O(\sqrt{N} \cdot \log^*(N) + \delta)$, which is a linear function of the network diameter. Note that both algorithms have non-optimal message complexity, and their worst-case time complexity reduces to $O(N)$ for the worst case of network diameter, where $\delta = O(N)$.

2.1 Overview of GHS

GHS constructs the MST by allowing **fragments** (i.e., a connected component of the MST) to merge in an asynchronous distributed fashion. Throughout this process, each fragment maintains its own spanning tree, and fragments grow in such a way that the spanning tree of a fragment will become a part of the MST of the graph. Each node is initially the **root** of its own trivial single-node fragment, and all the edges are **Basic**. Thereafter, adjacent fragments join to form larger fragments by labeling their minimum weight intermediate edge as a **Branch** of the MST. All other edges are labeled **Rejected**, when it becomes clear that they can never become Branches. We will use the terms **Upstream** and **Downstream** to denote the directions toward and away from the root of the fragment, respectively, for each Branch edge.

The new Branch is chosen by the root of one (or possibly both) of the fragments, as the **minimum outgoing edge** (or **MOE**) for its entire fragment. Since fragment merging is not an atomic operation, and every tree can have only one root, we use the term **leader** to denote the "central" node for a fragment while it is merging with an adjacent fragment. All "non-central" nodes belonging to the fragment are called **ordinary**.

To find the fragment MOE efficiently, GHS uses a sequence of three procedures. First, the root initiates the **Searching procedure**, by broadcasting a request to all nodes to help find the fragment MOE. Then all nodes execute the **Testing procedure** to identify their respective local MOEs. Finally, all nodes execute the **Reporting procedure** to collect the results from all Downstream nodes in their entire subtree before passing on a single aggregate response toward the root.

Even this basic algorithm contains several subtleties, which are described in [7]. 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 both of them wish to label that edge as a Branch, then it is clear that the resulting "two-sided" merger (a situation we refer to as an **equi-join**) can be permitted, because the combined fragment will still be a subgraph of the MST. In an equi-join, one of the two adjacent leaders will be the root of the new fragment through some simple tie-breaking mechanism, such as the node with lowest id becomes the root.

Rather than reducing parallelism by delaying each join until it becomes "two-sided", GHS permits 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**. We say that a fragment **submits** to another fragment when it sends a message declaring that it wants to join the other fragment along that edge. Thus, an outgoing edge leading to a lower-level fragment is not an acceptable candidate for the fragment MOE, but it cannot be ignored either! This delays the Testing procedure, and contributes to non-optimal time performance, as we will see below.

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.

In the remainder of this section we provide a detailed description of the GHS algorithm. Following the structure of the algorithm, our presentation is divided into two parts: a) finding the fragment MOE, and b) merging of fragments. A summary of all message types used by GHS and subsequent algorithms is provided in Table 1.

Table 1. The messages of the algorithms

	Messages common to GHS, AWE and MA
initiate	Broadcast message that signals a new searching phase for a fragment.
report	The reply to the $initiate$ message, it reports the Minimum Outgoing Edge of the subtree.
test	The message explores whether an edge is going outside the fragment.
reject	A negative answer to the $test$ message (internal edge).
accept	A positive answer to the $test$ message (external edge).
changeRoot	The order to authorize the leader of its role.
connect	The intention to join along a particular edge (MOE of the fragment), sent by the leader.
	New messages introduced by AWE, and common to MA
expInit	Feedback to the root, when $initiate$ travels too far and expires.
testDistance	Probe that explores the distance from the leader to the new root.
ack	Feedback to the leader, when $testDistance$ expires.
	New messages introduced by MA
MOEfound	Notification that the MOE of the fragment has been found.
acceptSub	Response to a connect message, when a submission is accepted.

2.2 Part I: Finding the fragment MOE

During the execution of the GHS algorithm, an ordinary node alternates between a *passive* **Found state**, where it waits to receive further instructions from the root, and an *active* **Finding state**, where it executes the Searching, Testing and Reporting procedures. The rules governing each procedure are described as follows:

GHS-0 An ordinary node remains in the **Found state** until the start of the next active period. It is aware that the global MST-finding algorithm is currently in progress, and remembers: (i) its own most-recently assigned fragment id, F; (ii) its level, L; (iii) the status of each of its attached edges (i.e., Basic, Branch, or Rejected); (iv) an Upstream pointer that indicates which Branch leads to the root; and (v) a Best Edge pointer that indicates the next hop toward the most-recently reported local MOE in its own subtree (i.e., either one of its Downstream Branches or its own most-recently tested Basic edge). A node that receives an $initiate()$ message will switch to the Finding state.

GHS-1 The initial state for the root node is the **Finding state**, from which it initiates a Searching procedure for its entire fragment by broadcasting an $initiate(F, L)$ message over all its Branch edges. All ordinary nodes in F enter the Finding state and execute the Searching procedure when they receive a copy of the $initiate(F, L)$ message from their Upstream Branch edge. To carry out the Searching procedure, each node takes note of its current fragment id, F, and level, L, and then relays a copy of the $initiate(F, L)$ message to each of its Downstream Branch edges. Thereafter, the nodes immediately execute the Testing procedure.

GHS-2 In the Finding state, each node executes the **Testing procedure** to determine its local MOE. Nodes query their Basic edges, one at a time in order of increasing weight, by sending a $test(F, L)$ message. Edges connecting nodes that belong to the same fragment are **Rejected** either by a $reject$ message or by receiving a concurrent $test$ message with the same fragment id. The only other allowable response is $accept$, indicating that the edge is indeed outgoing, and the level of the neighboring fragment is at least L. Note that if the neighboring node belongs to a smaller level fragment, the Testing policy demands that the answer is delayed until its level increases. Once the node has found its first outgoing edge (or exhausted its Basic edges), it executes the Reporting procedure.
It is interesting to examine the case where node u of fragment (F_u, L_u) tests node w of (F_w, L_w) assuming $L_u > L_w$, so a delayed answer is required. In this case, even though node w cannot respond to the $test(F_u, L_u)$ message from node u, it may decide to send its own $test(F_w, L_w)$ message over the same edge. Clearly, node u must reply immediately with an $accept$ message, which terminates the Testing procedure for node w. Thus, if this local MOE for node w becomes the fragment MOE for F_w, then F_w would submit to F_u over this same edge. Note that node w continues to delay its response to the earlier $test$ message from node u, and thus node u still "blocked" in the Testing state. Thus, u will accept the submission of F_w and send a copy of the latest $initiate$ message to w, which (finally!) raises its level to L_u. At this point, node w can respond to the original $test$ message from u, but now w is in the same fragment as u. A $reject$ message is sent to node u, and node u will go on to test its next Basic edge. This delay in answer is crucial for the correctness of the algorithm, namely, for avoiding cycles.

GHS-3 In the Finding state, each node executes the **Reporting procedure** to determine the best outgoing edge located anywhere in its own subtree. The Reporting

procedure is invoked when the local MOE is found, or whenever a $report$ message arrives from a child. Finally, a node finds the minimum weight outgoing edge (if any) among the local MOE and the incoming $report$ messages from each of Downstream Branches. The Reporting procedure terminates when the node has: (i) identified the best MOE among all these choices; (ii) stored the Best Edge pointer that leads to that choice; and (iii) sent the resulting MOE weight to its parent (unless it is the root) in another $report$ message. If no such edge exists, the Reporting procedure reports null. In any case, when the Reporting procedure concludes, the node switches to the Found state.

2.3 Part II: fragment-merging along the MOE

Once the root node has completed its Reporting procedure, it terminates the MST algorithm if no outgoing edges were reported. Otherwise, it initiates the task of joining with its closest neighboring fragment via the following sequence of steps and procedures:

GHS-4 When the root enters the Found state after completing the Reporting procedure, it immediately executes the **ChangeRoot** procedure. All other nodes remain in the Found state unless they receive a $changeRoot$ message from the root, since only one node at a time can execute the ChangeRoot procedure.
The ChangeRoot procedure causes the node to delete its Upstream pointer, which effectively locates it as the root of the fragment (at least temporarily). Thereafter, the node retrieves the Best Edge pointer, which it stored during the Reporting procedure. If the Best Edge points to one of its children, say node c, then the node sets its Upstream pointer to the Branch leading to node c, sends a $changeRoot$ message to node c and (re-)enters the Found state. If the Best Edge points to one of its own Basic edges, then the node concludes that it must be the leader (which is adjacent to the fragment MOE and responsible for carrying out the merger) and immediately invokes the Merging procedure.

Observation 1 *All nodes along the path connecting the root to the* **leader** *node execute the ChangeRoot procedure in sequence, resulting in a gradual reorientation of the Upstream pointers toward the leader.*

GHS-5 In the **Merging** procedure, the leader node v sends a $connect(F_v, L_v)$ message across the MOE to notify the adjacent node w, which is in fragment (F_w, L_w), of its intent to merge.

Observation 2 *At this point, we know that $L_w \geq L_v$. During the previous Testing procedure by node v, node w could not have replied to node v's test message if its level had been below L_v. Thereafter L_v cannot change during a single iteration of the MOE-finding procedure for F_v. However, node w's level could have increased because F_w executes its local copy of the MST-finding algorithm asynchronously, and hence may have advanced to the next iteration.*

If $L_w > L_v$, then node w accepts the merger as a *submission by a lower level fragment*, so it replies immediately by sending an $initiate(F_w, L_w)$ message to node v to inform it of the new fragment id and level. Note that it does not matter whether the information given to node v is current (i.e., node w is actively involved in finding its own fragment MOE) or obsolete (i.e., node w has already completed the Reporting procedure associated with that fragment id and level). In the former case, the nodes in fragment F_v will simply be included in the current iteration of Part I for fragment F_w. In the latter case, the nodes in fragment F_v will again carry out Part I for level L_w, but the results will be discarded by node w — since any result they find must be heavier than edge (v, w), which was the previous MOE for F_v, whereas the edge already reported by node w must be lighter than (v, w) or else the Testing procedure at w would still be blocked waiting for a reply from node v.
Conversely, if $L_w = L_v$ then node v must remain blocked in the Merging state until node w responds to its *connect* message. There are two cases to consider, based on the level-L_w MOE selected by fragment F_w:

1. If fragment F_w selects the same edge (v, w) as its MOE, then node w will eventually become its leader and complete an equi-join with fragment F_v by sending a $connect(F_w, L_w)$ message to node v. Thereafter, one of the two leaders[3] will become the root of the resulting level-$(L_v + 1)$ fragment through some simple tie-breaking mechanism (e.g. minimum node id).
2. If fragment F_w selects a different edge as its MOE, then node w will not respond to the $connect$ message from node v until it receives a new, higher-level $initiate$ message from its new root and hence can accept the merger as a submission by a lower level fragment.

