Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and related problems

(Detailed Summary)

Baruch Awerbuch †

Department of Mathematics and
Laboratory for Computer Science,
MIT, Cambridge, MA 02139

Abstract

This paper develops linear time distributed algorithms for a class of problems in an asynchronous communication network. Those problems include Minimum-Weight Spanning Tree (MST), Leader Election, counting the number of network nodes, and computing a sensitive decomposable function (e.g. majority, parity, maximum, OR, AND).

The main problem considered is the problem of finding the MST. This problem, which has been known for at least 9 years, is one of the most fundamental and the most studied problems in the field of distributed network algorithms.

Any algorithm for any one of the problems above requires at least $\Omega(E+V\log V)$ communication and and $\Omega(V)$ time in the general network. In this paper, we present new algorithms, which achieve those lower bounds. The best previous algorithm requires $\Theta(E+V\log V)$ in communication and $\Theta(V\cdot\log^* V)$ in time.

Our result enables to improve algorithms for many other problems in distributed computing, achieving lower bounds on their communication and time complexities.

† Supported by Chaim Weizmann Postdoctoral Fellowship and Air Force contract TNDGAFOSR-86-0078.

1. Introduction and Summary

1.1. Motivation

The problem of finding a distributed algorithm for a minimum weight spanning tree is a fundamental problem in the field of distributed network algorithms. Trees are an essential structure in various communication protocols, e.g. network synchronization [A-85], Breadth-First-Search [AG-85], and Deadlock Resolution [AM-86]. For the purpose of disseminating information in the network, it is advantageous to broadcast it over a minimum-weight spanning tree [DM-78], [AE-86], since information will be delivered to every node with small communication cost.

The problem of finding a leader is reducible to the problem of finding a spanning tree. In turn, leader election is an important tool for breaking symmetry in a distributed system. It allows application of highly centralized protocols in a completely decentralized environment, thus providing higher degree of control over the operation of the network. Among other systems applications, leader election is used in order to replace a malfunctioning central lock-coordinator in a distributed data-base [MMP-78], for finding a primary site in a replicated distributed file systems [AD-76], etc.

There are other problems, which are very closely related to the problems of finding a spanning tree or finding a leader. *Counting* the number of network nodes is one of them. There exists a class of functions, referred to as ***sensitive decomposable*** functions; such functions are sensitive to every input, but the influence of a set of arguments can be represented by a string whose size is not much bigger than the size of a string needed to represent just one argument. Examples of such functions are *maximum, sum, parity, majority, OR, AND.* As shown in this paper, complexity of finding a spanning tree, complexity of counting the number of network nodes and complexity of computing distributively any sensitive decomposable function are all within a constant factor of each other.

To summarize, construction of a spanning tree or finding a leader appears as a building block essentially in every complex network protocol, and is closely related to many problems in distributed computing.

1.2. Existing Results

The problem of finding a Minimum Spanning Tree (MST) in a distributed network has been studied since 1977. Dozens papers have been written on the subject, from [S-77],[K-78],[GHS-83] to [CT-85],[G-85]. A truly pioneering work of Gallager, Humblet and Spira [GHS-83] presented an algorithm which requires $\Theta(E+V\log V)$ messages and $\Theta(V\log V)$ time, introducing very fundamental ideas and concepts into the field of distributed network algorithms. Some researchers investigated lower and upper bounds for leader election algorithms in special network models, like ring, complete network, and grid. Among the numerous papers, let us mention [Afek-85], [FL-84], [KMZ-84], [B-80]. Until now, all existing algorithms for counting, leader election, etc. used the MST as a major building block and thus are dominated by its complexity.

As observed already in [GHS-83], and made precise in [AGV-87], $\Omega(E)$ messages are necessary in order to construct a spanning tree. As proved in [B-80] and [FL-84], $\Omega(V\log V)$ messages are needed in order to find a leader on a ring. It follows that $\Omega(E+V\log V)$ messages are necessary to construct a spanning tree in a general network. It is also obvious that in a network having $O(V)$ diameter, any distributed algorithm must take at least $\Omega(V)$ time, since this time is required just to traverse the network. Thus, the communication achieved in [GHS-83] is optimum and the time is within $\Theta(\log V)$ factor of the optimum.

For the general network model, the best algorithm currently known both for Spanning Tree and Minimum Weight Spanning Tree has been given by Chin and Ting [CT-85], and Gafni [G-85]. The time complexity of their algorithm is $\Theta(V\log^* V)$, i.e. slightly better than [GHS-83], while the communication complexity of the algorithm is still $\Theta(E+V\log V)$. The algorithm is almost optimal, except for $\Theta(\log^* V)$ factor in time. [CT-85] show that their time bound is tight by giving an example of a network, on which the the algorithm runs for $\Theta(V\log^* V)$ time. The cause of $\Theta(\log^* V)$ factor in time complexity in [G-85],[CT-85] is the fact that small trees sometimes wait for big trees. As a result, the waiting relations between trees has a complex combinatorial structure which is extremely hard to analyze.

The time complexity is a very meaningful measure for evaluating performance of a spanning tree algorithm, because of the nature of its applications, e.g. control and coordination of various network processes. Thus, from the theoretical point of view, it is challenging to reduce the time complexity of spanning tree algorithms to its optimum value.

1.3. Our Results

The mission of finding the ultimate algorithms for the Minimum Spanning Tree, Counting, Leader Election, and other related problems is accomplished in the current paper. We present here a new MST algorithm, that requires $O(E+V\log V)$ messages and $O(V)$ time, i.e. is optimal *both* in communication and time.

