A Sub-Linear Time Distributed Algorithm for Minimum-Weight Spanning Trees
(Extended Abstract)

Juan A. Garay * Shay Kutten † David Peleg ‡

Abstract: This paper considers the question of identifying the parameters governing the behavior of fundamental *global* network problems. Many papers on distributed network algorithms consider the task of optimizing the running time successful when an $O(n)$ bound is achieved on an n-vertex network. We propose that a more sensitive parameter is the network's diameter *Diam*. This is demonstrated in the paper by providing a distributed Minimum-weight Spanning Tree algorithm whose time complexity is sub-linear in n, but linear in *Diam* (specifically, $O(Diam + n^{0.614})$). Our result is achieved through the application of graph decomposition and edge elimination techniques that may be of independent interest.

1 Introduction

1.1 Motivation

In many papers on distributed network algorithms, the task of optimizing the running time is considered successful when an $O(n)$ bound is achieved on an n-vertex network. Typically, the justification is that there exist n-vertex graphs for which this bound is the best possible. The sequence of solutions to the Leader Election problem (LE) exemplify the reasoning above. Following the $O(n \log n)$ running time, and a first improvement by Gafni [G] (with an $O(n \log^* n)$ running time), Awerbuch gave an "optimal" $O(n)$-time solution to the problem [Aw]. This solution is optimal in the sense that there exist networks for which this is the best possible.

This type of optimality may be thought of as "existential" optimality. Namely, there are points in the class of input instances under consideration, for which the algorithm is optimal. A stronger type of optimality—which we may analogously call "universal" optimality—is when the proposed algorithm solves the problem optimally on *every* instance. An interesting "side effect" of universal optimality is that a universally-optimal algorithm precisely identifies the parameters of the problem that are inherently responsible for its complexity. For example, returning to the LE problem, a more careful look reveals that the inherent parameter is the network's diameter *Diam*. Indeed, it was observed in [P] that it is possible to give a trivial $O(Diam)$-time distributed LE algorithm (although it should be noted that the above-mentioned solutions were also message-optimal, whereas the algorithm of [P] is not). Indeed, in real large area networks $Diam << n$—for example, diameter of 6 versus a thousand nodes. (This is also true for other interesting cases, e.g., random graphs.)

The interesting question that arises is therefore whether it is possible to identify the inherent graph parameters associated with the distributed complexity of various fundamental network problems, and develop universally optimal algorithms for them. A closely related question has been dealt with before in the context of studying the role of *locality* in distributed computing. Various problems were shown to be essentially local, and hence amenable to a localized algorithm with very fast (e.g., polylogarithmic) running times. Notable examples include MIS and coloring [GPS, L, AGLP, PS]. Locality-based techniques were developed for reducing communication and time complexities also for other problems, the local nature of some of them is less apparent [AP1, AP2, AKP, NS93].

In contrast, we are interested here in problems that are essentially *global*, i.e., ones that do not admit localized solutions, but rather always require the algorithm to "traverse" the network. Problems of this type still raise the interesting (and practically important) question of deciding whether $\Omega(n)$ time is essential, or whether the network's *diameter* is the inherent param-

*I.B.M. T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598, USA. garay@watson.ibm.com. Part of the work was done while visiting the Weizmann Institute of Science, Rehovot, Israel.

†I.B.M. T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598, USA. kutten@watson.ibm.com.

‡Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, 76100 Israel. peleg@wisdom.weizmann.ac.il. Supported in part by a Walter and Elise Haas Career Development Award and by a grant from the Israeli Basic Research Foundation.

0272-5428/93 $03.00 © 1993 IEEE

eter. In the latter case, it would be desirable to devise algorithms for these network problems that have a better complexity for the case of graphs with low diameter.

In this paper we tackle the classical Minimum-weight Spanning Tree (MST) problem. This problem has been studied before as a canonical example for a graph-algorithmic problem whose communication-efficient distributed solution poses some surprisingly nontrivial subtleties [GHS]. The time complexity of the algorithm of [GHS] is $O(n \log n)$, which was later improved to the "optimal" $O(n)$ in [Aw]. As with other problems, such as the above LE example, it is natural to ask whether $O(n)$ is universally optimal, or it can be improved.

Once again, the MST problem proves to be a worthy candidate for this type of study. In other tree constructions, such as the Breadth-First-Search (BFS) tree (which is closely related to the LE problem), it is intuitively clear that the true time bound should be related to the network's diameter *Diam*, since the depth of the constructed tree is proportional to *Diam*, and thus it is possible to communicate on the tree while constructing it. In contrast, the MST of a given network may be considerably deeper than *Diam*, and in fact, may be as high as $\Omega(n)$. Hence construction methods based on communication on the tree structure itself are doomed to require $\Omega(n)$ time, and the problem of breaking the $\Omega(n)$ barrier seems intrinsically harder.

In this paper we get closer to identifying the inherent parameters governing the behavior of distributed MST construction, by presenting a distributed MST algorithm whose time complexity is sub-linear in n, and linear in *Diam* (specifically, $O(Diam + n^{0.614})$), thus breaking the $O(n)$ barrier. This result is achieved through the application of graph decomposition and edge elimination techniques that may be interesting in their own right.

