LIMITED DISTRIBUTION NOTICE
(NOT TO BE REMOVED FROM DOCUMENT):

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communication and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

A Faster Distributed MST Algorithm

Shay Kutten * David Peleg †

Abstract

We present a fast distributed algorithm for constructing a minimum weight spanning tree (MST). On an n-vertex network of diameter $Diam$, the new algorithm constructs an MST in time $O(n^{\frac{1}{2}+\epsilon} + Diam)$ for an arbitrarily small $\epsilon > 0$, improving on the results of [GKP].

The new algorithm is conceptually simpler than the previous one. It uses two new tools. The first is a fast distributed algorithm to compute a small k-dominating set D (for any fixed k) and its induced graph partition (breaking the graph into radius k clusters centered around the vertices of D). The second is a very simple convergecast protocol to inform a center about edges of the graph. Messages about edges that close cycles are not forwarded. This simple protocol is similar to the one used in the final phase of the algorithm of [GKP], and most of the novelty lies in a new careful analysis proving that the convergecast process is fully pipelined, and no waiting occurs at intermediate nodes. This enables the new algorithm to skip some of the intermediate phases employed in the previous one.

*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 Basic Research Foundation. Part of the work was done while visiting IBM T.J. Watson Research Center.

1 Introduction

1.1 Motivation and Results

Global distributed algorithms are sometimes called "optimal" when their running time is $O(n)$ (where n is the number of nodes in the network). However, it was observed [P, AKMPV, AK, GKP] that in many cases it is possible to have a running time which is $O(Diam)$, where $Diam$ is the diameter of the network, which in many actual networks may be significantly smaller than n.

In [GKP] the question of fast ($O(Diam)$) distributed algorithms was raised with respect to the classical task of *Minimum Spanning Tree* (MST) construction. It was pointed out that message congestion is a major obstacle to reducing the running time of MST algorithms. To (partially) overcome the congestion, the solution presented in [GKP] used several techniques to reduce the amount of information to be sent by combining information. One such technique was clustering, since the construction of a cluster can be relatively local (thus consuming little time). The idea was that the cluster can represent many edges and nodes, thus reducing the amount of information to be sent. The second information-reducing technique was "cycle elimination".

In this paper we first introduce a new fast distributed algorithm for the problem of computing a "small" k-dominating set in the network (where "small" here means a set of size at most $n/(k+1)$). The algorithm also partitions the network into clusters, such that each cluster has a member of the k-dominating set as its center. This new clustering technique can be plugged into the [GKP] algorithm to improve its running time. It seems that this rather general clustering method can be useful for other distributed tasks as well. For example, this type of clusters was used in [PU] in the context of routing with sparse routing tables, although no distributed protocol was given there for computing them. Efficient construction for k-dominating sets also applicable in the context of distributed data structures [P2], where it is proposed that a set of k-dominating centers can be selected for locating copies of a distributed directory.

Next, we present a convergecast protocol that informs a center about edges in the graph. The information about edges that close cycles is not forwarded. This very simple protocol is similar to a one used by [GKP], and the novelty lies in the new careful analysis proving that the convergecast process is fully pipelined. The time efficiency provided by the pipelining removes the need for the above mentioned "cycle elimination" technique of [GKP], thus simplifying the algorithm considerably.

The running time of our fast MST protocol is $O(n^{\frac{1}{2}+\epsilon} + Diam(G))$ for an arbitrarily small $\epsilon > 0$. Previous MST algorithms had running time of $O(n \log n)$ [GHS], $O(n \log^* n)$[CT, G], $O(n)$ [A] and $O(n^{0.614} + Diam)$ [GKP].

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 minimum weight spanning tree (MST) for G, namely, 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 artificially by appending the adjacent node's numbers to them. However, it is known that if the graph has neither distinct edge weights nor distinct node identifiers, then no distributed algorithm exists for computing an MST with a bounded number of messages [GHS].

For every subgraph F of the network, let $Diam(F)$ denote the diameter of F, i.e., the maximum distance between any two vertices of F, where distance is measured in the unweighted sense, i.e., in number of hops. Similarly, let $Rad(F)$ denote the radius of F, i.e., the maximum distance from any vertex of F to some center vertex of F (where the center of F is chosen so as to minimize this distance). For a collection P of subgraphs, let $Diam(P)$ (resp., $Rad(P)$) denote the maximum diameter (resp., radius) of a subgraph in P.

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 in which messages have size $O(\log n)$, and a node may send at most one message over 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, A, GKP].)