Besides Spanning Tree and Minimum Spanning Tree, our result yields an $\Theta(\log^* V)$ improvement in time complexity of other algorithms, which use Spanning Tree as a subroutine. Those include Leader Election [FL-84], Deadlock Resolution [AM-86], Counting the number of network nodes, and computation of *any* decomposables sensitive function. As a result, the algorithms for those problems reach the lower bounds in both time and communication complexities. Another example of a problem whose solution is improved is the Network Partitioning problem [A-85] which arises in the context of network synchronization.

Additional contribution is the following theorem, whose proof will be given in the full paper.

Completeness Theorem: The communication and time complexities of the problems of computing distributively a sensitive decomposable function, counting the number of network nodes and finding a spanning tree are within constant factor of each other.

Corollary: Any improvement in the complexity of the spanning tree algorithm leads to the same improvement in complexities of counting and of computing a sensitive decomposable function.

The improved performance of our MST algorithm is due to two new techniques. The first technique enables to estimate distributed time of a communication procedure in an asynchronous network. This technique has been adopted in the Acyclic Partition algorithm of [AM-86], reducing its time complexity to linear (even though it does not use Spanning tree as subroutine). The second technique has been adopted in [AP-86] for estimating the size of a dynamically growing tree.

Our MST algorithm consists of two stages. The first stage, referred to as *Counting* stage, finds *some* spanning tree and computes the number of nodes in the network. The second stage, referred to as the *MST* stage, receives as an input the number of nodes in the network and, using this information, finds the Minimum Weight Spanning Tree. Both stages require $O(E+V\log V)$ messages and $O(V)$ time, i.e. are optimal *both* in communication and time. It is worth mentioning that while the two stages use similar techniques for reducing the complexities, the algorithmic approaches taken are radically different.

The main idea of our improvement is to avoid situations in which small tree waits for big tree. This simplifies the structure of the waiting relations significantly, reduces the time complexity to linear, and simplifies enormously the analysis. The Counting stage first finds *some* spanning tree and elects a leader in the network. Then, the number of nodes is computed easily. It uses different algorithmic approach compared to [G-85],[CT-85] which is enabled by the fact that we do not insist that the produced spanning tree is necessarily minimum weight. Thus, it is relatively "easy" to guarantee that small trees never wait for big trees.

The rest of this paper is organized as follows. Section 2 describes the model, the complexity measures, and the problem of finding the MST. Section 3 presents stage 2 of the algorithm (MST), while section 4 presents stage 1 (Counting).

2. The problem

2.1. The model and the the complexity measures

We consider here the standard model of static asynchronous network [A-85],[AG-85]. This is a point-to-point communication network, described by an undirected *communication graph (V,E)* where the set of nodes V represents processors of the network and the set of edges E represents bidirectional non-interfering communication channels operating between neighboring nodes. No common memory is shared by the node's processors. All the processors have distinct identities. However, a node does not know identity of its neighbors. We confine ourselves only to event-driven algorithms, which do not use time-outs, i.e. nodes cannot access a global clock in order to decide what to do. This is a common model for static communication networks [GHS-83], [A-85],[AG-85].

The following complexity measures are used to evaluate performance for distributed algorithms. *The Communication Complexity, C,* is the worst case total number of elementary messages sent during the algorithm, where an elementary message contains at most $O(\log V)$ bits. *The Time Complexity, T,* is the maximum possible number of time units from start to the completion of the algorithm, assuming that the inter-message delay and the propagation delay of an edge is at most one time unit of some global clock. This assumption is used only for the purpose of evaluating the performance of the algorithm, but cannot be used to prove its correctness, since the algorithm is event-driven.

2.2. The problem of finding the MST

In a distributed algorithm for the Minimum Spanning Tree problem, each node has a copy of a node algorithm determining the response of the node to messages received at that node. Namely, the algorithm specifies which computations should be performed and which messages should be sent. The algorithm is started independently by all the nodes, perhaps at different times. At the start time, each node is ignorant of the global network topology except for its own edges. Upon the termination of the algorithm, every node knows its neighbors in the minimum spanning tree.

Without loss of generality, we assume that all the links are assigned distinct weights with a total ordering defined on the domain of the weights. This condition guarantees uniqueness of the minimum spanning tree. It is easy to achieve this, as observed in [GHS-83], by simply assigning the weight of each link (i,j) as the tuple [max(i,j),min(i,j)], and comparing these tuples lexicographically.

Similarly, one can define the problems of finding *some* spanning tree, electing a leader, counting the number of network nodes, and computing a sensitive decomposable function.

3. Stage 2: Minimum Spanning Tree

3.1. Background

Most distributed and parallel MST algorithms operate according to the following scheme. The algorithm maintains a spanning forest of rooted trees, each tree being a sub-tree of the MST. Initially, every tree consists of a single node. Upon termination of the algorithm, there will be a single tree spanning the whole network. For any node, its *father* is the next node on the path to the root; root has no father. The tree is represented by "father" pointers, leading from each non-root node to its father. The *best* edge of a tree is the minimum weight edge among all edges leading from it to other trees. Since edge weights are unique, the best edge must be in the MST. In the course of the algorithm, every tree finds its best edge and hooks itself (gets "absorbed" in terminology of [GHS-83]) onto the tree on the other side of that edge, becoming a sub-tree in bigger tree. This hooking is represented by the following manipulation of the father pointers. First, the root of the tree is moved to the internal endpoint of the best edge, changing father pointers accordingly. Second, the external end of the best edge becomes the father of the root. Note that two trees can hook onto each other if they both have the same best edge. Such edge is called the *core* edge [GHS-83]; its endpoints are roots in corresponding trees. This creates a cycle of length two in the pointer graph. To break such cycle, the hooking of the root with bigger identity is canceled; it becomes the root of the combined tree. Now, all nodes in the combined tree are informed about the name of their new root and requested to look for the best edge.