1.2 Model and Definitions

In this paper we focus on the problem of devising a time-efficient distributed MST algorithm. The statement of the problem is as follows. We are given an undirected graph $G = (V, E)$, with a weight function $\omega : E \rightarrow R^+$ on the edges, such that each node in V is associated with its own processor, and processors are able to communicate with each other via the edges in E. The goal is to have the nodes (processors) cooperate to construct a tree covering the nodes in V whose total edge weight is no greater than any other spanning tree for G.

We assume that nodes have unique identifiers, and that each edge $e \in E$ is associated with a distinct weight $\omega(e)$, known to the adjacent nodes. The usefulness of having distinct edge weights stems from the fact that this property guarantees that the MST is unique. As pointed out in [GHS], having distinct weights is not an essential requirement, since one can always "create" them by appending to them the adjacent node's numbers. However, it is known that if the graph has neither distinct edge weights nor distinct node identifiers, then no deterministic distributed algorithm exists for computing an MST with a bounded number of messages [An, GHS].

For every subgraph F of the network, let $Diam(F)$ denote the diameter of F, i.e., the maximum distance on F between any two vertices of F, where distance is measured in the unweighted sense, i.e., in number of hops. In order to be able to concentrate on the central issue of time complexity, we shall follow the common trend of stripping away inessential complications. In particular, we ignore the communication cost of our algorithm (i.e., the number of messages it uses). We also assume that the computation performed by the network is *synchronous*. (This assumption is not essential, since our decision to ignore communication costs allows us to freely use a synchronizer of our choice.)

Still, we shall not adopt the extreme model employed in previous studies of locality issues [GPS, L], in which messages of arbitrary size are allowed to be transmitted in a single time unit, since in this model, the refined distinctions we focus on here disappear. Clearly, if unbounded-size messages are allowed, then the problem can be trivially solved in time $O(Diam(G))$ by collecting the entire graph's topology into a central node, computing an MST locally and broadcasting the result throughout the network.

Consequently, we will assume the more realistic model (and one that is more common in evaluation of distributed algorithms) in which messages have size $O(\log n)$, and a node may send at most one message on each edge at each time unit. We will also make the assumption that edge weights are polynomial in n, so an edge weight can be sent in a single message. (This assumption is required for the time analysis of all previous algorithms as well [GHS, Aw].)

1.3 Our Results

The original distributed MST algorithm of Gallager *et al.* [GHS] operates by growing so-called *fragments* in a distributed manner, with each fragment containing a portion of the final MST. This algorithm has a time complexity of $O(n \log n)$. This was later improved by Awerbuch, who gave an $O(n)$ time algorithm [Aw]. In this paper, we are able to establish the following:

```
Algorithm Sublinear-MST;

Part  I: Controlled-GHS;
Part II: Cycle elimination;
Part III: Global edge elimination;
```

Figure 1: Structure of the Sub-Linear Distributed MST Algorithm.

Theorem 1.1 There exists a distributed Minimum-weight Spanning Tree algorithm with time complexity $O(n^{0.614} + Diam(G))$.

Our algorithm consists of three parts, depicted in Figure 1. As its name indicates, `Controlled-GHS` is a modified variant of the original algorithm of [GHS]. The purpose of the modification is to produce a balanced outcome in terms of number and diameter of the resulting fragments. This is achieved by computing in each phase a small dominating set on the fragment forest, and merging fragments accordingly. This, in turn, is achieved by invoking the distributed minimal independent set (MIS) algorithm of [PS]. At the end of this phase, we are left with a "small" number of fragments which all have "small" diameter.

Part II uses pipelining techniques to gather information in each fragment center, and uses this information to perform the elimination of some of possible multiple inter-fragment edges in the fragment graph. This leaves us with a sparse set of inter-fragment edges from which the final tree is to be selected.

Finally, Part III performs the global elimination of most of the remaining edges, leaving only a tree connecting the fragments, and thus yielding the final MST. This elimination process is carried out on a superimposed breadth-first search structure; this yields the final MST tree. The details of each part are given in the remainder of the paper, followed by the analysis of the total complexity.

There are two methods of computing an MST: (1) combining fragments (i.e., adding edges until the tree is constructed); and (2) eliminating cycles (i.e., deleting edges until only the tree is left).

It seems that all Distributed algorithms typically use the first approach, thus it is of interest that parts II and III of our algorithm use the second approach. It is also of interest that our algorithm combines the two approaches, while even sequential algorithms use either one or the other.

2 Part I: Controlled GHS

In this section, we provide a modified, *controlled* version of the Gallager-Humblet-Spira algorithm for MST, that is suitable for our purposes. We first overview (briefly) the original algorithm of [GHS].

2.1 Brief Description of the Gallager-Humblet-Spira Algorithm

In the original distributed MST algorithm of Gallager *et al.* [GHS] (which, from now on, is referred to as `GHS` for convenience), nodes form themselves into *fragments* of increasing size. Initially, all nodes are in singleton fragments. Nodes in each fragment F are connected by edges that form an MST, $T(F)$, for the fragment.