1.3 Paper Structure

The new MST algorithm is composed of two parts: Algorithm `Modified_Controlled_GHS`, followed by Algorithm `Pipeline`. These algorithms are described in Sections 3 and 4. Algorithm `Modified_Controlled_GHS` makes use of the partitioning algorithm developed in Section 2. Finally, the total time complexity is analyzed in Section 5.

2 Distributed Computation of a Small k-Dominating Set

As discussed in Subsection 2.1, it is known that a "small" (size $n/(k+1)$) k-dominating set always exists. The fast distributed algorithm for computing such a set is developed in stages. First, in Subsection 2.2 we show a distributed procedure to compute the set on a graph G in time $O(Diam(G))$. In Subsection 2.3 we show a fast distributed algorithm for computing a 1-dominating set with certain useful properties. This is next used in Subsection 2.4 to construct a fast distributed algorithm for partitioning a graph into low ($O(k^2)$) diameter clusters. Finally, in Subsection 2.5 we present a fast distributed algorithm that computes the desired "small" k-dominating set by first applying the fast partitioning algorithm of Subsection 2.4, and then running the procedure of Subsection 2.2 inside each (low-diameter) cluster.

2.1 Existence of Small k-Dominating Sets

A *k-dominating set* for a graph G is a subset D of vertices with the property that for every $v \in V$ there is some $u \in D$ such that $dist(u,v) \leq k$. (This definition is identical to that of [CN, PU] for example, but differs from other previous usages of this notion, e.g., [CGS].) For every such k-dominating set we shall define a partition P of the nodes, by associating with each node $v \in V$ a *dominator* $\mathcal{D}(v) \in D$, which is the node closest to v in the dominating set (breaking ties arbitrarily). The following is well-known (cf. [PU]).

Lemma 2.1 *For every connected graph G of n vertices and for every $k \geq 1$ there exists a k-dominating set D such that $|D| \leq \max\{1, \lfloor \frac{n}{k+1} \rfloor\}$.*

To motivate the distributed algorithm developed for the k-dominating set problem in Subsection 2.2, let us overview the proof given for the above lemma in [PU]. Let T be an arbitrary rooted spanning tree for G and denote its depth (the distance from the root) by h. If $k \geq h$ then D may consist of the root alone. Otherwise, divide the vertices of V into levels $T_0, \ldots, T_h$ according to their height in the tree, assigning all the vertices of height i to T_i. Now merge these sets into $k+1$ sets $D_0, \ldots, D_k$ by taking

$$D_i = \bigcup_{j \geq 0} T_{i+j(k+1)}$$

(i.e., T_i and every $(k+1)$st level thereafter). Clearly every D_i is a k-dominating set, and since these sets form a complete disjoint partition of V, at least one of the sets is of size at most $\lfloor \frac{n}{k+1} \rfloor$.

2.2 Distributed Computation of the Set in Diameter time

The proof of Lemma 2.1 suggests a sequential algorithm to find the k-dominating set claimed in the lemma. However, we need to show a distributed algorithm that can implement this computation in time $O(Diam(G))$ given a graph G and a root node r.

To initialize this process, it is necessary to construct the BFS tree, mark the levels and let each node know its depth, the total tree depth, and the set D_l it belongs to. This is done by the following procedure `Initialize`.

Procedure `Initialize`

1. Perform a breadth-first search (BFS) on the graph, starting from the root r (terminating at r). Label every node v by its distance from the root, $Depth(v)$.
2. A node v joins the unique set D_l for $0 \leq l \leq k$ such that $Depth(v) = l \pmod{(k+1)}$.
3. Perform a "broadcast and echo" process from the root to learn the largest distance label $M = \max_v\{Depth(v)\}$ (namely, the depth of the tree). Specifically, the root broadcasts a request throughout the tree. Each leaf that receives the request echos back a message containing its depth. An internal node that received echos from all its children sends an echo to its parent, containing the maximum depth label received from its children.
4. Broadcast the tree depth M to all nodes.

Let us next describe a procedure `Census`(l) whose task is to count the number of vertices in the set D_l. The procedure operates by a simple convergecast process on the tree from the leaves upwards. Each node v holds a counter $counter(v,l)$, used by the procedure for counting the number of D_l nodes in the subtree rooted at v.

Procedure `Census`(l)

Code for a vertex v in depth $Depth(v) = i$, with children $u_1, \ldots, u_p$.

Assumption: The procedure starts operating at time t_0.