Observation 3 *If fragment F_w selects an edge other than (v, w) as its MOE, then node w may not respond to node v's* connect *message for a very long (i.e., greater than $\Omega(2^L)$) time. For example, F_w could be the last member of a long chain of level-L_v fragments, where each of them has decided to submit to its neighbor in the chain.*

3 Awerbuch's three-phase MST algorithm

In [1], Awerbuch proposed an innovative three-phase distributed MST algorithm, which we refer to as AWE. AWE achieves optimal performance in terms of both message and time complexity. The different phases represent a tradeoff between the demands of the early part of the problem (involving

[3] Note that equi-joins always take place between leaders of the fragments. Each leader executes the algorithm without knowing if the other node (toward the outgoing edge) is a leader, until it receives a $connect$ message from the other node.

large numbers of small fragments, where limiting the number of messages is most critical) and the later part of the problem (involving small numbers of large fragments, where limiting the execution time is most critical). The switch to the last phase happens when fragments reach a size larger than the **phase III threshold size** which is set to $N^* \equiv \frac{N}{\log(N)}$ nodes.

The three phases are needed to guarantee optimal message complexity, by restricting the availability of some newly-defined message-intensive procedures that speed up its execution compared to GHS. These procedures reduce the amount of time that fragments can be blocked while waiting for a response to a *test* or *connect* message. However, the new procedures also bring with them a significant cost in terms of additional communications. In particular, a single fragment executing the new Leader Distance procedure (described below) may generate $O(N)$ messages in the worst case. Thus, if large numbers of fragments (say $O(N)$ of them) were allowed to execute this procedure, then the total number of messages could reach $O(N^2)$, in contrast the $O(N \cdot \log(N))$ message complexity of GHS. Thus, AWE only allows a fragment to use the new procedures after its size has reached the phase III threshold size, which implies that the total number of active fragments is $\Omega(\log(N))$. Consequently, even in the worst case, where all fragments invoke the Leader Distance procedure and generate $O(N)$ messages, the total number of messages is still $O(E + N\log(N))$, which is optimal.

3.1 Phase I: node counting

The purpose of this phase is to determine N, so that a fragment-size threshold of 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. Given the Spanning Tree, the number of nodes in the network can easily be counted. Any optimal algorithm for building an unweighted spanning tree can be used. In the original paper [1], an $O(E+N\log(N))$ messages and $O(N)$ time algorithm is proposed.

3.2 Phase II: basic small-fragment MST algorithm

During this phase, we start to build the MST using GHS, as described above. The only difference is that at the end of the Reporting procedure, the root estimates the size of its fragment to decide whether or not it is time to switch to the optimized Phase III algorithm. The estimation of the fragment size is done trivially in the Reporting procedure; all nodes that report are counted.

3.3 Phase III: optimized large-fragment MST algorithm

Awerbuch's optimizations add two new procedures to GHS in order to limit the execution time for each iteration of Phase III. The goal for both procedures is to promote an orderly schedule of level increases if multiple fragments merge at the same time to create a much larger fragment. One procedure works outward from the root of the fragment F to force its level to keep pace with the size $\log|F|$. The other procedure works inward toward the root from the leaders of distant sub-fragments that have recently joined F.

The **Root Distance Procedure**, adds a timeout mechanism to the Searching procedure. More specifically, a *hop count* field is added to the *initiate* message. The count is initialized by the root to 2^{L+1} and thereafter it 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 back an *expInit* message to the root. Thus, when the root receives the first *expInit* message for the current level, it increases the fragment level by one and restarts the Searching procedure.

The **Leader Distance**[4] **Procedure** ensures the timely level increase of a fragment after it has submitted and before it becomes an active member of its new fragment. Leader Distance can be seen as a timeout mechanism to the Merging procedure, to limit the time between level increases for the leader that has submitted to an adjacent fragment (GHS-5). During this time, the fragment must continue to use its old fragment id, F, and level, L, even though it has already decided to merge with the adjacent fragment.

Similar to the Root Distance procedure, a $testDistance$ message, containing a hop count initialized to 2^{L+1} is sent toward the new root. Each node along the path decrements the hop count. If the count hits zero before the message reaches the new root, an ack message 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.

4 Some problems in Awerbuch's algorithm

In studying AWE, we found several places where the descriptions were either incomplete or incorrectly specified. Consequently, there are some cases where this algorithm can fail, either by creating a cycle or by failing to achieve optimal time complexity.

4.1 The phase III fragment-merging policy is incomplete

Since AWE does not specify a new fragment-joining policy for its Phase III, we must assume that the policy was inherited from GHS (see GHS-5). Unfortunately, the GHS fragment-joining policy is not consistent with the detailed performance requirements assumed in [1].

What is the execution time for the Merging procedure? Awerbuch's proof of optimal time complexity assumes that if the fragment (F, L) is currently at the minimum level, then F will complete the Merging procedure within $O(2^L)$ time of the discovery of its fragment MOE. In particular, if node v carries out the Merging procedure as the leader of fragment F_v, then v is assumed to know within $O(2^L)$ time whether the resulting merger with the adjacent fragment F_w represents an equi-join or a submission to a larger fragment.

Unfortunately, a careful review of GHS-5 shows that the only allowable responses to the $connect(F_v, L_v)$ message sent by node v to the adjacent node w in fragment F_w are:

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

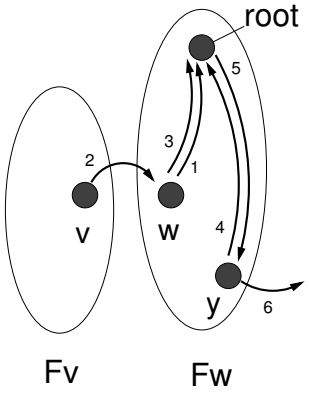

Fig. 1. Leader Distance fails because the path length to the root increases: root of F_w discards arriving *testDistance* message from v, but later submits to a more distant fragment via node y

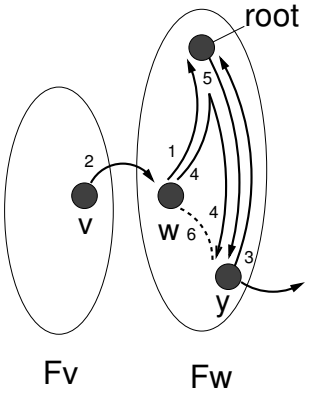

Fig. 2. Leader Distance fails when the path length to the root decreases: *testDistance* message from v reverses direction and counts part of its path twice when it encounters the *changeRoot* on its way to node y

- $initiate(F_w, L_w)$, for some $L_w > L_v$, in which case F_w treats the request from F_v as the submission of a lower-level fragment, or
- $connect(F_w, L_w)$, for $L_w = L_v$, in which case F_w treats the request from F_v as an equi-join.

Moreover, using observations 2 and 3, we see that even if both fragments F_v and F_w were at the minimum level during node v's Testing procedure, node w still may not respond for an arbitrarily long time if F_w picks another edge as its fragment MOE.

What is the exact starting time for the Leader Distance procedure? Even though the Merging procedure can take a long time to complete, Awerbuch never explains at what point during this process a leader should start executing the Leader Distance procedure. If the leader waited for a response to its *connect* message before starting the Leader Distance procedure, then observation 3 tells us that it may fail to increase its level in a timely manner. Conversely, if the leader starts the Leader Distance procedure as soon as it sends its *connect* message then the algorithm may attempt to measure an unstable path length. In particular, it is possible that the path length later increases or decreases, causing Leader Distance to either terminate prematurely (delaying the level increase for its sub-fragment), or overestimate the distance to the new root (causing its sub-fragment to reach an excessively-high level). We describe these scenarios in greater detail below.

Observation 4 *Leader Distance might terminate prematurely, leaving a submitted sub-fragment without a level increase for a very long time.*

Figure 1 illustrates a situation where the path length from node v, the leader of submitted sub-fragment F_v, to the root of its new fragment *increases dramatically* during the execution of the Leader Distance procedure. Suppose that edge (v, w) is the fragment MOE for F_v, and that node v sends its *connect* message (labeled "2") to the adjacent node, w, in fragment F_w after w has already sent its *report* message (labeled "1") and entered the Found state. Thus, F_v will not simply be absorbed into the ongoing MOE-finding procedure in F_w. If we further assume that $L_v = L_w$, then node w cannot send an immediate reply to v's *connect* message either. Thus, node v will be blocked in the Merging state, where it begins executing the Leader Distance procedure by sending a *testDistance* message (labeled "3") through node w. Unfortunately, if F_w has not yet competed its MOE finding procedure, then the orientation of the Branches of F_w will carry the *testDistance* message to its root node. If the *testDistance* message arrives at the root with a positive counter, it will be discarded — thus terminating the Leader Distance procedure for node v. Later on, when it receives the *report* message from node y (labeled "4"), the root of F_w could decide to send a *changeRoot* message to y (labeled "5"), instructing it to submit to some other fragment F', which in turn has submitted to fragment F'', and so on. Thus, even though the Leader Distance procedure for node v was terminated, the distance from v to the new root could be really large. As a result, the next level increase for fragment F_v may take a really long time, thus violating the time optimality proof in [1].

Observation 5 *Leader Distance might allow a submitted sub-fragment to overestimate the distance to the new root, and thus reach an unreasonably-high level.*

Figure 2 shows that a slight change to the situation in Fig. 1 can cause the path length to the new root to *decrease* during the execution of the Leader Distance procedure. This time, suppose that the transmission time for the the *testDistance* message sent by node v (labeled "4") is delayed slightly, such that it will not reach the root of F_w until slightly after the arrival of the *report* message sent by node y (labeled "3"). In this case, just before v's *testDistance* message finishes its exploration of the path leading to the root of F_w, the root of F_w completes its MOE finding procedure and sends a *changeRoot* message to node y (labeled "5"). Because the *changeRoot* message reverses the pointer to the Upstream branch at every intermediate node, v's *testDistance* message will simply turn around and follow the *changeRoot* message to node y. As a result, the length of the path through F_v explored by v's *testDistance* message can be significantly larger than the actual distance between nodes w and y (i.e., the dashed line labeled "6"). In the worst case, v's *testDistance* message could visit every node in F_w twice, so if $|F_w|$ is sufficiently large, v could use the Leader Distance procedure to increase its level beyond $\log(N)$.

Because of these issues, it is obvious that the Merging procedure needs some refinement, especially its relationship to the Leader Distance procedure.

4.2 Root Distance and Leader Distance employ inconsistent path-length measurements

Even if the Merging procedure were modified to eliminate the unstable path-length problem described above, it is still possible for the leader of a submitted sub-fragment to reach a higher level than its root because of subtle differences in the timeout mechanisms employed by the Root Distance and Leader Distance procedures. In particular, the Root Distance procedure is based on adding a hop count field to the *initiate* message, which is initialized to 2^{L+1} by a level-L root. Thereafter each node that receives a copy of the *initiate* message decrements the hop count by *exactly one* before relaying a copy of the message to each of its children. Conversely, the Leader Distance procedure is based on adding hop count field to the *testDistance* message, which is initialized to 2^{L+1} by a level-L leader. Thereafter each node that receives a copy of the *testDistance* message decrements the hop count by *its number of children* before relaying a (single) copy of the message to its own parent. Thus, if the nodes along the path have an average of at least two children, Leader Distance will terminate at a higher level than Root Distance.