Within each fragment F, nodes cooperate to find the minimum weight *outgoing* edge in the entire fragment (an outgoing edge of a fragment F is an edge with one endpoint in F and another at a node outside it). The strategy for identifying this edge involves broadcasting over the fragment's tree $T(F)$, asking each node separately for its own minimum weight outgoing edge. These edges are then sent upwards on the tree $T(F)$, towards the root. Each intermediate node first collects this information from all its children in the tree, and then passes up only the lowest-weight edge it had seen (which is therefore the lowest-weight edge in its subtree). The minimum weight outgoing edge is selected by the root to be included in the final MST.

Once a fragment's minimum weight outgoing edge is found, a message is sent out over that edge to the fragment on the other side. The two fragments may then combine, possibly along with several other fragments, into a new, larger fragment. The new fragment finds its own minimum weight outgoing edge, and the entire process is repeated until all the nodes in the graph have combined themselves into one single fragment. Each fragment (of size 2 or greater) is identified by the fragment's *core edge*, which is basically the edge along which the merging of the two previous fragments resulting in the current fragment took place. (This definition becomes more complicated when more than two fragments merge together in one step, but this is of no consequence to us here.)

A *level number* is associated with each fragment. If $level(F) = l$ for a given fragment F, then the number of nodes in F is greater than or equal to 2^l. Initially, all fragments (singleton nodes) are at level 0. When two fragments at level l are combined together, the resulting new fragment has level $l + 1$. Thus, the total number of messages is kept to $O(n \log n)$ (although some more complex rules are needed to allow merges between clusters of unequal levels). Similarly, it is not

hard to show by induction on the level numbers that the time complexity of the algorithm is $O(n \log n)$ time units.

We refer the reader to [GHS] for further details.

We remark that in the original **GHS** algorithm, all nodes operate asynchronously. This creates the risk of undesirable "growth patterns" of fragments, resulting in excessive communication costs (measured in number of messages), and thus increasing the time. We handle this problem by using special rules for merging fragments, designed to prevent these complications. These rules are based on a "balanced data structure" approach.

2.2 Computing a Small Dominating Set on a Tree

In this subsection we describe a procedure **Small-Dom-Set** for computing a small dominating set on a given tree. This procedure will be used later on in modifying algorithm **GHS** for our purposes. The procedure makes use of a sub-procedure for computing a maximal independent set in the tree. (A set M of vertices in a tree T is said to dominate the tree if every vertex outside M has a neighbor in M.) A distributed version of Procedure **Small-Dom-Set** will later be used as a component in our **Controlled-GHS** algorithm.

Our goal is as follows. Given a tree T with a vertex set $V(T)$, find a set of vertices $M \subseteq V(T)$ such that:

1. M dominates $V(T)$, and
2. $|M| \leq \frac{|V(T)|}{2}$.

Furthermore, we would like this procedure to be amenable to a fast distributed implementation.

The procedure is based on the following. For a vertex v in a tree T, let $\texttt{Child}(v)$ denote the set of v's children in T. We use a *depth* function $\check{L}(v)$ on the nodes, defined as follows:

$$\check{L}(v) = \begin{cases} 0, & \text{if } v \text{ is a leaf,} \\ 1 + \min_{u \in \texttt{Child}(v)}(\check{L}(u)), & \text{otherwise.} \end{cases}$$

We denote by $\check{L}_i$ the set of tree nodes at *depth* i,

$$\check{L}_i = \{v \mid \check{L}(v) = i\}.$$

Procedure **Small-Dom-Set** for computing the dominating set on a tree T is shown in Figure 2, and a pictorial example is given in Figures 3 and 4. (For the distributed implementation discussed in the next subsection, it is important to note that although the *depth* numbers $\check{L}(v)$ are defined for every vertex v in the tree, only the vertices belonging to the first three *depth*s, $\check{L}_0$, $\check{L}_1$ and $\check{L}_2$, are actually marked.)

Algorithm Small-Dom-Set;

Mark the nodes of T with *depth* numbers $\check{L}(v) = 0, 1, 2$.
Select an MIS, Q, in the set R of unmarked nodes;
$M := Q \cup \check{L}_1$;

Figure 2: Computing a small dominating set M on a tree T.

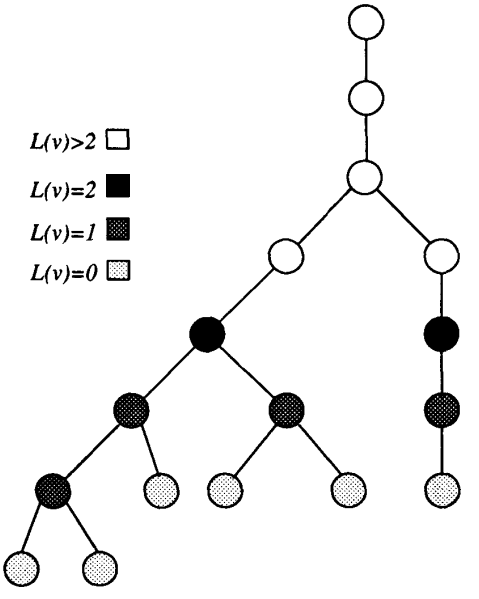

Figure 3: The *depth* numbers marked by the algorithm on a given tree T.