The communication and time complexity of election of the best edge and updating father pointers is linear in the size of the tree. Thus, a naive distributed implementation of that algorithm would require $O(V^2)$ messages and time, since a tree of size $\frac{V}{2}$ could be hooked onto other trees $\frac{V}{2}$ times, each hooking requiring linear work. A classical idea, which is well known for sequential algorithms, e.g. Union-Find, is that in order to merge two trees, it is advantageous to hook the smaller tree onto the bigger one, thus updating only the pointers of the smaller tree. Thus, each time a pointer of a node is changed, the size of the combined tree is at least doubled; it follows that the number of pointer changes at each node is at most $\log_2 V$. To achieve this, one must somehow ensure that the best edge of a tree always leads to a bigger or equal tree, then it would achieve communication complexity of $O(E+V\log V)$ and time complexity of $V\log V$. The problem is that in distributed systems, it is hard to estimate the size of a tree; counting in a naive way would by itself require $O(V^2)$ messages and time.

The first solution of that problem was achieved in [GHS-83] using the technique of *levels*. The level of a tree containing one node is 0. If two trees of the same level create a core, then the level of the combined tree is increased by 1. Level is a lower bound on the logarithm of the cardinality of the tree and thus the maximum achievable level is $\log_2 V$. The algorithm guarantees that the best edge of a tree leads to a tree of bigger or equal level. For that purpose, a tree delays selection of the best edge, until all trees to whom best edge might lead, have bigger or equal level. Roughly speaking, this means that a tree stays completely idle, without even attempting to find the best edge, until some neighboring tree has smaller level. Notice that this delay does not introduce deadlocks, since trees are waiting only for trees of smaller level. The tree inherits the level of the tree on the other side of the best edge. It follows that each time a node participates in election of the best edge or any pointer manipulation, its level increases by 1, i.e. those operations can be done at most $\log_2 V$ times. Thus, communication complexity is $O(E+V\log V)$ and time complexity is $V\log V$. The reason why time complexity is not linear is that there might be a long chain of sub-trees of level l, each sub-tree hooked onto the next sub-tree on the chain, resulting in a tree of level $l+1$, regardless of the length of the chain. In particular, a tree of level 1 with $\frac{V}{2}$ nodes may be created. Such a tree may undergo $\log V-1$ level changes, each requiring $\Omega(V)$ time.

Chin, Ting [CT-85] and Gafni [G-85] addressed this problem by updating the level to the logarithm of the cardinality of the tree, each time that computation of the best edge is performed. Thus, even if level was too low during the process of selecting the best edge, the situation is corrected before the next selection. It seems that with this idea the resulting algorithm should be linear in time. However, the time complexity is $\Theta(V\log^* V)$. The reason for the

$\log^* V$ factor is that updating the level of a long chain comes too late. Indeed, consider a long chain consisting of 2^{10} sub-trees of level 10. Eventually, the level of the resulting tree will be set to (at least) 20; but this level update will take time linear in the size of the tree, i.e. $\Omega(2^{20})$. During this period of time, a tree T_{11} of level 11 neighboring with a tree T_{10} of level 10, belonging to that chain, is idle, since it waits until level of T_{10} reaches 11 (at least). Once level update in the chain terminates, T_{11} can proceed looking for its best edge. From the point of view of the search for the best edge at T_{11}, creation of the chain of 2^{10} trees of level 10 caused loss of $O(2^{20})$ units of time. The only possible way that T_{11} can reimburse itself for this loss is by hooking itself on a node in that chain, sub-sequently inheriting level 20. Indeed, in this case T_{11} waited $O(2^{20})$ time, but at least it got a reward: an impressive level increase from 11 to 20 ! Unfortunately, since T_{11} is committed to hook thru its *minimum-weight* edge, it is most likely that it will not be able to benefit from this opportunity.

3.2. Our MST algorithm

The observation above suggests that waiving the minimum weight property can help to achieve a linear time algorithm. Namely, instead of hooking itself on its minimum weight edge, each tree will hook itself on edge leading to the neighboring tree of maximum level. Thus, if the tree waits for a long time, it will be rewarded properly. This is the main idea behind the Counting stage of our algorithm. The MST stage assumes knowledge of V, the total number of nodes, which is provided by the previous Counting stage. It has a lot of similarity with [GHS-83], [G-85], [CT-85]. The only difference is that level increases are originated by many nodes, not only by the root node.

The MST stage is performed in two phases. The first phase runs an algorithm *identical* to [GHS-83], and terminates when all trees reach the size of $\Omega(\frac{V}{\log V})$. In this phase, we even do not bother to use the trick of [G-85,CT-85] since all trees are relatively small. The new algorithmic ideas are introduced in the second phase. We update the levels in a very accurate fashion, which prevents small trees waiting for big trees and speeds up the algorithm. This more aggressive method of level updates requires more messages but does not increase the communication complexity, since the number of trees is small, at most $\log V$. The *only* reason why the Counting stage is needed is that without knowing what V is, one does not know when does the first phase of the MST stage terminate. The innovation here is that instead of updating level by counting the total number of nodes, as in [CT-85,G-85], the level is updated *locally* based on estimate of number of nodes in a sub-tree. We introduce two new (compared to [GHS-83]) mechanisms for level update, called *Root-Update* and *Test-Distance*. Roughly speaking, the *Root-Update* increases the leve' of the root if within certain time it failed to find the best edge; *Test-Distance* increases level of nodes in a sub-tree whose root is far enough from the root of the resulting tree. The main goal is to guarantee that the period of time in which l is the smallest level in the network is upper-bounded by $O(2^l)$.