4.3 The treatment of expiring messages is not described completely

Awerbuch's description of the algorithm in [1] does not explain the procedures for managing the return of *expInit* messages to the root after the expiry of an *initiate* message, nor the return of an *ack* message to the particular leader that generated an expired *testDistance* message. Although both types of response message are individually generated (rather than part of a coordinated broadcast/convergecast pattern) and addressed to an individual destination, neither one can be sent using ordinary unicast routing methods without compromising the complexity of the algorithm in some way.

The main challenge is presented by the ack messages and their routing. In the initial specification [1], each leader propagates its own message, and each *ack* message is targeted to reach a specific leader. The question is how does a node know where to forward an ack to reach the right leader?

As we will see in our modified algorithm, we propose to address these issues by requiring nodes to maintain some extra state. The required state is within a constant factor from the state that a node needs to keep anyway.

4.4 The Leader Distance Procedure can create cycles

As described, the Leader Distance can increase the level of a leader too aggressively and this can create cycles. We can see that the Leader Distance and Root Distance procedures are exploring the same path between the leader and the root using the same threshold for level increase. Thus, it is possible that the leader will arrive at the same (or higher!) maximum level before the root.

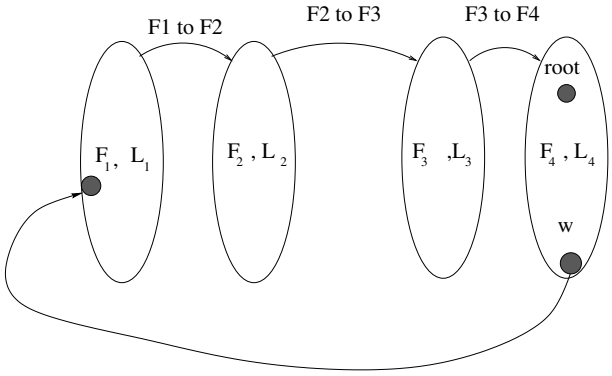

Fig. 3. The creation of a cycle

In more detail, consider the situation that results from executing the following sequence of four steps see Fig. 3.

1. Initially, we assume that all four fragments are in level L and disjoint. Then, (F_1, L) submits to (F_2, L), (F_2, L) submits to (F_3, L), and (F_3, L) equi-joins with (F_4, L) forming say $(F_5, L+1)$. In addition, we assume that the path length, p, between F_1 and the final root of F_5 via F_2 satisfies $2^{L+1} < p < 2^{L+2}$.
2. The leader of (F_1, L) initiates the Leader Distance procedure, which succeeds and promotes it to level $(F_1, L+1)$, because $2^{L+1} < p$. At the same time, the root of $(F_5, L+1)$ initiates a Searching procedure, but notice that p is too small for the Root Distance procedure to justify a level increase to $L+2$. Eventually, node w, which is adjacent to the *critical edge* that connects to $(F_1, L+1)$, receives the $initiate(F_5, L+1)$ message.
3. Node w of $(F_5, L+1)$ sends a test message along the critical edge. Assuming that F_1 has not yet received the $initiate(F_5, L+1)$ message, the answer will be immediate and positive since the two fragments have equal levels and different id's. Node w reports this edge as its MOE. This is an error according to the policy of the algorithm which demands reported edges to be outgoing.
4. Fragment F_1 receives the $initiate$ message, becomes part of F_5 and participates in the Searching procedure. As we already said, since $2^{L+1} < p < 2^{L+2}$, the Root Distance procedure of F_5 will not succeed and so F_5 will not rerun the finding procedure at a higher level. Instead, it will select the minimum weight edge from among those that have been reported. Should the critical edge turn out to be that minimum weight edge, node w will be appointed as the new leader — with instructions to connect along the critical edge and thus create a cycle.

In a nutshell, node w considers the critical edge as outgoing, because F_1 still has not received the final fragment identifier, but F_1 has increased its level enough on its own to be able to answer the $test$ message. However, the chosen MOE leads inside the same fragment, forming a cycle.

Neither GHS nor AWE provide any provision for detecting cycles in the MST, backtracking or repeating the Searching procedure. Thus, we need to modify the Leader Distance procedure to prevent symmetry-induced failures, such as Fig. 3.

4.5 Awerbuch's proof of optimal time complexity fails to handle the transition to Phase III

The proof of time complexity in Awerbuch's analysis does not cover the transition from Phase II to Phase III. Fundamental to that analysis is the following "proportional depth" assumption: no node currently at level L is more than 2^L hops below an actively-executing root or leader node. In particular, the proof of *Claim 2* from the time complexity analysis in [1] starts from the assumption that no node belonging to fragment F can be separated from its leader node by more than 2^L hops if F was at level L when it invoked the Phase III Merging procedure. Although fragment-joining according to the Phase III Merging procedure will *preserve* the proportional depth assumption *if it is true beforehand*, the following counterexample shows that Phase III fragments violating this assumption can be created during the transition period between Phase II and Phase III.

To see this, let us revisit the cycle-creation example shown in Fig. 3. This time, let us assume that the root fragment F_4 is already large enough to have switched to Phase III, whereas each member of the chain of fragments F_1, F_2, and F_3 is small enough to continue with Phase II. We further assume that the size of F_4 is too small for the root to trigger an immediate level increase due to the arrival of an *expInit* message or the detection of a high node count during the Reporting procedure.

Now consider what happens if, shortly after the F_4-node adjacent to the MOE for fragment F_3 sends its *report* message, the sequence of fragments F_3, F_2, and F_1 submit one after another to F_4. After each of the small fragments sends its *connect* message, it receives an immediate response in the form of an $initiate(F_4, L)$ message — which immediately raises each small fragment to level L without including it in the level-L Reporting procedure for F_4. Therefore, the root of F_4 will complete its Searching procedure without knowing about the submissions of this long chain of small fragments.

But in addition to the sequence of MOE edges by which the chain of small fragments joined F_4, there is one extra Basic edge that connects a node w in F_4 to the *last* small fragment in the series, F_1. Furthermore, suppose that this edge has the minimum weight of any Basic edge connected to node w. In that case, the Testing procedure at w — and hence the entire MOE-finding algorithm for F_4 — will be blocked until the entire chain of small fragments has been absorbed by F_4 and all of the associated nodes have reached level L.

At this point, our counterexample is complete: F_4 has now become a Phase III level-L fragment whose total size, $|F_4|$, and maximum path length can both be arbitrarily large. Moreover, since we assume that all of the small fragments F_1, F_2, . . ., are still in Phase II, none of them executes the Leader Distance procedure while waiting for the Merging procedure to complete. Thus, the set of nodes that belong to sub-fragment F_1 must remain at level L for a very long time, i.e., until a higher-level $initiate$ message has traversed the entire length of this arbitrarily large merged fragment. This contradicts the fundamental inductive basis for Awerbuch's time complexity proof, namely that during Phase III, the length of time that L remains the lowest level in the network bounded by $O(2^L)$. As a result of this counterexample, we must develop a new time complexity proof for the optimized Phase III algorithm.

Before leaving this example, we would like to point out that the technical difficulties that arise when fragment-joining occurs during the transition between Phases II and III cannot be solved by instructing the leader of fragment F_i to adopt the Merging algorithm for Phase III (i.e., including the Leader Distance procedure) if the new level of the sub-fragment is greater than $\log(N^*)$, where N^* is the phase III threshold size. Even though this strategy would meet our original goal of ensuring that the proportional depth assumption holds, it would also enable large numbers of small fragments to execute the Leader Distance algorithm, and hence increase the message complexity of the algorithm.

5 Modified Awerbuch algorithm – phase III

In this section, we provide a detailed description of a modified version of Awerbuch's algorithm that we will call **MA**. MA closely follows the general structure of AWE, which in turn is based on GHS with the addition of the Root Distance and Leader Distance procedures. However, MA includes numerous changes to address the flaws in AWE that were described earlier. In addition, our presentation clarifies certain missing details that are needed for an implementation (see [5]). We provide an overview of the main changes below.

First, MA extends the Root Distance procedure by allowing the root to increase its level when if finds that the total number of nodes is sufficiently high, and not just when it finds that the maximum path length is sufficiently long. Recall that the meaning of distance in both the Root Distance and Leader Distance is hop count along the spanning tree of a fragment.

Next, MA introduces two new messages, *MOEfound*, *acceptSub* compared to AWE, see Table 1. We could overload the meaning of existing messages, but we prefer to use the new messages for clarity. Roughly speaking, the *MOEfound* message is broadcast from the root to its fragment to notify all nodes that a MOE has been found.

Another change is that a node in MA can be at three states. The *active* Finding state remains as in GHS, where the node is actively participating in the search for the fragment MOE and has not completed its Reporting procedure. However, the *passive* Found state is split into two states, **FoundUndecided** and **FoundNotified** A node goes to the FoundUndecided state after sending its $report$. It enters the FoundNotified state, when it receives an notification from the root that a decision about the MOE has been made. The importance of the new states becomes apparent in the Merging procedure in Part II.

Finally, note a subtle dual use of the *MOEfound* message. In addition to triggering the change-of-state from FoundUndecided to FoundNotified for nodes outside the path connecting the root with the leader after the completion of the Searching procedure, it is also used by a submitted leader to announce a level increase to the rest of its sub-fragment when the Leader Distance procedure succeeds.

For facilitating the readers familiar with the previous algorithms, we highlight our changes using the keyword **NEW**. In Appendix B, we provide pseudocode with two types of actions procedures and responses to message arrivals. To facilitate the reading of pseudocode, we append in every step of the algorithm the related procedures or response-to-message that are related to the step.

5.1 Part I: finding the fragment MOE

Part I remains similar to GHS, except for the addition of the Root Distance procedure.

MA-0 An "ordinary" node is at the FoundUndecided state after completing its Reporting procedure. The FoundUndecided state is similar to the Found state described in GHS-0. In particular, a FoundUndecided node *delays its response* to all incoming *connect* messages, unless the level of the adjacent fragment is *strictly below* its own level.
NEW: After receiving a *changeRoot* or *MOEfound* message, the node advances to the **FoundNotified state**. A FoundNotified node responds immediately to an incoming *connect* message with the *same* level. The significance of this change will be apparent when we describe the Merging procedure.
Pseudocode. Procedures: ChangeRoot(). Messages: $changeRoot, MOEfound$.

MA-1 The initial state for the root node is the **Finding state**. The ordinary nodes in F enter the Finding state when they receive a copy of the *initiate* message from their Upstream Branch (towards the root).
The Searching procedure is the same as GHS-1, with the following additions. First, the $initiate(F, L, H)$ message includes a hop count field, H, which is initialized to 2^{L+1} by the root node and thereafter is decremented by one at the arrival of the message at every other node. Second, the Searching procedure *expires* when $H = 0$, causing the node to abandon the Searching procedure and execute the Root Distance procedure, which is explained below.
Pseudocode. Procedures: Initiate(). Messages: $initiate$.