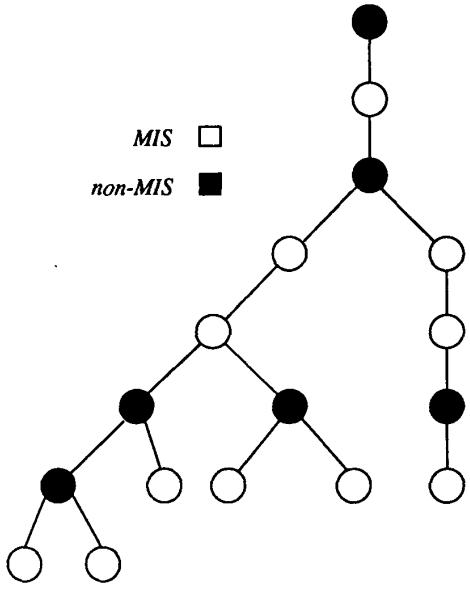

Figure 4: A small dominating set M on the tree T.

The fact that Procedure **Small-Dom-Set** produces a dominating set is established by the following lemma, whose proof is straightforward:

Lemma 2.1 Let M be the outcome of Procedure **Small-Dom-Set** on the tree T. Then for every node $v \notin M$ there exists an adjacent node $v' \in M$. ∎

The "smallness" of the resulting dominating set is guaranteed by the following lemma.

Lemma 2.2 $|M| \leq \frac{|V(T)|}{2}$.

Proof By construction, $M = \check{L}_1 \cup Q$. It is clear that

$$|\check{L}_1| \leq |\check{L}_0|. \quad (1)$$

Claim 2.3 $|Q| \leq \frac{|R \cup \check{L}_2|}{2}$

Proof For every $v \in Q$ we select a *distinct* match $\omega(v) \in (R \cup \check{L}_2) - Q$, hence $|R \cup \check{L}_2| \geq 2 \cdot |Q|$. The matching is done as follows: Pick $\omega(v)$ to be an arbitrary child of v (by definition, vertices in Q always have children in $R \cup \check{L}_2$). Distinctness is guaranteed by the fact that each node in the tree has a unique parent. ∎ (**Claim2.3**)

It follows from (1) that

$$|\check{L}_1| \leq \frac{|\check{L}_0 \cup \check{L}_1|}{2},$$

which, together with Claim 2.3, yields

$$|M| = |Q \cup \check{L}_1| \leq \frac{|R \cup \check{L}_2|}{2} + \frac{|\check{L}_0 \cup \check{L}_1|}{2} = \frac{|V(T)|}{2} \quad . \quad \blacksquare$$

2.3 Controlled-GHS

In this subsection, we provide a modified, *controlled* version of **GHS** (named **Controlled-GHS**) that is able to achieve the following:

1. Upon termination, the number of fragments is bounded above by N (for N to be specified later).
2. Throughout the execution, the diameter of every fragment F satisfies $Diam(F) \leq d$ (for d to be specified later),

Intuitively, since we focus on a synchronous algorithm, and we don't care about communication complexity, our version of **GHS** is simpler than the original algorithm. In particular, we are oblivious to balancing fragment sizes, and we do not need to use the level rules used in the original algorithm, since levels are imposed by phase synchronization. More specifically, **Controlled-GHS** also starts with singleton fragments, and executes a total of I phases. Each phase of **Controlled-GHS** consists of the following two stages.

Stage 1:

Execute a phase of **GHS** up to a point where each fragment F has chosen its minimum weight outgoing edge, i.e., has decided which other fragment in the current fragment collection it wants to merge with.

This decision induces a "forest" structure on the fragment collection (possibly with length-2 loops at the tree roots). Henceforth we refer to this structure as the *fragment forest*, denoted FF.

Stage 2:

Break the resulting trees into "small" ($O(1)$ depth) trees, and merge only these small trees.

This process is depicted in Figure 5. Stage 2 is accomplished by computing a dominating set $M(T)$ on each tree T of the fragment forest FF, and then letting each fragment $F \notin M$ pick one neighboring fragment $F' \in M$ and merge with it. This causes the actual merges performed in a phase of **Controlled-GHS** to have the form of "stars" in the fragment forest FF, and prevents merges along long chains, and hence bounds the diameter of the resulting fragments.

The dominating sets are computed using a distributed implementation of Procedure **Small-Dom-Set**, applied separately to each tree T in the fragment forest FF. A key aspect in this computation is the use of the distributed Minimal Independent Set (MIS) algorithm

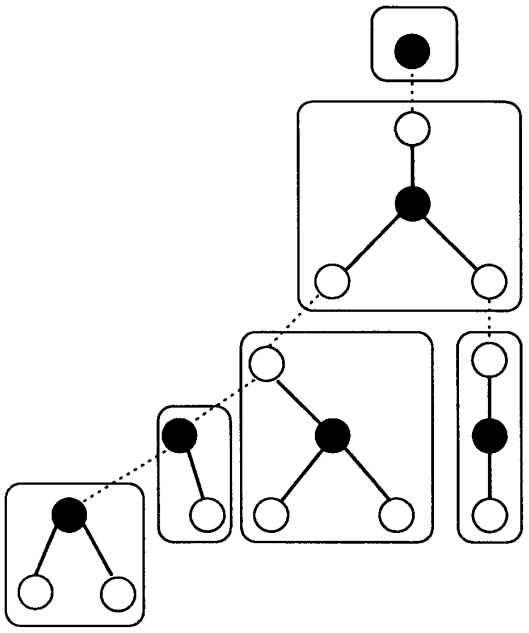

Figure 5: A Phase of Controlled-GHS: the tree T of Figure 4 is broken into "small" trees (represented by the solid edges) according to the MIS.

of Panconesi and Srinivasan [PS]. Note that although Procedure **Small-Dom-Set** is applied to the trees of the fragment forest FF, it is actually executed on the original network itself. Hence the procedure needs to be implemented by simulating each fragment by a single representative, say, its root. The (straightforward) details are omitted.