Let us now describe the algorithm with some more details. Whenever a node r becomes root of a tree T with level l, it broadcasts an initialization message, containing r, l as parameters, over T. This message is further relayed into trees which hook themselves onto T. Upon receipt of this initialization message, a node j remembers those parameters in local fields $Level_j$, $Root_j$, and starts execution of a local search procedure. This procedure either finds the minimum weight edge outgoing from j to a node i outside of T (with $Root_i \neq Root_j$) or declares that such node does not exists, i.e. all edges incident to j are internal edges in T.

Towards that goal, node j scans its incident edges, not yet known to be internal edges, in the decreasing order of their weights. Scanning an edge consists of sending a special test message to the node k on the other side of the edge and getting the reply from k. That reply is negative if $Root_k = r$ (i.e. k is in T) and is positive otherwise. The crucial property of the algorithm is that only nodes with level bigger or equal to l reply immediately to such message. A node k with $Level_k < l$ delays response to that message until $Level_k$ reaches l. If this level increase at k is due to the hooking of k's tree onto T, then k will have $Root_k = r$; thus the reply is negative and the edge to k is marked at j as "internal". In this case, the search procedure is resumed, namely next edge (after the edge (j,k)) is scanned. Otherwise, if reply is positive, then the edge (j,k) is declared to be the local candidate for the best edge of T.

The names of the local candidates are collected at the root. If none of the tree nodes could find a local candidate, then the algorithm terminates, since the tree spans the whole network and thus is the MST. Otherwise, the root chooses the one with the minimum weight, say edge (v,w) with v being the internal endpoint, as the best edge of T. Next, a special message travels to the internal endpoint v of that edge, reversing all father pointers on its way and transforming v into a root of T. When v receives that message, it acts as follows. If the edge (v,w) is a core edge and v is its biggest endpoint, v does not hook itself, since it is the root of the resulting tree; its level is increased by 1, and it broadcasts its own initialization message over the resulting tree.

Otherwise, v hooks itself on the other endpoint w of that edge. Observe that T became a sub-tree in a bigger tree, and v is the root of that sub-tree. If w has already received an initialization message which was broadcast by the root of the resulting tree, then this message is relayed by v over T, so that T will participate in the process of election of the best edge in the resulting tree.

Until this happens, v iterates the *Test-Distance* procedure. The main property of *Test-Distance* is that if it *succeeds*, then the size of the resulting tree is bigger than $2^{l(v)+1}$, where $l(v)$ is level of node v. In this case, the level of all the nodes in T, including v, is increased by 1, and *Test-Distance* procedure is executed again.

Upon each invocation of *Test-Distance*, node v sends a special *exploration* token to its father w. The token carries a counter, which is initialized to $2^{l(v)+1}$. Upon arrival of a token at a certain node, that node subtracts the number of its sons from the counter. If counter is positive, and the receiving node is *not* a root node, then that node forwards the token (with decreased counter) to its father. Eventually, after moving along pointers to fathers, either the token reaches the root with positive counter, or the counter of the token becomes non-positive, before reaching the root. In the former case, token "dies" and the *Test-Distance* procedure *fails*; in the latter case an acknowledgement is sent back to v. Upon its arrival at v, node v broadcasts a special message over T, which causes every node in T to increase by its level by 1. Upon termination of this broadcast, *Test-Distance* procedure is declared to be a *success*, and it is restarted again.

The process continues until eventually *Test-Distance* fails, i.e. the token of v reaches the root and dies. Clearly, at that time v is within distance of at most $2^{l(v)+1}$ from the root. Indeed, since every node which receives the token has degree at least 1, the token can travel for a length of at most $2^{l(v)+1}$ towards the root.

Consider now the process of selection of the best edge in a tree of level m. The *Root-Update* procedure is activated either when initialization message has advanced for distance bigger than 2^{m+1} or if some node detected more than 2^{m+1} internal edges while testing its edges in search for the local candidate. In either case, the process of selection of best edge is interrupted, the level of the root is increased by 1, new level is broadcast over the tree and new selection process is started.

Theorem W1: The MST stage eventually terminates. Upon its termination the father pointers form the MST of the network.

Proof: Will be given in full paper. •

3.3. Complexity of the MST stage

3.3.1. Communication

Theorem W2: The communication complexity of the MST stage is $O(E+V\log V)$.

Proof: Given below.

First Phase: The fact that number of message in the first phase is $O(E+V\log V)$ follows from the analysis of [GHS-83].

Second Phase: Compared to [GHS-83], the only additional messages are the messages sent by the *Root-Update* and the *Test-Distance* procedures. Clearly, operation of *Root-Update* can be charged to *(node,level)* pairs with constant charge per pair. It only remains to account for the exploration messages (and acknowledgements) in the *Test-Distance* procedure.