MA-1.5 A node participates in the **Root Distance procedure** in two ways: detecting a long distance, and reporting such a distance to the root. For the detecting part, a node decrements the hop count field in an arriving *initiate(L)* message. When the hop count hits zero, $H = 0$, we say that Root Distance **succeeds**. Upon success, the node sends an $expInit(L)$ message towards the root.
Let's see how a node reacts to an $expInit(L)$ from one of its Downstream Branches, First, if the node is not the root, it simply relays the $expInit(L)$ message to its Upstream Branch edge, and enters the FoundNotified state, causing it to discard all subsequent $expInit(L)$ messages.
When the root receives an $expInit$ message, it increases its level to $L + 1$ and restarts the Searching procedure.
Implementing the reporting of $expInit$ messages. To prove the optimality of the message complexity, we need to argue that each node transmits only a fixed number of *expInit* message per level. This argument would not hold in the case where every *expInit* message generated by one of the leaf nodes in a balanced tree were forwarded independently back to the root. Thus, each node maintains enough state to filter out redundant *expInit* messages after relaying the first at a given level.
Pseudocode. Procedures: Initiate(), ExpInit(). Messages: $initiate, expInit$.

MA-2 There are no changes to Awerbuch's **Testing procedure**, which is the same as GHS-2 with the addition of the following timeout feature. During the Testing procedure, each node counts the total number of Basic edges that it has rejected. If this count for a level-L nodes reaches 2^{L+1}, the node abandons its Testing procedure and sends an *expInit* message to the root.

Observation 6 *Responding to a timeout during the Testing procedure by sending an* expInit *message should be allowed, because this event provides equivalent justification in comparison to Root Distance succeeding (MA-1.5) for abandoning the current search for the fragment MOE and restarting it at a higher fragment level. Moreover, sending the* expInit *message is also necessary to bound the running time for each iteration of Part I to $O(2^{L+1})$ when the fragment very large, but with small depth and many internal edges.*

Pseudocode. Procedures: Test(). Messages: $test, reject, accept$.

MA-3 The **Reporting procedure** is the same as GHS-3, with the following additions.
NEW: First the *report* message includes a node count field, C, in addition to the minimum MOE weight in the given subtree. Each node sets C to one plus the sum of the node counts reported by its children (if any). Second, if the final value of C is at least 2^{L+1}, then the root raises its level to $\lfloor \log_2(C) \rfloor$ and restarts the Searching procedure, rather than advancing to the ChangeRoot procedure in Part II.
Pseudocode. Procedures: Report(). Messages: $report, MOEfound$.

5.2 Part II: fragment merging along the MOE

In this part, we present the Merging procedure, which has most of the modifications of our MA algorithm.

MA-4 We introduce the following modification to the **ChangeRoot procedure** with respect to the GHS procedure[5]. In a nutshell, we require that all nodes become aware that their fragment has decided on its MOE.
NEW: The ChangeRoot procedure starts when the root decides on the new MOE for the fragment at the end of Reporting. It then sends a $changeRoot$ message on the path to the node with the MOE, and a $MOEfound$ message to all other nodes. Either message makes nodes to switch from the FoundUndecided to FoundNotified state.
Nodes located off the path between the root and leader receive an $MOEfound$ message, which they simply broadcast to each of their downstream Branch edges, and then switch from the FoundUndecided to FoundNotified state.

[5] Note that Awerbuch's algorithm retains the **ChangeRoot procedure** described in GHS-4.

The nodes located along the path between the root and leader receive a $changeRoot$ message. These nodes execute the basic ChangeRoot procedure described in GHS-4, and in addition broadcast an $MOEfound$ message over the rest of their downstream Branch edges, and then switch from the FoundUndecided to the FoundNotified state.

Observation 7 *After the ChangeRoot procedure, the Branches of the MST of a fragment are fixed in direction.*

The significance of switching to the FoundNotified state is that the tree of the fragment has now been reoriented towards the leader of the fragment. Thus, the Upstream pointers for every node in the FoundNotified state point towards the leader (and the root of the final fragment), and will not change again until the new fragment is formed and a new Searching procedure takes place. This observation avoids the malfunctioning of the Leader Distance procedure which existed in AWE.

Pseudocode. Procedures: ChangeRoot(). Messages: $changeRoot$, $MOEfound$.

MA-5 The modified algorithm uses Awerbuch's **Merging procedure**, which is based on GHS-5 augmented with the Leader Distance procedure. However, as we discussed in Sect. 4.1, the details of Merging procedure are not fully specified and require further refinement before they can be used.

NEW: We assume that Merging procedure starts when the leader node v sends a $connect(F_v, L_v)$ message across the MOE to notify the adjacent node w, which is in fragment (F_w, L_w) of its intent to merge. We have two cases[6], according to the relative levels:

1. If $L_w > L_v$ then node w replies immediately with an $acceptSub(F_w, L_w, S_w)$ message, where S_w is the state of node w. We see below how a leader reacts to an $acceptSub$ message. If node w is in FoundUndecided state, it will remember the edge and send one more $acceptSub$ message, when it enters the FoundNotified state.
2. If $L_w = L_v$, node w waits until it enters the FoundNotified state and then replies to F_v. The reply from w could be either to accept the submission of F_u or to proceed with an equi-join. If the response is a $connect$ message, then the two nodes complete the equi-join in constant time, and the new root[7] invokes the Searching procedure. Otherwise, the response must be an $acceptSub(F_w, L_w, S_w)$ message, causing F_v to submit to F_w. The submitting leader, v, will upgrade its level to match the other fragment, $L_v \leftarrow L_w$, and notify all ordinary nodes in F_v of the level increase by broadcasting another $MOEfound$ message.

[6] 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 the test message of v.

[7] New root in an equi-join is the node with maximum identifier or some other symmetry breaking rule could be applied [7].

NEW: Leader Distance starts only when node w enters the *FoundNotified* state. Thus, if the response from w carries state *FoundUndecided*, then node v will wait for w to become *FoundNotified*, and expect a *second* $acceptSub$ message. Clearly, we have at most two $acceptSub$ messages per submission, so the message complexity is not affected.

Pseudocode. Procedures: EquiJoin(). Messages: $connect$, $acceptSub$.

MA-6 In the **Leader Distance** procedure, the leader explores iteratively the distance towards the new root, and increases its level, if the distance is large. This way, leaders increase their level in a timely manner, after it has submitted and before it becomes an active member of its new fragment. **NEW:** Most of this step of the algorithm is significantly modified.

At the start of each iteration of the Leader Distance procedure, the leader launches a $testDistance(L, H)$ message towards the root, where L is the leader's current level and H is the maximum hop count before the message expires. If $testDistance(L, H)$ expires then the leader can increase its level. The expiration produces an $ack(L)$ message, which is sent back to the leader. Upon receiving the *ack* message, the leader increases its own sub-fragment level to $L+1$, broadcasts a new *MOEfound* message to all nodes belonging to its sub-fragment to notify them of the level increase, and immediately starts the next iteration of the Leader Distance procedure.

We define the mechanisms of Leader Distance more carefully to avoid the problems we identify in the AWE algorithm in Sect. 4.

The Rules of Leader Distance. In the MA algorithm, Leader Distance obeys the following rules. Assume as above, that the leader v of fragment F_v submits to w of fragment F_w.

Rule 1. A leader does not invoke the Leader Distance distance, until the Upstream node, w, enters the *FoundNotified* state.

Discussion for Rule 1. The idea is that Leader Distance explores an Upstream path of $FoundNotified$ nodes, so the path is not going to change. As a result, Rule 1 guarantees the following property, which is critical for the correct function of Leader Distance.

Observation 8 *A leader explores with the Leader Distance a part of the path towards the root, which is not going to change in direction or length.*

With Rule 1, node v can be blocked by the absence of a response from node w for a time that is *at most* proportional to the size of F_w fragment. The reason is that the root of a fragment completes its Reporting at level L_w in $O(2^{L_w})$. Note that the leader is also at level L_w, since its submission was accepted, so this delay does not affect the time optimality.

Observation 9 *The delay of starting the Leader Distance procedure at a leader of level L is $O(2^L)$.*

Rule 2. Leader Distance succeeds and generates an ack message if and only if:

- The hopcount of the $testDistance$ message hits zero, or
- The $testDistance$ message arrives at a node who has already seen a $testDistance$ message from another active leader who is currently at a higher level.

Discussion for Rule 2. The second clause of Rule 2 means that if there is another active leader at a higher level, then it is safe for any leader to reach that level. The importance of this will become apparent below, when we show how nodes can treat all Downstream active leaders with the same messages ensuring message optimality.

NEW: Before relaying the *testDistance* message to its parent, a node decrements its hop count by one (rather than its number of children, as specified in [1]). Furthermore, the initial value for the maximum hop count is set to 2^{L+2}, which is *twice as large* as the initial value specified in [1]. These changes are designed to handicap the ability of the Leader Distance procedure to trigger level increases in comparison to the Root Distance procedure, which leads to the following key observation that we will use in the correctness section. (Note that we implicitly use observation 8 about the stability of the explored paths.)

Observation 10 *Whenever the path from a leader to the root of the final fragment is long enough for Leader Distance to trigger an increase to L at the leader, it is guaranteed that Root Distance will ultimately trigger an increase to level $L + 1$ at the root.*

NEW: Optimizing Leader Distance. We propose an efficient solution in which each node serves as a "clearinghouse" for the coordination and sharing of Leader Distance information among its descendants. Intuitively, all Leader Distance procedures passing through a node are "merged": all descendant leaders are notified together for level increases. In fact, we use the most aggressively increasing leader to set the pace for all.

All nodes in the FoundNotified state maintain the following information in support of the Leader Distance procedure:

- $LDlevel$: the highest level so far attained by a leader in their subtree.
- $minLeaderDistance$: the minimum remaining hop-count among all *testDistance* messages at level $LDlevel$ which have passed through the node.
- An *active* status flag for each Downstream Branch, which is set when the edge points to leaders with an activated Leader Distance.

All these parameters are initialized when the node receives an $initiate$ message and starts a new instance of Searching: $LDlevel \leftarrow 0$, $minLeaderDistance \leftarrow \infty$, and $active$ flags are turned off for all edges.

When a node receives a $testDistance(L, H)$ message from Downstream Branch b, it sets b's *active* status flag and then proceeds as follows:

1. If $L < LDlevel$, the node sends an $ack(LDlevel - 1)$ message over edge b and drops the *testDistance* message. (Recall that the $ack(L)$ message means that the level-L Leader Distance procedure expired, so the leader can raise its level to $L + 1$.)
2. If $L = LDlevel$, we have two cases. The node ignores the *testDistance* message, if $H \geq minLeaderDistance$; otherwise, the node will set $minLeaderDistance \leftarrow H$ and forwards the *testDistance* message through its Upstream Branch edge.
3. If $L > LDlevel$, the node sends an $ack(L - 1)$ message to all *active* Downstream Branch edges except b to bring all other attached leaders up to this new level, sets $LDlevel \leftarrow L$ and $minLeaderDistance \leftarrow H$, and forwards the *testDistance* message through its Upstream Branch edge.