2.4 Analysis of Controlled-GHS

The bounds on the number and diameter of fragments are established by the next lemma.

Lemma 2.4 In each phase of Controlled-GHS,

1. the number of fragments at least halves, and
2. $Diam(F)$ increases by a factor of at most 3, for every fragment F.

Proof The number of new fragments in each tree T of the fragment forest at the end of a phase is equal to $|M(T)|$. By Lemma 2.2, $|M(T)| \leq |V(T)|/2$. Hence the same holds for the entire fragment forest, and Claim 1 of the lemma follows. Claim 2 is readily satisfied, since the merges are star-shaped. ∎

Corollary 2.5 After running Controlled-GHS for I phases,

1. the number of fragments is at most $N = \frac{n}{2^I}$, and
2. $Diam(F) \leq d = 3^I$, for every fragment F. ∎

Let us now turn to analyzing the time complexity of Controlled-GHS. Computing MIS (in a synchronous manner) using the algorithm of [PS] on a graph $G = (V, E)$, $|V| = n$ takes time $O(2^{\sqrt{\log n}})$ (which is less than $O(n^\epsilon)$ for any $\epsilon > 0$).

Examining the performance of Procedure **Small-Dom-Set**, we note that since the procedure is executed on the original network itself, it is slowed down by a factor of $O(Diam(F))$. It is easy to see that a leaf in the fragment forest identifies itself as such, and subsequently marks itself $\check{L}_0$, in time $O(Diam(F))$, and similarly, fragments mark themselves $\check{L}_1$ and $\check{L}_2$ in time $O(Diam(F))$ as well. The more expensive part is the MIS computation. The original time complexity of the algorithm of [PS] is slowed down to $O(2^{\sqrt{\log n}}) \cdot Diam(F)$ in our case.

Hence in each phase i, $1 \leq i \leq I$, Controlled-GHS takes time $\leq Diam(F) \cdot O(2^{\sqrt{\log n}}) \leq 3^i \cdot O(2^{\sqrt{\log n}})$. Thus, the total time is given by $\sum_{i \leq I}(3^i \cdot O(2^{\sqrt{\log n}})) = 3^I \cdot O(2^{\sqrt{\log n}})$.

The properties of the algorithm are summarized by the following lemma.

Lemma 2.6 When Algorithm Controlled-GHS is activated for I phases, it takes $O(3^I \cdot 2^{\sqrt{\log n}})$ time, and yields up to $N = n/2^I$ fragments, of diameter at most $d = 3^I$. ∎

3 Part II: Cycle Elimination on the Fragment Graph

Since the MST of a given network may be considerably deeper than $Diam(G)$ (possibly as high as $\Omega(n)$), an algorithm based on communicating on the tree structure itself is doomed to require $\Omega(n)$ time. Thus, improving the time bound of an MST construction requires a different approach. The idea is to deviate from the GHS algorithm at an appropriately chosen point, and switch to an algorithm based on eliminating edges that are known for sure not to belong to the MST.

3.1 Outline of the Short Cycle Elimination Procedure

Let $\tilde{F}$ denote the *fragment graph* that is the outcome of Part I. The vertices of this graph are the fragments

constructed in Part I, and its edges are all the inter-fragment edges, i.e., all edges of G whose endpoints belong to different fragments. In this graph, it is possible that cycles exist. For instance, it is even possible that multiple edges exist (from different nodes belonging to the same fragment) connecting any two fragments. Our elimination approach focuses on a special type of edges, defined as follows.

Definition 3.1 An edge e in a weighted graph G is said to be a *bottleneck* edge of G if there exists a cycle C in G, such that e is the heaviest edge in C. The cycle C is called the *critical* cycle of e.

Our procedure will rely on the following well-known lemma.

Lemma 3.2 Given a weighted graph $G = (V, E)$, if e is a bottleneck edge of G then $e \notin MST(G)$. ∎

This motivates the "short cycle elimination" procedure that we now describe. As before, let $T(F)$ denote fragment F's tree. We will distinguish one of the nodes adjacent to the fragment F's core edge (say, the node with the highest id) as the fragment's *center*, $r(F)$. Equivalenty, $r(F)$ is $T(F)$'s root. The goals of Part II of `Sublinear-MST` are, for each fragment $F \in \tilde{F}$:

1. to eliminate "short" cycles (up to a distance l to be specified later) going through the fragment; and
2. to concentrate, via $T(F)$, all the information pertaining to every other fragment up to distance l from F in $r(F)$, the fragment's center.