1. If v is not a leaf: At time $t_0 + (M - i) - 1$, receive from children messages containing the values of $counter(u_i, l)$ for $1 \leq i \leq p$.
2. At time $t_0 + (M - i)$ do:

(a) Compute $Below = \sum_i counter(u_i, l)$.

(b) Set

$$counter(v,l) = \begin{cases} 1 + Below, & \text{if } v \in D_l; \\ Below, & \text{otherwise.} \end{cases}$$

(c) If not the root, then send $counter(v,l)$ to parent in the tree.

Lemma 2.2 *At the end of Procedure* `Census`(l)*, the counter* $counter(v,l)$ *holds the number of* D_l *nodes in the subtree rooted at* v*, for each node* v*. In particular, for the root* r*,* $counter(v,l) = |D_l|$. ∎

We are now ready to combine the above procedures into an algorithm for selecting a small k-dominating set. The key to an efficient combination lies in *staggering* the executions of the different `Census` operations, so that they are pipelined, and do not disrupt one another.

Algorithm `Diam_DOM`

1. Apply Procedure `Initialize`.
2. Let t_1 be the time unit just after the completion of this procedure.
   For $l = 0$ to k do:
   At time $t_1 + l$, start Procedure `Census`(l).
3. After completion of all `Census`(l) executions, the root r computes l such that $counter(r,l)$ is the minimum. D_l is the output k-dominating set.

Lemma 2.3 *Algorithm* `Diam_DOM` *computes the* k*-dominating set claimed in Lemma 2.1, and its time complexity is* $5 \cdot Diam(G) + k$.

Proof: It is immediate from Lemma 2.2 that the algorithm is an implementation of the construction in the proof of Lemma 2.1, and thus it computes a dominating set as claimed therein. The time requirements of Procedure `Initialize`, constructing the synchronous BSF and performing the "broadcast and echo" process, is $4 \cdot Diam(G)$. At this point every node can calculate the time t_1. The lemma now follows from the crucial observation that the separate executions of Procedure `Census`(l) for $0 \leq l \leq k$ never disrupt one another, since the participation of a given vertex v of depth i in a particular execution `Census`(l) is limited to (possibly) receiving messages at time $(t_1 + l) + (M - i) - 1$ and sending a message at time $(t_1 + l) + (M - i)$, hence no collisions occur. Therefore the last `Census` operation terminates after $Diam(G) + k$ time. ∎

Remark: By employing a slightly more involved pipelining of the `Census` operations, it is possible to reduce the total completion time to $5 \cdot Diam(G)$. The key observation is that the `Census` operations can be staggered on the basis of their *start level* rather than their *start time*. The point is that not all `Census` operations need to start at the nodes of depth

M; in particular, a `Census`(l) operation corresponding to a set D_l whose lowest level is at depth $M - j$ can start from that level. Hence the $k + 1$ `Census` operations can all start at the same time t_1, only at different levels of the tree. This improvement does not lead to any asymptotic improvement in the complexity of our algorithm, hence we preferred the simpler version of the algorithm.

2.3 A Distributed Balanced Dominating Set Algorithm

Definition 2.4 *A* balanced *dominating set of a graph G with n nodes is a set D of nodes in G, and an associated partition P of G's nodes with the following properties:*

(a) $|D| \leq \frac{n}{2}$.

(b) D is a dominating set.

(c) In the partition P, each cluster has at least two nodes (including the node in D).

The algorithm `Balanced_DOM` is a variant of the dominating set algorithm `Dist-SDS` in [GKP]. The changes are necessary since Algorithm `Dist-SDS` did not ensure that every cluster has at least two nodes. Namely, the partition P might contain a singleton set, with a node of D that dominates no other node.

Algorithm `Balanced_DOM`

1. Perform Algorithm `Dist-SDS` of [GKP] on a BFS spanning tree of the graph G. Let D and P be the output dominating set and partition.
2. For every singleton $\{v\}$ in P, v quits the set D, and selects an arbitrary neighbor u (possibly $u \notin D$) as its dominator, $\mathcal{D}(v) = u$.
3. Each node $u \notin D$ that was selected as a dominator in step (2) adds itself to D, quits its old cluster in P, and forms a new cluster, consisting of itself and all the nodes that chose it as their dominator.
4. Consider a node $v \in D$ whose cluster in the modified partition P is now a singleton. Then v chooses arbitrarily one node u that was in v's cluster in the original partition and left it in step (2), and joins u's cluster. Also, v quits the dominating set.