Conversely, if the node receives an $ack(L)$ message from its Upstream Branch edge, it discards the message if $L < LDlevel$; otherwise, it sets $LDlevel \leftarrow L + 1$ and forwards a copy of the *ack* message to all *active* Downstream Branch edges.

NEW: While a leader node remains blocked in the Merging state, it does not forward any other *testDistance* messages that may arrive from its children. Instead, the leader uses the data from any incoming *testDistance* messages to raise the level and/or reduce the hop count that will be carried by its own waiting *testDistance* message.

Pseudocode. Procedures: LeaderDistance(). Messages: $testDistance$, $acceptSub$, ack.

Note that as a result of these new features, it is now possible for a leader to increase its level by more than one at a single step. In addition, all active leaders in a given subtree will share a common set of level increases. This information-sharing can only speed up the level increases for these leaders, while at the same time reducing the total communications cost for Leader Distance.

6 Message complexity of MA

In this section, we show that the message complexity of MA algorithm is $O(E + N \log N)$, and hence optimal.

6.1 Phase I – preprocessing step

The message complexity of the first phase is bounded by $O(E + N \log N)$. For the sake of brevity, we refer the reader to the proof in the initial work [1].

6.2 Phase II – basic small-fragment MST algorithm

Recall that in this phase, we begin executing the GHS algorithm, but we exit this phase before GHS terminates. Recall that GHS is proven to have the optimal in message complexity [7]. Thus, the message complexity of this phase is bounded by $O(E + N \log N)$.

For completeness, let us examine the message complexity of a *complete* execution of GHS, although in our algorithm, this phase stops early and leads to the next phase.

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 $O(N \cdot \log(N))$.

6.3 Phase III: modified large-fragment MST algorithm

This phase is based on GHS, and we can apply the arguments we used above. Recall that MA uses many messages with the same name and functionality as GHS, such as $test$, $reject$, $connect$, $initiate$, $changeRoot$. For these messages, it is easy to show that the total number of these messages of MA are sent only a constant time over each edge, or sent once over each Branch for each level [7]. Below, we examine the new messages we introduce for each part of this phase separately.

Part I: Searching and Reporting. The only significant change here is the addition of the Root Distance procedure, which does not increase the message complexity in comparison to the basic GHS algorithm. To see this, we consider two cases. If the Root Distance proceedure does not succeed, then there is no change in the number of messages compared to the basic GHS algorithm. On the other hand, if Root Distance does succeed in increasing the level, then the algorithm will repeat Part I (Searching and Reporting) at a higher level, before advancing to Part II (fragment merging). Nevertheless, we can easily see that every node participates in *at most one* iteration of the Searching procedure at each level. Moreover, if Root Distance succeeded because of the return of some $expInit$ messages to the root (as opposed to node counting by the root), then we know that the total number of $expInit$ messages is bounded by the number of $initiate$ messages in the worst case. This is true for MA because each node forwards only the first $expInit$ message it sees at each level, and suppresses any other such messages.

Part II: Merging. The modification here is the introduction of the Leader Distance procedure. Let us discuss the problem and the solutions intuitively. A Leader Distance procedure can create too many $testDistance$ messages. To avoid this, we restrict the number of Leader Distance procedures that can be invoked, so that the total sum of $testDistance$ messages does not exceed the optimal number of messages, namely $O(N \log N)$ (see also [1]).

The MA algorithm only allows the Leader Distance procedure to be executed in Phase III. In Phase III, the minimum size of each fragment in Phase III is $\frac{N}{\log(N)}$, therefore we have at most $\log N$ distinct fragments in Phase III. This means that we can have at most $O(\log N)$ Leader Distance invocations in total. Thus, even if each procedure generated $O(N)$ *testDistance* and/or *ack* messages, the total number of messages due to this procedure would still only be $O(N \log N)$, which is not enough to make the message complexity suboptimal.

Note that the MA algorithm broadcasts an *MOEfound* message to all nodes in a fragment at most once per level, either in combination with the *changeRoot* procedure at the start of Phase II, or when the Leader Distance procedure succeeds. Thus, each edge of the MST will carry $\Omega(\log N)$ *MOEfound* messages, which is within the optimal bound. Finally, the $acceptSub$ messages are sent at most twice per Branch edge, for a total of $\Omega(N)$.

7 Time complexity of MA

In order to study the time complexity of the MA algorithm, we will need the following definitions. Let T_ℓ be the **earliest time** at which the lowest level in the network is at least ℓ. Similarly, we let $\tau_\ell \equiv T_{\ell+1} - T_\ell$ be the length of time that ℓ remains the lowest level in the network. We also define

$$\begin{aligned}\ell^* &\equiv \lceil \log(N^*) \rceil - 1 \\ &\leq \lceil \log(N) - \log\log(N) \rceil - 1 \\ &\leq \log(N)\end{aligned}$$

be the *highest* level that any fragment can reach in Phase II before it must switch to Phase III. Recall that $N^* = \frac{N}{\log(N)}$ is the phase III threshold.

7.1 Phase I – preprocessing step

This is the counting phase, whose purpose is to find the total number of nodes, N. In the original paper [1], an $O(E + N \log N)$ messages and $O(N)$ time algorithm is proposed.[8]

7.2 Phase II – basic small-fragment MST algorithm

In this phase, each fragment executes GHS until it reaches the phase III threshold size N^*. It can be shown [1] that $T_{\ell^*+1} = O(N)$ via the following argument.

Starting at time T_ℓ, the execution of the GHS algorithm in any level-ℓ fragment, say F, must proceed without any blocking. In particular:

- every $test$ message sent by a node in F will trigger an immediate response (see GHS-2), and
- if the leader of F sends a $connect$ message (see GHS-5) to the fragment, F' say, on the other side of its MOE, it will receive an immediate response unless both fragments are at the same level, and F' has not yet completed its own level-ℓ iteration of the GHS algorithm.

Thus, the key observation [6] is that $\tau_\ell \leq c \cdot S_\ell$, where c is some positive constant that is independent of ℓ, and S_ℓ is the size of the largest level-ℓ fragment. Therefore

$$T_{\ell^*+1} \equiv \sum_{\ell=0}^{\ell^*} \tau_\ell \leq c \cdot \sum_{\ell=0}^{\ell^*} S_\ell.$$

[8] If we assume that one node can be authorized to initiate the algorithm, the problem becomes simpler, the counting phase requires only $O(E)$ messages and $O(N)$ time [5].

Clearly, if $S_\ell \leq N^*$ for all ℓ then the time complexity result would be obviously true. Thus, we assume that $S_\ell > N^*$ for at least some values of ℓ. This situation could easily happen if a long chain of small fragments each decides to submit to its neighbor, as described in Sect. 4.5. Hence, we define $\beta = \{\ell : S_\ell > N^*\}$ to be the set of levels for which the maximum fragment size violates this "easy" upper bounding argument.

Since fragments are always merged to form larger fragments, and the root will switch to Phase III as soon as it determines that its fragment size is at least N^*, we recognize that a given node can only count toward *two* elements[9] from the set β. For the first occurrence, the root of the large fragment is not aware of its total size because the small fragments waited to send their *connect* messages until the receiving node had already sent its *report* message and entered the Found state. However, since these late-merging fragments will be incorporated into the next Searching procedure, the fragment will switch to Phase III at the end of the next iteration. Thus, we have the following:

$$\begin{aligned} T_{\ell^*+1} &\leq c \cdot \left(\sum_{\ell \notin \beta} S_\ell + \sum_{\ell \in \beta} S_\ell \right) \\ &\leq c \cdot \left(\log(N) \frac{N}{\log(N)} + 2N \right) \\ &\equiv O(N) \end{aligned}$$

which completes the time complexity proof for Phase II.

7.3 Phase III – modified large-fragment MST algorithm

We will now show that the time complexity for our modified Phase III algorithm is $O(N)$, i.e., that

$$\sum_{\ell=\ell^*+1}^{\log(N)} \tau_\ell = O(N)$$

Note that our proof is substantially different from the approach in [1] in order to address the problems described in Sects. 4.1 and 4.5. To facilitate our revised proof, we will begin by partitioning the nodes into four subsets:

- $\mathcal{R}(t)$: the set of nodes that are serving as the root of some fragment at time t,
- $\mathcal{L}(t)$: the set of nodes that are serving as an active leader for some sub-fragment at time t,
- $\mathcal{O}(t)$: the set of "ordinary" nodes, which are neither a root nor a leader at time t, and if their current level is L then they are located at most 2^L hops below the nearest root or active leader node, and
- $\mathcal{D}(t)$: the set of "deep" nodes, which are neither a root nor a leader at time t, and if their current level is L then they are located more than 2^L hops below the nearest root or active leader node.

[9] This condition was incorrectly given as *only one* element in [1], although a constant difference does not have a significant effect on the proof. We believe that the cause of this discrepancy is the lack of a precise definition in [1] for the level of a fragment that may have been involved in mergers other than an equi-join.

This partition forms the basis for our proof, where we separately account for the time required by each of the four subsets to advance from one level to the next during Phase III. In particular, let $\tau_\ell(\mathcal{R} \cup \mathcal{L} \cup \mathcal{O})$ be the time required to *update* every node that at time T_ℓ is both "non-deep" and has level ℓ, such that: (i) their levels have been increased by at least one; and (ii) their status as "non-deep" nodes has been restored if necessary. Similarly, let $\tau_\ell(\mathcal{D})$ be the time required to *update* every node that at time T_ℓ is both "deep" and has level ℓ, such that: (i) their levels have been increased by at least one; and (ii) they have been converted to "non-deep" nodes. Clearly

$$\begin{aligned} \tau_\ell &= \max\{\tau_\ell(\mathcal{R} \cup \mathcal{L} \cup \mathcal{O}), \tau_\ell(\mathcal{D})\} \\ &\leq \tau_\ell(\mathcal{R} \cup \mathcal{L} \cup \mathcal{O}) + \tau_\ell(\mathcal{D}), \end{aligned}$$

and hence the total time for Phase III is upper bounded by

$$\sum_{\ell=\ell^*+1}^{\log(N)} \tau_\ell(\mathcal{R} \cup \mathcal{L} \cup \mathcal{O}) + \sum_{\ell=\ell^*+1}^{\log(N)} \tau_\ell(\mathcal{D})$$

Thus, to prove that the time complexity for Phase III is $O(N)$ we will now show the following:

- Using a similar approach to [1], we show that $\tau_\ell(\mathcal{R} \cup \mathcal{L} \cup \mathcal{O}) = O(2^\ell)$, which is sufficient to show that the first summation is $O(N)$.
- If $\mathcal{D}_\ell(T_\ell)$ is the set of "deep" level-ℓ nodes present at time T_ℓ, then $\tau_\ell(\mathcal{D}) \equiv O(|\mathcal{D}_\ell(T_\ell)|)$.