For clarity, we will first explain the procedure of eliminating cycles of length 2. Once the intuition is understood, the process of eliminating longer cycles will consist — some implementation details apart — of the repeated application of the same basic idea.

3.2 Cycles of Length 2

In the context of our fragment graph $\tilde{F}$, cycles of length 2 arise in $\tilde{F}$ when two nodes of a fragment are connected to two nodes (possibly the same) of another fragment. The goal is then to eliminate these multiple edges, i.e., to remain with only one edge (particularly, the one with lesser weight) connecting each pair of neighboring fragments. The method we use is as follows. Basically, the nodes of the fragment F execute a *convergecast* procedure concerning the fragments adjacent to F, i.e., they collect information on the edges connecting F to the adjacent fragments and send it upwards on the tree $T(F)$ to the center $r(F)$. In order to execute the procedure, each fragment node v creates the record $Path_1(F')$, for each $F' \in \tilde{F}$ adjacent to F. This record contains the following items:

1. an edge $e = (u, v)$, such that $v \in F$ and $u \in F'$;
2. the id's of u, v, F';
3. the weight $\omega(e)$.

The goal is to have at the center $r(F)$, upon termination of this process, the data structure

$DS_1(F) = \{$a record $Path_1(F') \mid F'$ adjacent to $F\}$.

The procedure can be summarized as follows:

- Initially, each node in F prepares a local list of records $Path_1(F')$, one record for each fragment F' that is adjacent to it.
- These $Path_1$ records are collected and shipped upwards on the tree $T(F)$ towards the root $r(F)$, while eliminating duplicities. That is, each node in $T(F)$ collects information about all fragments adjacent to nodes in its subtree, but it sends upwards only *one* record concerning each such fragment; if several exist, it eliminates all but the one with minimum edge weight $\omega(e)$.

This convergecast process must be carried in a controlled fashion, in order to ensure that all duplicities are eliminated. On the other hand, the upward shipping has to be pipelined in order to guarantee reasonable time complexity. This pipelining is guaranteed by adopting the following rules.

- The records are pipelined by fragment id. I.e., if $Id(F_1) < Id(F_2)$, then each vertex along the tree $T(F)$ receiving these two records sends $Path_1(F_1)$ before $Path_1(F_2)$.
- A leaf starts sending records to its parent at time 0, with the smallest id adjacent fragment, and continues in increasing order until exhausting its neighboring fragments.
- In general, an intermediate node starts sending at time $\hat{L}(v)$, where $\hat{L}(v)$ is the *depth* function defined as follows:

$$\hat{L}(v) = \begin{cases} 0, & v \text{ is a leaf,} \\ 1 + \max_{u \in \texttt{Child}(v)}(\hat{L}(u)), & \text{otherwise.} \end{cases}$$

- Node v stores all the records it receives from its children regarding the adjacent fragments. If the records it receives concern the same fragment, then it eliminates all but the one with lowest edge weight $\omega(e)$. At every step after time $\hat{L}(v)$, v sends up $Path_1(F')$ for the lowest id fragment F' it knows of until that point.

Upon termination, the center $r(F)$ is able to assemble data structure $DS_1(F)$, as defined above. Finally, $r(F)$ broadcasts $DS_1(F)$ on the tree $T(F)$ to all nodes in F.

Based on the structure $DS_1(F)$, each node $v \in F$ adjacent to an outgoing edge e can decide whether this edge is still a candidate for remaining in the final tree, or has already been eliminated. If so, it marks the edge e "unusable".

It is easy to verify the following basic properties of the above pipelining policy.

Lemma 3.3 Each node $v \in F$ sends to its parent exactly one record $Path_1(F')$ for each fragment F' that is adjacent to nodes in v's subtree in $T(F)$; these records are sent up in increasing order of fragment id.

Proof [sketch] The proof is based on showing that at time $j + i$, a node v on *depth* $\hat{L}(v) = j$ has already received from its descendents all the records concerning paths leading to the ith smallest-id neighboring fragments that are adjacent to vertices in the subtree of $T(F)$ rooted at v. This can be proved by double induction (on j and i). ∎

Let $\tilde{F}_1$ be the graph remaining of $\tilde{F}$ after the first iteration (omitting the edges marked "unusable").

Lemma 3.4 If F' is a fragment such that $dist_{\tilde{F}_1}(F, F') = 1$, then $\tilde{F}_1$ contains a unique edge connecting F to F', and $DS_1(F)$ contains the corresponding record $Path_1(F, F')$. ∎

Lemma 3.5 The graph $\tilde{F}_1$ contains no parallel edges. Moreover, for every two neighboring fragments F, F' in $\tilde{F}$, the unique remaining edge e in $\tilde{F}_1$ is the lightest-weight one. ∎

Lemma 3.2 now yields

Corollary 3.6 $E\tilde{F}_1$ contains an MST for the fragment graph $\tilde{F}$. ∎

3.3 Remaining Cycles

In the previous subsection, information is gathered concerning (minimum-weight) paths to neighboring (or, distance 1) fragments. We will be basically repeating this process for l phases, eliminating at phase i, $1 \leq i \leq l$, the inter-fragment cycles of length $2i - 1$ and $2i$, and keeping minimum-weight paths to distance i fragments. The elimination is performed by throwing away the path containing the edge with maximum weight in the detected cycles. However, the process is more complex than the one described before, and more careful reasoning and pipelining is called for.