Lemma 2.5 *Algorithm* `Balanced_DOM` *constructs a balanced dominating set.*

Proof: We first observe that whenever some node v is left as a singleton after step (3) of the algorithm, a node u as defined in step (4) of the algorithm exists. To see this, let us examine the status of v in the original partition produced on step (1). First we exclude the possibility that v has joined D in step (3), since in that case there would be at least one other node u that chose v as its dominator in step (2) and joined its cluster. Hence v must

have belonged to the original dominating set D produced in step (1). Next, we note that v cannot have belonged to a singleton in the original partition of step (1), since all of those have quit D by step (2). Hence v's original cluster must have contained some other vertex u that has left it by step (3).

Now properties (b) and (c) are immediate from the algorithm. Property (a) follows from property (c). ∎

2.4 Fast Distributed Partitioning

Given Algorithm `Balanced_DOM` it is easy to partition a graph into clusters:

Algorithm `DOM_Partition`(k)

For $\lceil \log k \rceil$ times **do**:

1. Perform Algorithm `Balanced_DOM`, assigning each node not in the dominating set to a cluster with an arbitrary neighbor in the dominating set.
2. Contract each cluster to one node. The distributed implementation of this contraction is that the center node of the cluster (the one from the dominating set D) will from now on perform the operations for the whole cluster, while the other nodes in the cluster will just serve as links between it and the other clusters.

End_For

Lemma 2.6 *Every cluster C in the partition P output by Algorithm* `DOM_Partition` *has the following properties.*

(a) $|C| \geq k$.

(b) $Rad(C) \leq 4k^2$.

Proof: Property (a) follows from Lemma 2.5 and property (c) in Def. 2.4. Also, by the same lemma and property (b) in Def. 2.4, the maximum radius r_j after the jth execution of Algorithm `Balanced_DOM` obeys the relation $r_j \leq 3r_{j-1} + 1$ (for $j > 1$). This implies that after $\lceil \log k \rceil$ executions of Algorithm `Balanced_DOM` the radius of a cluster is bounded by

$$4^{\lceil \log k \rceil} \leq 4^{\log k + 1} \leq 4k^2.$$

Property (b) of the Lemma follows. ∎

2.5 Fast Distributed Computation of Small k-dominating sets

The fast algorithm `Fast_DOM` is now obtained simply as the combination of the previous algorithms. Namely:

Algorithm `Fast_DOM`

1. Perform Algorithm `DOM_Partition`(k), yielding the partition P'.
2. Perform Algorithm `Diam_DOM` on each cluster C_i of the partition P'.

Let D_i, P_i be the resulting k-dominating set and partition of the cluster C_i, for each i. Let $D = \bigcup_i D_i$, and let P be the resulting partition of the entire graph (consisting of the clusters of all partitions P_i).

Corollary 2.7 *(a)* $|D| \leq \frac{n}{k+1}$.

(b) $Rad(P) \leq k$.

Proof: By Lemma 2.1, $|D_i| \leq \lfloor \frac{|C_i|}{k+1} \rfloor$. Thus $|D| = \sum_i |D_i| \leq \frac{1}{k+1} \sum_i |C_i| = \frac{n}{k+1}$. The radius bound on the partition P follows from Lemma 2.3 for each P_i separately. ∎

Lemma 2.8 *The time complexity of Algorithm* `Fast_DOM` *is* $O(\log k \cdot 2^{\sqrt{\log n}} + k^2)$.

Proof: The complexity of Algorithm `DOM_Partition` is $\lceil \log k \rceil$ times that of Algorithm `Balanced_DOM`, which can easily be seen to have the same order of magnitude of Algorithm `Dist-SDS` in [GKP], i.e. $O(2^{\sqrt{\log n}})$ (where n is the number of nodes in the original graph). The complexity of Algorithm `Diam_DOM` on each cluster C_i is of the order of the diameter of C_i, which is $O(k^2)$ by property (b) of Lemma 2.6. The lemma follows. ∎

3 First Part: The Modified GHS Protocol

In this section we use the `Controlled-GHS` algorithm from [GKP] with a very slight modification. The algorithm in [GKP] used a regular dominating set, while here it uses the k-dominating set computed by algorithm `Fast_DOM` presented in Section 2. Let us first describe the original GHS algorithm [GHS], and then the `Controlled-GHS` algorithm of [GKP].