Notice that the sets $\mathcal{D}_\ell(T_{\ell_1})$ and $\mathcal{D}_\ell(T_{\ell_2})$ are mutually disjoint across different levels, because the update time $\tau_\ell(\mathcal{D})$ continues until all of these "deep" nodes have been converted to "non-deep" status, and there is no mechanism in Phase III by which a "non-deep" node can be converted back to "deep" status. *Consequently, the second summation is also* $O(N)$.

Part 1. Calculation of $\tau_\ell(\mathcal{R} \cup \mathcal{L} \cup \mathcal{O})$

Consider the evolution of the system starting from time T_ℓ, the earliest time at which every node has reached level ℓ. We assume that $\ell > \ell^*$, so that all nodes have already switched to Phase III.

Suppose the set $\mathcal{R}(T_\ell)$ contains some level-ℓ root nodes. We note that T_ℓ is an upper bound on the starting time for the Searching procedure for every level-ℓ fragment. In addition, since ℓ is the minimum level, the MOE-finding algorithm for any node in a level-ℓ fragment cannot be blocked. Moreover, because of the timeouts provided by the Root Distance procedure (MA-1.5) and Awerbuch's additions to the Testing procedure (MA-2), the root of the level-ℓ fragment F must receive a response to its ongoing Searching procedure no later than time $T_\ell + c_1 \cdot 2^{\ell+1}$. At this point, the root of F must have either:

- **Restarted the MOE-finding algorithm at a higher level.** If the root of F receives an $expInit$ message or completes the Reporting procedure and finds that $|F| \geq 2^{\ell+1}$, then it must immediately raise its level and restart the MOE-finding algorithm.
- **Found the fragment MOE.** If the root completes the Reporting procedure and finds that $|F| < 2^{\ell+1}$, F will attempt to merge with the fragment F', which is adjacent to

its MOE. Thus, we must delete the root of F from $\mathcal{R}(\cdot)$ and at the same time add newly-created level-ℓ leader of F to $\mathcal{L}(\cdot)$. Using a further time of at most $c_2 \cdot 2^\ell$, the ChangeRoot procedure will move the leader of F to the node adjacent to the fragment MOE, at which point this final leader will send its *connect* message to the adjacent fragment, F'. If fragment F' is already at a higher level, then the leader of F will receive a response in $O(1)$ time. Otherwise, it must wait for at most $c_3 \cdot 2^{\ell+1}$ to receive a response from an adjacent level-ℓ fragment.[10] If the response from F' is another *connect* message, then the two leaders will complete an equi-join in $O(1)$ time, causing the two level-ℓ leaders for F and F' to be deleted from $\mathcal{L}(\cdot)$; one of them is immediately added as a level-$(\ell+1)$ root to $\mathcal{R}(\cdot)$, while the other becomes an ordinary level-ℓ node. Otherwise, the leader of F starts executing the Leader Distance procedure from level ℓ.

Therefore, no later than time $T_\ell + c_4 \cdot 2^{\ell+1} \equiv T_\ell + O(2^{\mathcal{L}+1})$, all remaining root nodes in $\mathcal{R}(\cdot)$ must have reached at least level $\ell+1$, where $c_4 = c_1 + c_2/2 + c_3$.

Now suppose the set $\mathcal{L}(T_\ell)$ contains some active level-ℓ leader nodes. Each of these nodes became a level-ℓ leader either by being a lower-level leader that was promoted to level ℓ as a result of the Leader Distance algorithm, or as a replacement for some level-ℓ root that found its fragment MOE. In the former case, the node would have immediately restarted the Leader Distance algorithm as soon as it reached the new level, whereas in the latter case, we just showed that each of these newly-created leader nodes starts executing the Leader Distance procedure by time $T_\ell + c_4 \cdot 2^{\ell+1}$ in the worst case. Now consider the distance between an active leader and the root, r, of its parent fragment:

- **Distance to root is more than $2^{\ell+2}$ hops.** In this case, the Leader Distance clearinghouse algorithm guarantees that the leader must increase its level to at least $\ell+1$ within an additional delay[11] of at most $c_5 \cdot 2^{\ell+2}$.
- **Distance to root is at most $2^{\ell+2}$ hops.** In this case, the Leader Distance clearinghouse algorithm might still be able to trigger a level increase using *leaderDistance* messages originating from more distant fragments, or else the Leader Distance algorithm could simply terminate because all *testDistance* messages that passed through this leader node managed to reach the root and were dropped.

  Observation 11 *Since leaders do not employ an explicit timeout function to detect the loss of their* testDistance *messages, they must recognize the termination of the Leader Distance algorithm (along with their own status as an active leader) by the arrival of an* initiate *message whose level is strictly higher than their own current level.*

  But recall that if the leader node is less than $2^{\ell+2}$ hops from the root, then we can easily upper bound the time delay until the node terminates its leader status upon the reception of a suitable *initiate* message. In particular, we just showed that the root will reach level $\ell+1$ no later than time $T_\ell + c_4 \cdot 2^{\ell+1}$. Thus, if the leader is within $2^{\ell+1}$ hops from the root, it will receive a level-$(\ell+1)$ *initiate* message within a further delay of $c_6 \cdot 2^{\ell+1}$. Conversely, if the leader is more than $2^{\ell+1}$ hops from the root (but less than $2^{\ell+2}$ hops, or Leader Distance would not have terminated!), then the root will restart the Searching procedure at level $\ell+2$ in time $c_1 \cdot 2^{\ell+2}$ and the level-$(\ell+2)$ *initiate* message will reach the leader within a further delay of $c_6 \cdot 2^{\ell+2}$.

Therefore, if we define $c_7 = c_4 + \max\{2c_5, 2c_6 + 2c_1\}$, then no later than time $T_\ell + c_7 \cdot 2^{\ell+1} \equiv T_\ell + O(2^{\mathcal{L}+1})$, every active leader node in $\mathcal{L}(T_\ell)$, together with any newly-created leaders created during the update process for root nodes in $\mathcal{R}(T_\ell)$, must have increased its level to at least level $\ell+1$, and possibly also been demoted from set $\mathcal{L}(\cdot)$ to set $\mathcal{O}(\cdot)$.

Finally, suppose the set $\mathcal{O}(T_\ell)$ contains some ordinary level-ℓ nodes. By definition, none of these level-ℓ ordinary nodes were located more than 2^ℓ hops below the nearest root or active leader node at the moment they reached level ℓ. Thus, once the corresponding active parent node raises itself beyond level ℓ, it will broadcast another *initiate* or *MOEfound* message to its (nearby) descendants, raising them to the same new level also. We consider three sub-cases in greater detail.

- **Active parent succeeded in raising its own level.** Suppose the active parent triggered the level increase by completing one iteration of the RootDistance procedure (within $c_1 \cdot 2^{\ell+1}$ time) or LeaderDistance procedure (within $c_5 \cdot 2^{\ell+2}$ time). Thereafter, all ordinary descendants of this active parent node will receive the level increase message within a further delay of $c_6 \cdot 2^{\ell+1}$, without changing the distance to their active parent, and without leaving the set $\mathcal{O}(\cdot)$.
- **Location of active parent changed because of an equi-join.** Suppose the active parent was a root that found its fragment MOE, which subsequently led to the completion of an equi-join with another level-ℓ fragment and its replacement by a level-$(\ell+1)$ root (within $c_4 \cdot 2^{\ell+1}$ time). Because the location of the leader could be maximally offset from the location of the root, the maximum distance from an ordinary node to the new level-$(\ell+1)$ root may double. Nevertheless, since the maximum depth limitation also doubles with each level increase, all ordinary nodes must remain in the set $\mathcal{O}(\cdot)$ after they receive the level increase message within a further delay of at most $2c_6 \cdot 2^{\ell+1}$.
- **Location of active parent changed when Leader Distance terminated.** Finally, suppose the active parent was a leader that lost its status (within $c_7 \cdot 2^{\ell+1}$) following the termination of the Leader Distance procedure because of insufficient distance separating this leader from the root. The maximum path length from the root to the furthest ordinary node below the terminated leader is $2^{\ell+2} + 2^\ell$, which is clearly less than $2^{\ell+3}$. Thus, in the worst case, the root may need to restart the Searching procedure one extra time before its level-$(\ell+3)$ *initiate* message finally reaches the furthest ordinary node no later than time $T_\ell + c_8 \cdot 2^{\ell+1}$, where $c_8 = c_7 + 4c_6 + 4c_1$. In addition, the ordinary nodes

[10] Note that the delay at this step is bounded for our modified Phase III algorithm because the introduction of the *MOEfound* messages allows us to guarantee a timely response to the *connect* message. As we explained in Sect. 4.1, this same step is not valid for Awerbuch's original algorithm because the response time by F' is not bounded.

[11] Note that in step MA-6 we added a new feature that leaders may need to delay the forwarding of other *leaderDistance* messages while waiting for a response to their own *connect* messages. However, this effect has already been incorporated into the constant c_4.

that previously belonged to the terminated leader now satisfy the depth requirements for ordinary nodes relative to the root.

Taking the worst of these three sub-cases, we see that no later than time $T_\ell + c_8 \cdot 2^{\ell+1} \equiv T_\ell + O(2^{\mathcal{L}+1})$, all ordinary nodes will have reached at least level $\ell + 1$, and none of those nodes can ever be added to the set $\mathcal{D}(\cdot)$.

Part 2. Calculation of $\tau_\ell(\mathcal{D})$

Suppose the set $\mathcal{D}(T_\ell)$ contains some deep level-ℓ nodes. Recall our example in Sect. 4.5 that showed how "deep" fragments can be created at any level greater than ℓ^* during the transition to Phase III, and that the existence of such fragments serves as a counterexample to Awerbuch's time complexity analysis described in [1]. Our approach to handling "deep" fragments is to recognize that a given "deep" level-ℓ fragment, F, can be transformed into an ordinary fragment of (much) higher level in $O(|F|)$ time. To see this, let us revisit the three sub-cases we introduced for the analysis of the set $\mathcal{O}$, while replacing the ordinary sub-fragment by a deep sub-fragment below the target active parent node. In each case, we assume a maximum path length of p between any deep level-ℓ node and its active parent node, where $\log(p) \gg \ell$.