Let the *final* exploration token of a sub-tree be the exploration token which reaches the root. The *final* exploration message is a transmission of the final exploration token. The total number of non-final exploration messages cannot exceed by more than a constant the total number of transmissions of final exploration messages. Indeed, each exploration token traverses path which is twice longer than that of its predecessor; the lengthes of those pathes form a geometrical progression, whose sum is dominated by its last term. It is thus sufficient to show that the number of final exploration messages is $O(V\log V)$. Clearly, a node receives at most one final exploration message from each sub-tree at Phase 2.

Consider now a directed tree, referred to as the *merging tree*, whose vertices are trees existing during Phase 2. The leaves of the merging tree are the initial trees of Phase 2. The parent of a vertex representing tree T is the vertex representing the tree which resulted from merging T with some other trees. Observe now that if a network node i is traversed by exploration tokens of a tree A, then it *cannot* be traversed by tokens of any other tree C, which is an ancestor of A in the merging tree, since any ancestor C of A contains node i.

Thus, the total number of final exploration messages received by a node is upper-bounded by the maximal cardinality of a sub-set of vertices in the merging tree, such that no vertex in the sub-set is ancestor of another. Clearly, the largest possible sub-set consists of all the the leaves in the merging tree, corresponding to initial trees of Phase 2. Since in the second phase every tree has size $\frac{V}{\log V}$ at least, and initial trees of Phase 2 are node-disjoint, then the number of such trees is at most $\log V$. This implies that a node can receive at most $\log V$ final exploration tokens. Thus, the total number of exploration messages is $O(V \log V)$. This completes the Proof. •

3.3.2. Time

Theorem W3: The time complexity of the MST stage is $O(V)$.

Proof: Given below.

First Phase: Let τ_l be the length of of the interval of time in which level l is the lowest level in the network and S_l be the size of the biggest tree of level l. As observed in [G-85], $S_l \geq c \cdot \tau_l$, where c is a constant independent of l. Let α be the set of all l with $S_l < \frac{V}{\log V}$ and β be the set of all l with $S_l \geq \frac{V}{\log V}$. Observe that the time complexity of the first phase is

$$\sum_{l \in \alpha} \tau_l + \sum_{l \in \beta} \tau_l \leq \frac{1}{c} \cdot \left(\sum_{l \in \alpha} S_l + \sum_{l \in \beta} S_l \right)$$

For any two different levels in β, the biggest trees with those levels must be node-disjoint, since after size of $\frac{V}{\log V}$ is reached, the first phase is over for such tree. Thus, the time of the first phase is $O(V)$, because

$$\sum_{l \in \beta} S_l \leq V \quad \text{and} \quad \sum_{l \in \alpha} S_l \leq \sum_{l=0}^{\frac{\log V}{\log\log V}} \frac{V}{\log V} \leq V.$$

Second Phase: Denote by A_l the first time at which l is the lowest level in the network, and by C_l the first time after A_l at which every node at distance less than $2 \cdot 2^{l+1}$ from the root has level $l+1$ at least.

Claim 1: $C_l - A_l = O(2^{l+1})$.

Proof: After time A_l, consider a tree with level l. Since this level is the minimum level in the network, all test messages sent by nodes in that tree are answered immediately. It is easy to see that either the time required to complete the selection of the best edge is only $O(2^{l+1})$ time, or level of the root is increased to $l+1$ by the *Root-Update*. If the best edge is selected, and it is not the core edge, then the root ceases to be the root. If it is the core edge, then the root stays a root but its level is increased to $l+1$. In either case, by the time $A_l + O(2^{l+1})$, all tree roots have level $l+1$ at least.

It takes additional $O(2^{l+1})$ time to propagate level increase from a root to all the nodes whose distance from their roots does not exceed $2 \cdot 2^{l+1}$. •

Claim 2: $A_{l+1} - C_l = O(2^{l+1})$.

Proof: Consider a node n in a sub-tree T of depth D, $2^d \leq D < 2^{d+1}$, which has level s, $s \geq d$ when it becomes a sub-tree. Let H be the distance of the root of T to the root r of the whole tree.

If $s \geq l+1$, then level of n is already $l+1$ at least. If $s < l+1$ and $H < 2^{l+1}$, then node n is within distance of $D + H \leq 2^s + 2^{l+1} \leq 2 \cdot 2^{l+1}$ from r and its level is $l+1$ at least by the time C_l.

It only remains to consider the case of $s < l+1$ and $H > 2^{l+1}$. Since $s < l+1$, T must become a sub-tree before time C_l. Afterwards, each *Test-Distance* Procedure of level q, $s < q \leq l+1$, terminates successfully in time $O(2^{q+1})$ after its invocation, raising the level of all the nodes in the tree T to q. The total time consumed by all those *Test-Distance* procedures is at most

$$\sum_{q=s+1}^{l+1} O(2^{q+1}) = O(2^{l+1}).$$

Thus, by the time $C_l + O(2^{l+1})$, all nodes have level $l+1$ at least. •

Claims 1,2 imply that $\tau_l = A_{l+1} - A_l = O(2^{l+1})$, i.e. the period of time in which l is the smallest level in the network is upper-bounded by $O(2^l)$. The time complexity of the second phase is upper-bounded by

$$\sum_{l=0}^{\log V} \tau_l \leq \sum_{l=0}^{\log V} O(2^l) = O(2^{\log V}) = O(V). \bullet$$

4. Stage 1: Counting algorithm

4.1. Outline

The algorithm uses ideas of [GHS-83], [G-85] and [CT-85], but is substantially different from them.