3.1 Brief Description of the `GHS` Algorithm

In the original distributed `GHS` algorithm of [GHS], 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. (In the sequel we loosely use the word "fragment" to mean both the collection of nodes F and the corresponding tree $T(F)$.) 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 upcasts only the lowest-weight edge it has 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.)

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). This problem is handled by using special "level" rules for merging fragments, designed to prevent these complications.

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

3.2 The `Controlled-GHS` Algorithm

Next, we sketch the modified, *controlled* version of `GHS` (named `Controlled-GHS`) given in [GKP]. This algorithm achieves the following:

1. Upon termination, the number of fragments is bounded above by N (for some specific $N \ll n$).
2. Throughout the execution, the diameter of every fragment F satisfies $Diam(F) \leq d$ (for some specific d).

Intuitively, since [GKP] focuses on a synchronous solution, and ignores communication complexity, this version of `GHS` is simpler than the original algorithm. In particular, it is oblivious to balancing fragment sizes, and does not require the "level" rules used in the original algorithm.

More specifically, `Controlled-GHS` also starts with singleton fragments, and executes a total of I iterations. 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.

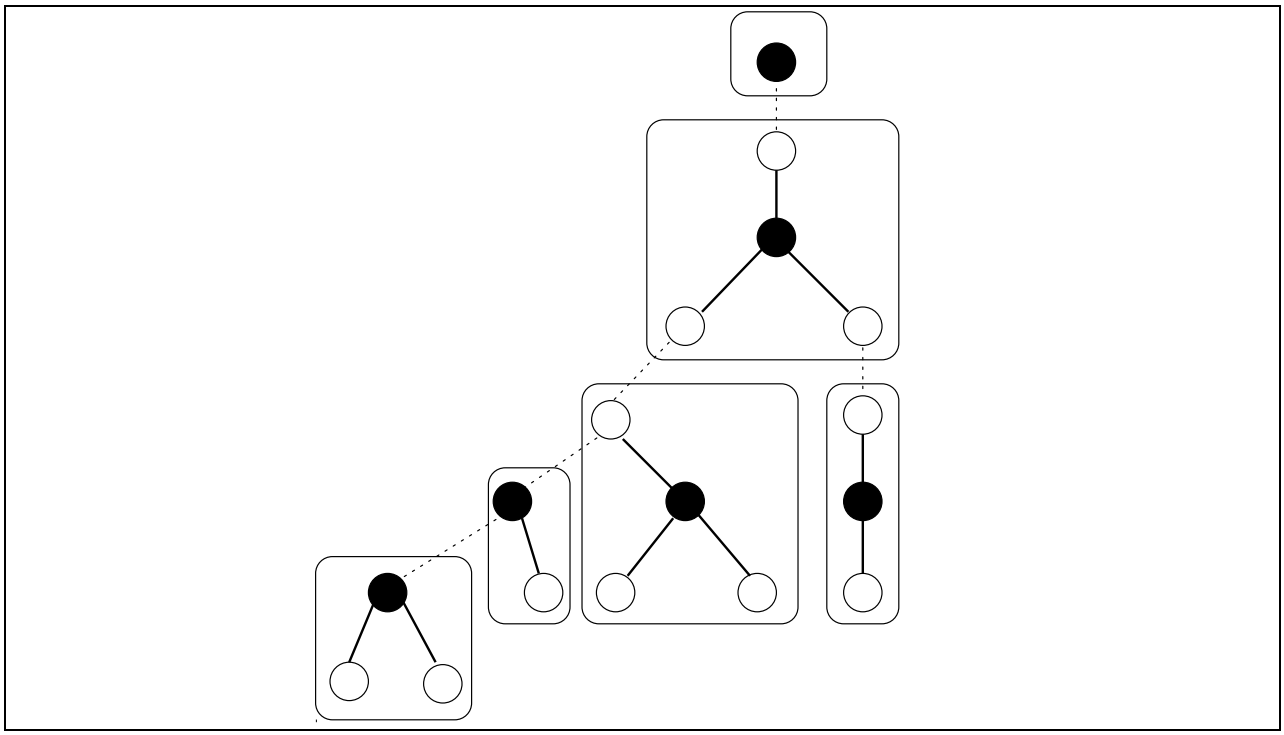

Figure 1: A Phase of `Controlled-GHS`: the fragment tree T is partitioned into "small" trees (represented by the solid edges) according to the dominating set composed of the dark nodes.

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 each of the resulting trees $T = T(F)$ into "small" ($O(1)$ depth) trees, and merge only these small trees.