- **Active parent succeeded in raising its own level.** Suppose the active parent is the root of the deep fragment, F (i.e., F must have been created during the transition to Phase III.) Based on our previous discussion of the ordinary nodes located below a terminated leader, we know that the root will increase its own level to $\ell + 1$, $\ell + 2$, ... using the rootDistance procedure and hence restart its Searching procedure at times no later than $T_\ell + c_1 \cdot 2^{\ell+1}$, $T_\ell + c_1 \cdot \left(2^{\ell+1} + 2^{\ell+2}\right)$, ..., respectively. For all levels up to $\lfloor \log(p) \rfloor$, the *initiate* messages will expire before reaching the furthest deep node in F. However, the *initiate* message for level $\lfloor \log(p) \rfloor + 1$ will reach the furthest deep node, within an additional time of at most $c_6 \cdot 2^{\lfloor \log(p) \rfloor + 1}$. Thus, in this case the length of the interruption to Phase III for transforming F into an ordinary level-$\lfloor \log(p) \rfloor + 1$ fragment is at most $(c_1 + c_6) \cdot 2^{\lfloor \log(p) \rfloor + 1} \equiv O(|F|)$.
  Conversely, suppose the active parent is the leader of the deep sub-fragment, F, and that the path length from the leader of F to the root of the entire fragment is so large that the Leader Distance algorithm will not terminate before the leader has raised its own level to $\lfloor \log(p) \rfloor + 1$. Thus, the same argument holds if we replace the time of the level updates for rootDistance by the larger update times (because of the larger hop count) appropriate for the leaderDistance, i.e., the time of the first level increase is no later than $T_\ell + c_5 \cdot 2^{\ell+2}$, the time of the second level increase is no later than $T_\ell + c_5 \cdot \left(2^{\ell+2} + 2^{\ell+3}\right)$, and so on. By upper bounding the geometric sum by the next term, we see that the elapsed time until the delivery of the final *MOEfound* message raises the level of the furthest deep node is at most $c_6 \cdot 2^{\lfloor \log(p) \rfloor + 1}$. Thus, the length of the interruption to Phase III for transforming F into an ordinary level-$\lfloor \log(p) \rfloor + 1$ sub-fragment is at most $(2c_5 + c_6) \cdot 2^{\lfloor \log(p) \rfloor + 1} \equiv O(|F|)$.
- **Location of active parent changed because of an equi-join.** In this case, the root of F does not realize it belongs to a deep fragment, and hence carries out an equi-join with another level-ℓ fragment, F' say, without first raising the level of its own fragment to $\lfloor \log(p) \rfloor + 1$. Using our previous result for the case where $\mathcal{R}(T_\ell)$ contains some level-ℓ root nodes, we see that F will complete its equi-join no later than time $T_\ell + c_4 \cdot 2^{\ell+1}$. But $\log(p) > \ell$, or F could not have been a deep fragment. Hence, accounting for the time to complete the equi-join introduces an additional delay of $c_4 \cdot 2^{p+1}$ in the worst case. Moreover, since the roots of both of the participating fragments thought they were at level ℓ, the maximum path length to the furthest deep node can increase by at most $O(2^{\ell+1})$ hops as a result of this equi-join, which will add at most one extra iteration to the Root Distance procedure. Thus, in this case the length of the interruption to Phase III for transforming F into an ordinary level-$\lfloor \log(p) \rfloor + 1$ fragment is at most $(c_4 + 2 \cdot c_1 + 2 \cdot c_6) \cdot 2^{\lfloor \log(p) \rfloor + 1} \equiv O(|F|)$.
- **Location of active parent changed when Leader Distance terminated.** Finally, suppose the active parent is the leader of the deep sub-fragment, F, and that the path length from the leader to the root is so short that the Leader Distance procedure terminates before the leader of F has raised its own level to $\lfloor \log(p) \rfloor + 1$. But in this case, the total distance from the root to the furthest deep node in F is upper bounded by $3p$. Thus, the previous analysis of the case where the active parent is the root of F can be used directly to show that the length of the interruption to Phase III for transforming F into an ordinary level-$\lfloor \log(3p) \rfloor + 1$ sub-fragment is at most $(c_1 + c_6) \cdot 2^{\lfloor \log(3p) \rfloor + 1} \equiv O(|F|)$.

This completes the time complexity proof for Phase III, and hence the entire modified Awerbuch algorithm.

8 Correctness of MA

In this section, we prove that the MA algorithm is correct. Namely, we show that it terminates and finds the MST. Note that the proofs are based on informal arguments, since a rigorous formal proof of correctness for even the basic GHS algorithm would require on the order of a hundred pages, as stated in Lynch's seminal book [10].

Theorem 1 *The proposed MA algorithm terminates.*

Proof First, we observe that as long as we have two or more fragments with different ids, at least one will attempt to join (e.g. send a *connect* message). We can prove this by contradiction. First, assume only two fragments with distinct ids. There exist at least one edge between them, and this edge must be Basic. Eventually, the fragment with the minimum level will test that edge. According to the algorithm, the reply will be immediate if the other fragment is of higher level. If the other fragment is of equal level, the delay in the reply will be limited to the running time for reporting procedure of the other fragment.

If more than two fragments exist, then consider the minimum-level fragment that has the minimum outgoing edge among all fragments of the minimum level. With arguments similar to the above, we can show that the reply to its test message will be answered in finite time. □

In addition, we must prove that the algorithm actually constructs a tree, which means that it does not create a cycle. In Appendix A, we illustrate how the proposed MA algorithm avoids creating cycles in our example of Sect. 4.4, where AWE fails. The following theorem shows that this property holds in the general case.

Theorem 2 *The proposed MA algorithm does not create cycles.*

Proof First, let us start from the GHS algorithm which is the basis for our algorithm. It can be proved that no cycles are created in the GHS algorithm [7] [10]. The key idea is that the decisions are taken centrally within a single fragment, and if multiple fragments decide to merge with each other at the same time there exists a hierarchy of levels to control the submission process: high-level fragments cannot submit to low-level ones. For equal-level fragments, we either have an equi-join, or else one fragment will wait until the other fragment increases its level and then accepts it as a submission to a higher level fragment.

In our algorithm, cycles might be created if the levels of the nodes are not updated properly. For this reason, we will examine the different mechanisms that increase the levels of a node, while we briefly mention how levels are used to ensure loop-freedom, which is also discussed in the original paper [7].

First, consider the Root Distance procedure. If Root Distance succeeds, the root throws away whatever information it had about the fragment MOE and restarts the Searching procedure at a higher level. Since no decisions to merge are taken without repeating the entire Searching procedure, it cannot create a cycle.

Second, consider the Leader Distance procedure. The key idea here is captured in the following Lemma. Let us first define as **final initiate** the $initiate$ message of a Searching procedure that will actually lead to identifying a MOE for a fragment. In contrast, non-final $initiate$ messages are followed by a new Searching procedure from the same source at a higher level.

Lemma 1 *A node that receives a final* $initiate$ *message of level L has never been at that level before.*

It is easy to prove this lemma, if we consider the rules of the Leader Distance procedure and the observations 8 and 10. In particular, observation 10 states that a leader with the Leader Distance procedure can only raise its sub-fragment to a lower level than the final level that the root will reach with the Root Distance. This is because the threshold for triggering a level increase with the Leader Distance is the discovery of a path length between the leader and the root that is twice as long as the threshold for Root Distance.

Let us use proof by contradiction and assume that a cycle is created. For this to happen, we need a node u of a fragment F_u at level L_u to attempt to join to a node w of F_w at L_w, which is already part of the same fragment, but is not aware of the new identity yet.

Node u will test the edge (u, w). (Note that we are only interested in the $test$ message sent in response to the final $initiate$ message, since the results of all previous Testing procedures will be discarded by the Root Distance procedure without ever considering them as a candidate for a fragment merging decision.) For w to answer this $test$ message, we have to have $F_w \neq F_u$ and $L_w \geq L_u$. Given that L_u is the level of the final $initiate$ message, it should be clear that node w must either have (i) a lower level than L_u, or (ii) the same fragment id, F_u. However, if case (i) still applies then node w is not allowed to send a reply (GHS-2), and must delay its response until case (ii) applies. Therefore, the only possible response by w to this $test$ message from u is a $reject$. □

Finally, it is not difficult to prove that the MA algorithm finds the minimum spanning tree. Recall that the MST problem can be solved by a "greedy" algorithm. It is sufficient to verify that the fragments identify and join along their MOE. The description of the algorithm 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 Searching procedure.

9 Conclusions

In this paper, we identified some problems with Awerbuch's distributed MST algorithm [1], involving both correctness and optimality issues. We then presented a modified algorithm (MA), which introduces several new features to solve these problems. We also show that with these changes, our MA algorithm satisfies the desired correctness properties, while meetining the time and message complexity bounds required for optimality.

This work arose from our pragmatic efforts to find a good MST algorithm for real applications, reported in [3,5]. 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.

Acknowledgements. 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 also thank Petros Faloutsos for his keen comments and support.

References

1. Awerbuch B: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In: Proceedings of the 19th Symposium on Theory of Computing, 1987, pp 230–240
2. Chin F, Ting HF: An almost linear time and $O(VlogV+E)$ messages distributed algorithm for minimum weight spanning trees. In: Proceedings of Foundations Of Computer Science (FOCS) Conference, Portland, Oregon, 1985
3. Faloutsos M: Corrections, improvements, simulations and optiMSTic algorithms for the distributed minimum spanning tree problem. Master's thesis, University of Toronto, Computer Science, 1995. Master's thesis, Technical Report CSRI-316
4. Faloutsos M, Molle M: Optimal distributed algorithm for minimum spanning trees revisited. In: Proceedings of 1995 Principles Of Distributed Computing (PODC), 1995

5. Faloutsos M, Molle M: What features really make distributed minimum spanning trees algorithms efficient? In: Proceedings of the 1996 International Conference on Parallel and Distributed Systems (ICPADS '96), Tokyo, 1996, pp 106–114
6. Gafni E: Improvements in the time complexity of two message-optimal election algorithms. In: Proceedings of 1985 Principles of Distributed Computing (PODC), Ontario, 1985
7. Gallager RG, Humblet PA, Spira PM: A distributed algorithm for minimum weight spanning trees. ACM Trans. on Programming Languages and Systems 5(1):66–77 (1983)
8. Garay JA, Kutten S, Peleg D: A sub-linear time distributed algorithm for minimum-weight spanning trees. In: Proceedings of Foundations of Computer Science (FOCS), 34, 1993
9. Kutten S, Peleg D: Fast distributed construction of k-dominating sets and applications. In: Proceedings of Principles of Distributed Computing (PODC), 1995
10. Lynch N: Distributed Algorithms. Morgan Kaufmann, 1997
11. Singh G, Bernstein AJ: A highly asynchronous minimum spanning tree protocol. Distributed Computing 8(3):♣ (1995)

A Correctness: How MA avoids the cycle problem of Sect. 4.4

Here we show how the proposed MA algorithm avoids cycles in the example in Fig. 3, in which AWE did create cycles.

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 Searching procedure). In [1], 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.

B Pseudocode implementation

We present the pseudocode that implements the phase II and III of our algorithm. Before the pseudocode, we present the variables and some conventions that we use.

Messages arrive and are stored in a FIFO message queue. Nodes respond to messages in order of arrival. One possible response to a message is to put it at the back of the queue effectively delaying its handling.