The main difference is that it does not insist of constructing a *minimum* spanning tree of the network. It does construct a spanning tree, but not necessarily the one of minimum cost. It uses the idea of *levels*, introduced in [GHS-83], and attempts to adjust levels to the actual size of the tree, as suggested in [G-85] and [CT-85]. However, the level adjustment technique is much more subtle. Essentially, level is adjusted according to *height* and *degree* of nodes in the tree, as well as total *number* of the nodes in the tree; this gives a good estimate on the the time spent on processing of that tree. Since we waived the requirement that the tree should be of minimum cost, trees will almost never wait to other trees. Now us proceed with a more detailed description.

The algorithm maintains in the network a forest of directed rooted trees, which span all the nodes of the network. A link which enters the forest stays in the forest forever. Each tree is kept in a distributed fashion, i.e. node only keeps pointers to its father and its sons in the tree. The root of each tree is called *leader* of that tree. Initially, every node forms a degenerate tree consisting of one node. Upon the termination of the algorithm, there is only one tree; its root is the leader of the network.

Each tree has a level, which supposedly reflects the size of the tree. The level of a tree containing a single node is 0. At any time level of a tree is a lower bound on the logarithm of the number of nodes in the tree. In the course of the algorithm, trees are expanding and clash with neighboring trees. In such clashs, bigger level trees are allowed to invade territory of smaller-level trees, capturing their nodes. Captured nodes of the smaller-level tree inherit the level of the bigger tree and the name of the leader of the bigger tree, updating their father-son pointers accordingly. A tree can be invaded at the same time by many bigger-level trees from different directions, each capturing another piece of its territory.

Schematically, the algorithm performed by each tree at each level can be be viewed as consecutive applications of 3 basic procedures, *Link-Search, Level-Update and Marriage. Level-Update* Procedure updates the level of the tree resulting from merging tree, setting it to the logarithm of its cardinality. Level-Update succeeds in its task, unless the tree in which it is running is being invaded at the same time by some other tree. In case that it does succeed, Link-Search is called. In *Link-Search* Procedure, the tree expands, conquering neighboring trees of smaller level until either a tree of the same or bigger level is found. If no neighboring trees of the same level have been found, then the algorithm is finished. Otherwise, minimum-weight link leading to another tree of the same level is found, and the following *Marriage* Procedure merges together pairs of trees of the same level which have the same minimum-weight outgoing link. If the Marriage procedure cannot match the tree to another tree, then the algorithm is terminated.

This loop is repeated by each tree which has not yet been conquered. It is not hard to see that all the procedures executed by all the nodes were executed in some serial order, then the algorithm above is correct.

4.2. Detailed description of the algorithm

Upon the invocation of the algorithm and after each level increase, *Link-Search* procedure is called. During the procedure, each node scans its incident links, in the order of their weights, starting with the link of the smallest weight, until it finds a link leading to another tree of bigger or equal level; such link is called *feasible* link of that node at that level. Links already known to be internal links in the tree are not scanned any more. If while this scanning a node detects a link leading to a tree with smaller level, then the bigger tree starts invasion of the smaller tree thru that link, while at the same time the node in the bigger tree continues search for the feasible link, attempting to find a link to another tree of bigger or equal level.

The procedure is *interrupted* whenever a node is detected such that the sum of its height in the tree and its degree in the tree exceeds 2^{k+1}, where k is the level of the tree. In this case Level-Update Procedure is called, since, obviously, the tree contains much more nodes than ought to be in a tree of its level, and thus the level should be increased. The idea behind this interrupt mechanism is that Link-Search is interrupted whenever the time it spends is too big. This time is measured implicitly by the communication depth of Link-Search which is bounded by the maximum sum of node height and node degree in the tree, taken over nodes which participated in this procedure.

Level-Update Procedure attempts to update the level of the tree to the (integer) value of the logarithm of the number of nodes (cardinality) of the tree. Level-Update procedure succeeds in case that the tree is not being absorbed at that time by bigger-level tree and aborts otherwise. In the latter case, the level is not changed. The Level-Update procedure operates similarly to "two-phase commit" protocols. It first attempts to *lock* the nodes of the tree; it succeeds to lock the nodes which have not been captured in the meantime by bigger trees. A locked node cannot be captured by other tree until the lock is released. If all the nodes of the tree have been locked, then the procedure succeeds as a whole; then level of each node is set to the (integer) value of the logarithm of the cardinality of the tree. Otherwise, the procedure aborts without changing a level of any node. In both cases, all the nodes are unlocked upon the termination of the procedure. In case that Level-Update aborts, the leader of this tree becomes *inactive*, in the sense that it will not trigger execution of any additional procedure in its tree. The reason for it is that it realizes that it will never become the network leader and its tree will be absorbed by bigger-level trees. Observe that upon termination of Level-Update, either the level is increased or the tree becomes inactive.

Eventually, either there will be a non-interrupted execution of Link-Search, or the tree is invaded by another tree. In the latter case, the tree leader is killed. In the former case, it acts as follows. If none of the tree nodes found a feasible link, then the tree must cover the entire network. In this case, the algorithm terminates, the tree is a final spanning tree, the root of the tree is declared to be the leader, its name is broadcasted over the tree to all the nodes, and the total number of nodes is counted. Otherwise, some feasible links have been found. If *all* feasible links lead to trees of the same level, then the *preferred link* is elected as the feasible link with minimum weight; the tree on the other side of this link is called the *preferred tree*. Otherwise, if there exists a feasible link leading to a tree with bigger level, then the tree becomes inactive.