The process is based on constructing a larger data structure DS_i, containing records $Path_i(F')$, for every fragment F' at distance i of less from F (where distance is measured in $\tilde{F}$). This time, however, once $DS_i(F)$ is determined at $r(F)$, it also identifies the last node in F on the path to each fragment F', and appoints it as the node "responsible" for that fragment.

In order to set the stage for the $(i + 1)$st iteration of the process, we do the following. The root $r(F)$ broadcasts DS_i on $T(F)$ as before. However, now every node $v \in F$ that is responsible for some fragments in the vicinity, sends $DS_i(F)$ over the external link to the other end. Since this is done in every fragment, nodes in F adjacent to neighboring fragments F' get the data structure $DS_i(F')$, and consequently learn of fragments at distance $i+1$ from F in $\tilde{F}$. This information can now be used by the nodes in charge to compile their initial suggestions for $Path_{i+1}(F')$ for all the new fragments F' they learn of.

The convergecast is again executed and, as in the first iteration, competing candidate paths are eliminated at any vertex along the way during the pipelining. The criterion for elimination is the following. When two paths (of length i), both leading from F to the same fragment F', compete with each other, we throw away the path that contains the edge with maximum weight in the detected cycle (composed of the two paths combined). Again, the heaviest edge is marked "unusable".

Example: Consider the fragments F and F' illustrated in Figure 6. These fragments are at distance 2 in the fragment graph $\tilde{F}$. This fact is discovered in the second phase of the algorithm by the nodes v_1 and v_2 in F, who identify a length-2 path to F'. The convergecast process on the tree $T(F)$ will recognize the fact that there are two paths, and that the edge (v_1, u_1) is the heaviest along the resulting length-4 cycle. Consequently, the path $F - F_1 - F'$ will be discarded, the sedge (v_1, u_1) will be marked "unusable", and the only path to be maintained in $DS_2(F)$ regarding the fragment F' will be $F - F_2 - F'$. The node responsible for this path in F is v_2. ∎

3.4 Analysis and Complexity of Part II

Correctness follows from the following claims. Let $\tilde{F}_i$ be the graph remaining of $\tilde{F}$ after the ith iteration (omitting the edges marked "unusable"). By induction on i, we prove the following.

Lemma 3.7 If F' is a fragment such that $dist_{\tilde{F}_i}(F, F') \leq i$, then $\tilde{F}_i$ contains a unique path connecting F to F', and $DS_i(F)$ contains the corresponding record $Path_i(F, F')$. ∎

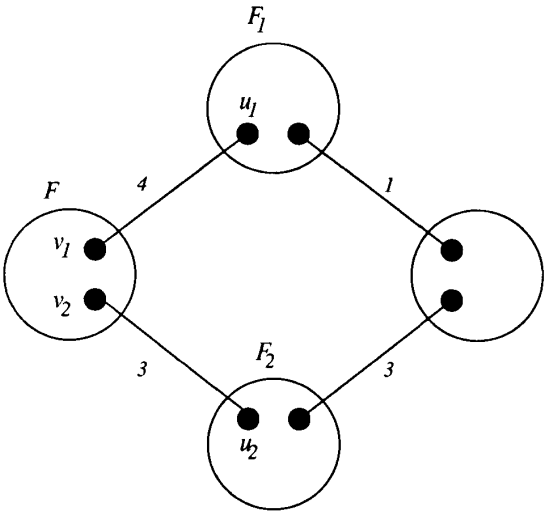

Figure 6: Cycle Elimination: paths from F to F' discovered on the second phase ($i = 2$). The number next to each edge denotes its weight.

Recall that the *girth* of a graph G, denoted $Girth(G)$, is number of edges in the smallest cycle in G. (A single edge is not considered a cycle of length 2, so $Girth(G) \geq 3$ for every G.)

Lemma 3.8 $Girth(\tilde{F}_i) > 2i$. Moreover, the set of edges $\{e \mid e \in \tilde{F}, \ e \notin \tilde{F}_i\}$ is identical to the set $\{e \mid e$ is the heaviest edge on some cycle of length $2i$ or less in $\tilde{F}\}$. ∎

Lemma 3.2 now yields

Corollary 3.9 $E(\tilde{F}_i)$ contains an MST for the fragment graph $\tilde{F}$. ∎

We now choose to fix $l = \log n$, hence we are able to eliminate all cycles in $\tilde{F}_l$ of length $2 \log n$. We then make use of the following known result in extremal graph theory, presented in [B], which provides near-tight bounds on the relationship between the girth of a graph and the number of edges it contains.

Lemma 3.10 [B] For every integer $r \geq 1$ and n-vertex, m-edge graph $G = (V, E)$ with $Girth(G) \geq r$, $m \leq n^{1+2/(r-2)} + n$. ∎

Combined with Lemma 3.8, we get that for $l = \log n$,

Corollary 3.11 $|E(\tilde{F}_l)| = O(N)$. ∎

Finally, let us discuss the complexity of performing the short cycle elimination procedure. Each iteration of the cycle elimination phase involves an exchange of DS_{i-1} over all external links, a convergecast of DS_i records and a broadcast of the final picture of DS_i from the root.