Here are the variables node uses:
myID: is the node id;
myLevel: is the level of the node;
myStatus: is the status of the node;
myFid: is the fragment id that a node belongs to;
inbranch: is the edge towards the root or leader;
myHops: is the number of hops I am away from the current root;
bestEdge: is the direction or the edge where the MOE can be found;
bestWeight: is the weight of the current MOE;
testEdge: is the adjacent edge that a node is currently testing to see if it is my local MOE

Boolean:
IamRoot: is true when the current node is the root;
IamLeader is true when the current node is the leader;
localMOEfound: is true when a node has found the MOE edge among its adjacent edges;
myPhaseIII is true if the fragment is in phase III;
(myPhaseIII is set by each root upon receiving the fragment size at the end of the Reporting procedure and is communicated to the leader)

A node uses a variety of counter variables:
reportCount measures how many reports from children I expect;
sizeCount measures the size of my subtree according to the reports I have received so far;

A node uses several variables for each adjacent edge i:
edge(i).status is the current status of the edge: Basic, Branch, Rejected;
edge(i).active is a flag used in Leader Distance procedure (see below)
edge(i).resendAcceptSub is a flag to resend an acceptSub msg when the current node switches to FoundNotified state to a node that currently it has sent an acceptSub(FoundUndecided)

We list the variables that are used to implement the Leader Distance procedure:
edge(i).active is a flag that marks whether there are leaders with active Leader Distance procedures
LDlevel: the maximum level of a leader in my subtree;
minLeaderDistance: the minimum number of hops that a $testDistance$ message of the LDlevel

Here are some conventions we use:

- Procedures start with capital letters, while messages start with low case letters.
- The variables of a node start with lowercase letter, while the variables of an incoming message start with capital letters.
- Names of states start with capital letters.
- Messages carry the level, fragment id, node id, and status of the sender node. We highlight the variables we use in each procedure.

The pseudocode below should be seen as the essential backbone of the algorithm. Our experience with implementing these algorithms [5] suggests that an actual implementation should provide safeguards and redundant checks, since debugging a distributed application is many times more difficult than a sequential procedure.

```
Procedure WakeUp();
.     // Initialize the variables of a node
.     // Executed once in the beginning for each node
```

```
.   myFid = myID;  // I am my own fragment
.   myLevel = 0;
.   myStatus = Find;
.   IamRoot = true;
.   myPhaseIII = false;
.   execute procedure Initiate()
.
Response to message initiate(Level, Fid, Stat, Hops) on
edge j;
.   if Hops == 0 and myPhaseIII == true then
.       send expInit(Level) on edge j;
.   inbranch = j;  myHops = Hops;
.   if myLevel ≤ Level then begin
.       myLevel = Level;
.       myFid = Fid;  myStatus = Stat;
.       execute procedure Initiate(Hops)
.   end
.   // If my level is higher then I am a successful leader
.   // and I will wait for a later initiate at a higher level
.
Procedure Initiate(Hops)
.   // initialize all variables that participate in searching,
.   // testing, reporting
.   bestEdge = nil;  bestWeight = ∞;
.   testEdge = nil;  localMOEfound = false;
.   IamLeader = false; // In case, I was leader
.   reportCount = 0;
.   sizeCount = 1; // sizeCount count yourself
.   // Send initiate to children and initialize edge flags
.   for every branch edge k and k ≠ j do begin
.       edge(k).active = false ;
.       edge(k).resendAcceptSub = false ;
.       reportCount = reportCount + 1;
.       send initiate(Level, Fid, Stat, Hops–1) on edge k;
.   end
.
Procedure Test()
.   if  there are no adjacent edges in Basic state then begin
.       // localMOE does not exist
.       localMOEfound = true;
.       execute procedure Report()
.   end else if I rejected more than 2^(myLevel+1) edges
.       in this Finding then begin
.       execute procedure ExpInit(myLevel);
.   end else begin
.       testEdge = min weight edge in Basic state;
.       send test(myLevel, myFid) on testEdge;
.   end
.
Response to message expInit(L) on edge j;
.       execute procedure ExpInit(L);
.
Procedure ExpInit(level L);
.   // Report only the first expInit at this level to root
.   // Root increases level and repeats Initiate() procedure
.   if myStatus == Find and L ≥ myLevel
.       and PhaseIII == true then begin
.       if IamRoot == false then begin
.           myStatus == FoundNotified;
.           // Ignore other expInit
.           send expInit(Level) on edge j;
.       end else  // I am root
.           myLevel = myLevel + 1;  // Increase level
.           execute procedure Initiate(2^(myLevel+1))
.       end
.   end
.
Response to message test(Level, Fid) on edge j;
.   if myFid == Fid then begin // Reject Edge
.       edge(j).status = Rejected;
.       if testEdge == j then  // I have sent test
.           // on j, so other node will reject it too
.           execute procedure Test();
.       else send Reject(myFid) on edge j;
.   end else if Level > myLevel then
.       place message at the message queue;
.   else // Level ≤ myLevel and different fragment
.       send accept(myLevel, myFid) on edge j;
.
Response to message accept on edge j;
.   // answer to test message
.   testEdge = nil;  localMOEfound = true;
.   if weight(j) < bestWeight then
.       begin bestEdge = j;  bestWeight = weight(j); end
.   execute procedure Report();
.
Response to message reject on edge j;
.   // A reject may come delayed over a Branch
.   if edge(j).status == Basic then
.       edge(j).status = Rejected;
.   execute procedure Test();
.
Response to message report(Weight, Count) on j;
.   reportCount = reportCount – 1;
.   sizeCount = sizeCount + Count;
.   if Weight < bestWeight then
.       begin bestWeight = Weight;  bestEdge = j; end
.   execute procedure Report()
.
Procedure Report()
.   if reportCount == 0 and localMOEfound == true
.   then begin  // Reporting is completed
.       if IamRoot == false then begin
.           // Forward report to root
.           mystate = foundUndecided;
.           send report(bestEdge, sizeCount) on inbranch;
.       end else begin // Root: decide what to do
.           if bestWeight == ∞ then
.               halt ; // Algorithm Ends
.           if sizeCount > SizeThreshold then
.               myPhaseIII = true;
.           if sizeCount > 2^(myLevel+1) then begin
.               // size has triggered a level increase
.               myLevel = myLevel + 1;
.               execute procedure Initiate();
.           end else begin
.               mystate = FoundDecided;
.               IamRoot = false;
.               execute procedure ChangeRoot()
.           end
.       end
```

```
.     end
.     // Note: if Reporting is not complete, do nothing
.
Procedure ChangeRoot()
.     send MOEfound to all Branch edges except
.         bestEdge;
.     myState = FoundNotified;
.     execute procedure ResendAcceptSub();
.     if edge(bestEdge).state == Branch then begin
.         // Forward changeroot to leader
.         send changeRoot(myPhaseIII) on bestEdge;
.         inbranch = bestEdge;
.         // orient tree towards the leader
.     end else begin
.         // you are leader, your bestEdge is the MOE
.         IamLeader = true;
.         send connect(myLevel, myFid) on bestEdge;
.         edge(bestEdge).status = Branch;
.     end
.
Procedure ResendAcceptSub()
.     // Send acceptSub(FoundNotified) to those nodes
.     // that I had sent acceptSub(FoundUndecided)
.     // Invoked whenever I switch to FoundNotified
.     for every Branch edge k with edge(i).resendAcceptSub
.         == true do begin
.         send acceptSub(myLevel, myFid, mystate)
.             on edge k;
.         edge(i).resendAcceptSub = false
.     end
.
Response to message changeRoot(PhaseIII) on edge i
.     myPhaseIII = PhaseIII
.     execute procedure ChangeRoot()
.
Procedure SendAcceptSub(edge j)
.     send acceptSub(myLevel, myFid, myStatus) on edge j;
.     if mystate == FoundUndecided then
.         edge(j).resendAcceptSub = true
.     // remember to resend acceptSub when FoundNotified
.
Response to connect(Level, Fid, NodeID) on edge j;
.     // NodeID is the id of the sending node
.     if L < myLevel then begin   // accept submission
.         if mystate == Find then begin
.             // submission joins Finding procedure
.             send initiate(myLevel, myFid,myStatus,
.                 myHops – 1 ) on edge j;
.             reportCount = reportCount + 1;
.         end else execute SendAcceptSub(j)
.     end
.     else if L == myLevel and myStatus == FoundNotified
.     then begin
.         if IamLeader == true and bestEdge == j then
.             execute procedure EquiJoin(j, NodeID)
.         else execute SendAcceptSub(j)
.     end else
.         // L == myLevel and myStatus is not FoundNotified
.         put connect back in message queue
.
Response on acceptSub(Level, Fid, Status) from edge j;
.     if mylevel < Level then begin
.         //increase level of fragment
.         mylevel = Level;
.         for every Branch edge and k ≠ j do
.             send MOEfound(mylevel, myFid) on edge k;
.     end
.     if Status == FoundNotified and myPhaseIII == true
.     then execute procedure LeaderDistance()
.
Response to MOEfound(Level, Fid, Status) on edge j;
.     // Message has two purposes:
.     // switch to FoundNotified, increase level
.     if myLevel < Level or myStatus = FoundUndecided
.     then begin
.         myLevel = Level;
.         myStatus = FoundNotified;
.         for every Branch edge and k ≠ j do
.             send MOEfound(myLevel, myFid,
.                 myStatus) on k;
.     end
.
Procedure LeaderDistance()
.     LDlevel = myLevel;
.     // Initialize max known leader level to my level
.     MaxRootDistance = 2^(myLevel+2);
.     send testDistance(myLevel, MaxRootDistance)
.         on inbranch;
.
Response to testDistance(Level, RemainingDistance) on
edge j;
.     edge(j).active = true;
.     if RemainingDistance == 0 and Level ≥ LDlevel
.     then begin
.         // Message expires, and all active leaders
.         // will increase their levels
.         LDlevel = Level + 1;   minLeaderDistance = ∞;
.         for every branch k with edge(k).active == true,
.         k ≠ j do
.             send acknowledgment(Level) on edge k ;
.     end
.     else if Level < LDlevel then
.         // Higher level leader exists,
.         // sender of this msg should increase its level
.         send acknowledgment(LDlevel – 1) on edge j;
.     else if Level == LDlevel then begin
.         // Keep the distance from the furthest leader
.         if RemainingDistance < minLeaderDistance then
.         begin
.             minLeaderDistance = RemainingDistance;
.             send testDistance(Level, RemainingDistance
.                 – 1) on inbranch;
.         end
.     end
.     else if Level > LDlevel then begin
.         // New msg has higher level:
.         // all other leaders must increase their level
.         LDlevel = Level;
.         minLeaderDistance = RemainingDistance;
.         for every branch k with edge(k).active == true and
.         k ≠ j do
```

```
.           send acknowledgment(Level – 1) on edge k ;
.       send testDistance(Level, RemainingDistance – 1)
.           on inbranch;
.   end
.
Response to acknowledgment(Level) on edge j;
.   if IamLeader == true and myLevel ≤ Level then begin
.       myLevel = Level + 1;
.       for all branches k, k≠ j
.           send MOEfound(myLevel) on k;
.
Procedure EquiJoin(edge j, NodeID Nid)
.   // Pick the root of the new fragment
.   // Break the tie between the two nodes: any method goes
.   // Here use the min node id
.   IamLeader = false;
.   if myNodeID < NodeID then begin
.       IamRoot = Yes;
.   myLevel = myLevel + 1;   // Increase level for equijoin
.       execute procedure Initiate()
.   end
```