At this point, if the tree is active, then *Marriage* Procedure is called, which merges together pairs of trees of the same level, having the same preferred link. In such pair, the tree with bigger identity conquers its mate with smaller identity, in spite of the fact that their levels are the same.

4.3. Implementation details

In the Link-Search Procedure, scanning of the edges is implemented by sending *exploration* messages along these edges. After each node of the tree finishs its search of the feasible link, it reports the result of the search to the root of the tree. A leaf node sends the report whenever it finishes the search, while internal node does it only after receiving reports from all the sons. This well known communication pattern has been referred to as *convergecast* in [A-85], [AG-85]. This report either contains the identity of the feasible link and the level of the tree on the other side or simply says that no feasible link has been found, i.e. all incident links are internal links. The root node collects such reports from all nodes of its tree, including the nodes that have just been captured or are going to be captured.

The locking mechanism, used in the Level-Update Procedure, is implemented in 2 phases; each phase involves one broadcast and one convergecast over the tree. In the first broadcast, nodes are informed that the locking algorithm has started; a node receiving the first broadcast becomes locked if it has not yet been invaded by another tree. A locked node cannot be invaded by another tree until it is unlocked; the arriving exploration messages are buffered and processed immediately after the node will be unlocked. In the following convergecast, the leader of the tree finds out whether all locks has been obtained. If this is the case, then the locking succeeded; otherwise it fails. If the locking has been successful, then new level is computed. The following broadcast informs all the nodes whether the locking was successful. If locking was successful, then each node updates its level. In any case, it becomes unlocked and processes the exploration messages stored in its buffer while locking algorithm was running. The following convergecast is needed only for purpose of synchronization, i.e. to ensure that all nodes have completed this procedure.

4.4. Correctness

Theorem U1: The algorithm above eventually terminates, with father pointers forming a spanning tree in the network.

Proof (Sketch): Obviously, if the algorithm ever terminates, then it terminates correctly. A node may declare itself to be the leader of the network only if it is the root of a tree which spans the entire network. Clearly, such a node must be unique and all other nodes will know the name of the leader, since leader broadcasts its name along the tree after declaring itself as the leader. It only remains to show that the algorithm does terminate.

Since the trees can only grow, then, if the algorithm does not terminate, there must exist a "final" forest that consists of many trees, none of which will grow in the future. Consider among them the set S trees of the highest level. Link-Search procedures among these trees could not abort, since they have the highest level. Thus, every tree in S has a preferred link, leading to another tree in S. These preferred links are part of minimum weight spanning tree among all spanning tree which contain all the trees in S. Thus the trees in S together with preferred links cannot contain a simple cycle.

It follows that there exists a pair of trees in S whose preferred links lead to each other. Marriage procedure will merge such trees. This merging cannot be interrupted by higher-level trees since there are not any. This merging leads to a bigger tree with increased level, and contradicts the assumption that this is the final forest.

Details of the proof will be given in the final paper. •

4.5. Complexity of the Counting Stage

4.5.1. Communication complexity

Theorem U2: The total number of messages is $O(E+V\log_2 V)$.

Proof: We need to prove the following

Lemma: The maximum level is $\log_2 V$.

Proof: immediately follows from the fact that level is set to the logarithm base 2 of the number of nodes in the tree. •

Now, let us divide messages in two categories: *exploration* messages which are sent by nodes during the search for chosen link and *control* messages which include all the rest. Consider an exploration message sent from a node i to a node j. Let $Name(i)$ and $Level(i)$ denote name and level of node i upon transmission of that message, and let $Name(j)$ and $Level(j)$ denote name and level of node j upon arrival of that message. If $Name(i)=Name(j)$ then this is the last message to be sent over that link, and it will be charged to the link $(i \rightarrow j)$. Otherwise, the charging is as follows.

If $Level(i) > Level(j)$ then we charge the message to the pair $[Level(i), j]$; each pair is charged only once because upon receipt of that message node j immediately increases its level to $Level(i)$.

If $Level(i) \leq Level(j)$ then we charge the message to the pair $[Level(i), i]$; each pair is charged only once since this is the last exploration message sent by i at that level, i.e. level of i will increase before additional messages are sent. Since, the maximum possible level is $\log_2 V$, it follows that the total number of exploration messages is $O(E+V\log_2 V)$.

Now, we are going to prove that the total number of control messages is $O(V\log_2 V)$, and thus the exploration messages dominate the communication complexity of the whole algorithm. Control messages include the messages sent by Link-Search, Level-Update and Marriage Procedures. It is easy to see that messages of these procedures are sent only along tree links, and each procedure involves constant number of messages per each tree link. It only remains to show that at certain level, each procedure is called at most once. This is obviously true for Link-Search, which is called by tree leader (only) at times when level of the tree increases. Marriage procedure is also called once per level. Level-Update procedure is called once after each Link-Search Procedure and once in Marriage Procedure, i.e. at most twice per level. Thus, each of the procedures is indeed called only constant number of times. •

4.5.2. Time complexity

Theorem U3: The time complexity of the algorithm is $O(V)$.

Proof: Let time t_k be the last time that an active tree at level k exists in the network. A tree is defined to be active if its leader is active. We claim that $t_k = O(2^k)$. Since the the biggest level k that is achievable is $\log_2 V$, this claim establishes the statement of the Theorem. To prove the claim, all we need to show that $t_k - t_{k-1} \leq c \cdot 2^k$, since summing this inequality over all levels k yields $t_k \leq 2c \cdot 2^k = O(2^k)$.