Observe that each record $Path_i(F)$ of DS_i consists of $O(i \log n)$ bits of information, so it can be sent in one time unit. Furthermore, even if each node in F is connected to *all* other existing fragments, there are no more than $\tilde{F}$ records for it to ship upwards, since it forwards at most one record per adjacent fragment. Since the convergecast is carried out in a fully pipelined fashion, the i'th iteration requires $O(Diam(F)+i|\tilde{F}|)$ time, and the entire phase II takes $O(Diam(F)+|\tilde{F}|\log^2 n)$ time. By Lemma 2.6, the total running time of this process is bounded as follows.

Lemma 3.12 The total running time of Part II (short cycle elimination) is $O(d + N \log^2 n) = O(3^l + \frac{n}{2^l} \cdot \log^2 n)$. ∎

4 Part III: Global Edge Elimination

We now proceed to reduce the total number of edges (to the necessary $N - 1$). Our method is as follows:

1. Build a *breath-first search* tree B on G, the original graph;
2. from every fragment's center $r(F)$, upcast the list of (uneliminated) external edges adjacent to F on B;
3. the final computation (elimination of edges) is performed centrally at B's root, who then broadcasts the resulting MST to all nodes, over the tree B.

Lemma 4.1 Part III's total running time is bounded by $O(\frac{n}{2^l} + Diam(G))$, and its output is an MST for G. ∎

Proof [sketch] The correctness of the output again follows from Lemma 3.2. Let us now estimate the complexity of this part of the algorithm. From the previous section, the total number of edges to be sent up to the root is bounded by $O(N)$. Also, since B is a BFS tree, its depth is $O(Diam(G))$. Thus, pipelined, all edges arrive at the root in time $O(N + Diam(G))$. Broadcasting from the root the final MST will take another $O(N + Diam(G))$ time. ∎

5 The Complexity of the Combined Algorithm

Summarizing the results of the last three sections, the complexities of all three parts of our algorithm are as follows, for the given parameter I specified for the first part:

Part I: $3^I \cdot O(2^{\sqrt{\log n}})$
Part II: $3^I + \frac{n}{2^I} \cdot \log^2 n$
Part III: $Diam(G) + \frac{n}{2^I}$

The total time complexity is thus optimized when choosing I such that $3^I = \frac{n}{2^I}$, namely, $I = \frac{\ln n}{\ln 6}$. For this choice of I, we get $3^I = \frac{n}{2^I} = n^\varepsilon$ for $\varepsilon = \frac{\ln 3}{\ln 6} = 0.6131..$, which yields a total time complexity of $O(Diam(G) + n^\varepsilon \cdot 2^{\sqrt{\log n}})$. Since $2^{\sqrt{\log n}} = o(n^{\varepsilon'})$ for any $\varepsilon' > 0$, we get a bound of $O(Diam(G) + n^{0.614})$ on the time complexity. Theorem 1.1 follows.

References

[An] D. Angluin, Local and Global Properties in networks of processors, *Proc. 12th Symp. on Theory of Computing*, pp. 82–93, 1980.

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

[AGLP] B. Awerbuch, A. Goldberg, M. Luby and S. Plotkin, Network decomposition and locality in distributed computation, *Proc. 30th Symp. on Foundations of Computer Science*, pp. 364–375, October 1989.

[AKP] B. Awerbuch, S. Kutten, and D. Peleg, Competitive Distributed Job Load Balancing, *Proc. 24th ACM Symp. on Theory of Computing*, 1992, 571–580.

[AP1] B. Awerbuch and D. Peleg, Network synchronization with polylogarithmic overhead, 31^{st} *IEEE Symp. on Foundations of Computer Science*, pages 514–522, October 1990.

[AP2] B. Awerbuch and D. Peleg, Concurrent online tracking of mobile users, *Proc. ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, Zurich, Sept. 1991, 221–233.

[B] B. Bollobás. *Extremal Graph Theory*. Academic Press, 1978.

[G] E. Gafni, Improvements in the time complexity of two message-optimal election algorithms, *Proc. 4th Symp. on Principles of Distributed Computing*, pp. 175–185, August 1985.

[GHS] R. Gallager, P. Humblet and P. Spira, A distributed algorithm for minimum-weight spanning trees, *ACM Transactions on Programming Languages and Systems*, Vol. 5 (1), pp. 66–77, January 1983.

[GPS] A. V. Goldberg, S. Plotkin, and G. Shannon, Parallel symmetry breaking in sparse graphs, *Proc. 19th ACM Symp. on Theory of Computing*, May 1987.

[L] N. Linial, Locality as an obstacle to distributed computing, *Proc. 27th IEEE Symp. on Foundations of Computer Science*, October 1987.

[NS93] M. Naor and L. Stockmeyer, What can be computed locally? *Proc. STOC 93.*

[P] D. Peleg, Time-optimal leader election in general networks, *Journal of Parallel and Distributed Computing*, Vol. 8, pp. 96–99, 1990.

[PS] A. Panconesi and A. Srinivasan, Improved distributed algorithms for coloring and network decomposition problems, *Proc. 33rd Symp. on Theory of Computing*, 1992, 581–592.