Since the time it is created, a tree at level k executes, first, Link-Search Procedure, then Marriage Procedure and, finally, Level-Update Procedure. In case that Link-Search was not interrupted, Marriage Procedure is executed. Then, Level-Update Procedure is performed. Upon its completion, either tree becomes inactive or its level is increased to $k+1$.

Claim U3: Each one of the procedures above takes at most $O(2^k)$ time, if we we ignore the time spent in Link-Search Procedure, that is caused by waiting to nodes of lower level to become unlocked.

It is easy to see that the Claim implies that $t_k - t_{k-1} = O(2^k)$. Indeed, by time t_{k-1}, all processes of level k have already been created and all nodes, locked by Level-Update procedures of processes of levels less than k are already unlocked. Thus, after t_{k-1}, Link-Search procedures called by level k processes never wait until lower-level trees unlock their territory while invading it. According to the claim, all the procedures take $O(2^k)$ time if this waiting this ignored, proving the Claim.

To see that the Claim is true, observe that in the Link-Search messages never reach a node with height bigger than $O(2^k)$ and no node scans more than $O(2^k)$ incident edges. Thus, then execution of this procedure (after t_{k-1}) takes $O(2^k)$ time. Marriage procedure involves sending proposal to preferred tree of the same level, getting a reply. If there is no engagement, i.e. a tree makes a proposal and this proposal is turned down, then the tree becomes inactive. In case that proposal is accepted, the tree with bigger leader identity absorbs its preferred tree. Level-Update involves propagating a message over the resulting tree and getting acknowledgements back for constant number of times. All these actions take time proportional to the height of the trees involved. These heights cannot exceed $O(2^k)$ because of the interruption rule. Thus, overall, every one of the procedures above takes $O(2^k)$ time after t_{k-1}. ●

Acknowledgements

Oded Goldreich and Silvio Micali have collaborated with the author in the earlier stages of this research and contributed some of the ideas used in the Counting Algorithm. Oded Goldreich helped to prove the Completeness Theorem.

References

[A-85] B. Awerbuch, "Complexity of Network Synchronization", *Journal of the ACM,* Vol. 32, No. 4, October 1985, pp. 804-823.

[AD-76] P.A.Alsberg and J.D.Day, "A Principle for Resilient Sharing of Distributed Resources", *Proceeding 2nd Int. Conference on Software Engineering,* San-Francisco, October 1976.

[Afek-85] Y.Afek and E.Gafni, "Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks", *Proceedings of 1985 PODC Conference,* Minacki, Ontario, August 1985.

[AE-86] B. Awerbuch and S. Even, Reliable Broadcast in Unreliable Networks, to appear in *Networks.*

[AG-85] B.Awerbuch and R.Gallager, "Distributed Breadth-First-Search Algorithms", *IEEE Symposium on Foundations of Computer Science,* October 1985, Portland, Oregon.

[AGV-87] B.Awerbuch, O.Goldreich and R.Vainish, "On message complexity of broadcast: a basic lower bound", *unpublished manuscript,* January 1987.

[AM-85] B. Awerbuch and S.Micali, "Dynamic Deadlock Resolution Protocols with Bounded Complexities", unpublished manuscript, MIT, 1985.

[AM-86] B. Awerbuch and S.Micali, "Dynamic Deadlock Resolution Protocols", *Proceedings of 1986 FOCS Conference,* Toronto, Ontario, October 1986.

[AP-86] B.Awerbuch and S.Plotkin, "An $O(E+V\log^2 V)$ leader election in faulty networks", *unpublished manuscript,* MIT, December 1986.

[B-80] J.E.Burns, "A Formal Model for Message-Passing Systems", TR-91, Indiana Univ., Bloomington, May 1980.

[CT-85] F.Chin and H.F.Ting, "An almost linear time and $O(V\log V+E)$ messages Distributed Algorithm for Minimum Weight Spanning Trees", *Proceedings of 1985 FOCS Conference,* Portland, Oregon, October 1985.

[DM-78] Y.K. Dalal and R. Metcalfe, "Reserve Path Forwarding of Broadcast Packets", *CACM,* Vol. 21, No. 12, pp. 1040-1048, December 1978.

[FL-84] G. Frederickson and N.Lynch, "The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring", *Proceedings of the 16'th ACM Symposium on Theory of Computing,* Washington, D.C., April 1984.

[G-85] E.Gafni, "Improvements in time complexities of two message-optimal algorithms", *Proceedings of 1985 PODC Conference,* Minacki, Ontario, August 1985.

[GHS-83] R.G. Gallager, P.A. Humblet and P.M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees", ACM Trans. on Program. Lang. & Systems, Vol. 5, pp. 66-77, January 1983.

[K-78] P. Kanellakis, "An Election Problem in a Network", *6.854 term paper,* MIT, May 1978.

[KMZ-84] E.Korach, S.Moran and S.Zaks, "Tight Lower and Upper Bounds for some Distributed Algorithms for Complete Network of Processes", *Proceedings of 1985 PODC Conference,* Vancouver, BC, August 1984.

[MMP-78] D.A.Menasce, R.Muntz and J. Popek, "A Locking Protocol for Resource Coordination in Distributed Databases" *Proceedings of ACM SIGMOD,* June 1978.

[S-77] P.Spira, "Communication Complexity of distributed minimum spanning tree algorithms", *2nd Berkeley Conference on Distributed Data Management and Computer Networks,* Berkeley, California, June 1977.