This process is depicted in Figure 1. Stage 2 is accomplished by computing a dominating set $M(T)$ on each tree $T = T(F)$ 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, hence bounding the diameter of the resulting fragments. The dominating sets are computed using the aforementioned distributed procedure `Dist-SDS`, applied separately to each tree $T(F)$ in the fragment forest FF.

3.3 The Modification to the `Controlled-GHS` Algorithm

`Modified_Controlled_GHS` also starts with singleton fragments, and executes a total of I phases (for I to be determined later). Each phase of `Modified_Controlled_GHS` consists of two stages similar to those of algorithm `Controlled-GHS`, except that in the second stage,

the fragment trees induced by the preferred edge in Algorithm GHS are broken into $O(k^2)$ depth trees (using Algorithm DOM_Partition), instead of $O(1)$. Hence Stage 2 of algorithm Controlled-GHS is replaced by:

Stage 2': Using Algorithm Fast_DOM of Sect. 2.5, break the trees of the fragment forest FF into "small" ($O(k^2)$ depth) trees, and merge only these small trees.

Lemma 3.1 *After i iterations of Algorithm* Modified_Controlled_GHS *the resulting fragment forest FF_i has the following properties:*

(a) its radius is $Rad(FF_i) \leq (2k+2)^i$,

(b) the number of the fragments $|FF_i| \leq \frac{n}{(k+1)^i}$,

(c) the tree $T(F)$ constructed for each fragment $F \in FF_i$ consists of edges of the MST.

Proof: The proof for properties (a) and (b) follows by induction on i, by Corollary 2.7 and by observing that (denoting the maximal cluster radius after the ith iteration by r_i), $r_i \leq (2k+1)r_{i-1} + k$. Property (c) follows, as in [GHS, GKP], from the correctness of the "blue rule" used for constructing the fragments (cf. [T], p. 71). (Essentially, the blue rule states that an edge that is the minimum between two fragments is a part of an MST.) ∎

Lemma 3.2 *The complexity of Algorithm* Modified_Controlled_GHS *is $O(2^{\sqrt{\log n}}(2k+2)^I)$.*

Proof: Since each node in the graph on which we perform the ith iteration is in fact a fragment of FF_{i-1} that resulted from the $(i-1)st$ iteration, the complexity of the ith iteration is that of Algorithm Fast_DOM, slowed down by a factor proportional to $Rad(FF_i)$, namely, $O((\log k \cdot 2^{\sqrt{\log n}} + k^2)(2k+2)^i)$. For a constant k the lemma follows. ∎

4 Second Part: Global Edge Elimination by Pipelining

The second part of our algorithm starts at the point where we are given a fragment graph $\tilde{F}$, whose vertex set $V(\tilde{F})$ is a fragment forest FF containing $N = n/(k+1)^I$ fragments of the MST, and whose edge set $E(\tilde{F})$ is a collection of (possibly more than $O(N)$) inter-fragment edges, which are candidates for joining the MST. We now proceed to reduce the total number of remaining inter-fragment edges to the necessary $N-1$.

4.1 Algorithm

We need the following technical definition. For a set of edges Q and a cycle-free subset $U \subseteq Q$, define $Cyc(U,Q)$ as the set of all edges $e \in Q \setminus U$ such that $U \cup \{e\}$ contains a cycle.

We are now ready to describe our pipelined procedure.

Procedure Pipeline

1. Build a *breath-first search* tree B on G, with a root $r(B)$.

2. Throughout the execution, each node v on the tree maintains a set Q of all the inter-fragment edges it knows of (either directly or by hearing of from its children in the previous rounds), ordered by non-decreasing edge weights. Initially this set contains only the inter-fragment edges adjacent to the fragment F whose center $r(F)$ is v itself (if such F exists). It also maintains a set U (initially empty) of all the edges it has already sent up to its parent in the tree.

3. A leaf v starts sending edges upwards at pulse 0. An intermediate node v starts sending at the first pulse after it has heard from all its children.

4. At each pulse i, the node sends up to its parent in the tree the lightest edge e in Q that has not been sent up in the previous rounds up to round $i-1$ ($e \notin U$), and does not close a cycle with the edges of U (if such an edge exists).

   More formally, the node computes the set of *remaining candidates*

   $$RC = Q \setminus (U \cup Cyc(U, Q)).$$

   If $RC = \emptyset$ then v sends a "terminating" message to its parent in the tree, and terminates its participation in the protocol. Else, it sends up to its parent the lightest edge e in RC.

5. The root $r(B)$ computes locally the set S of the $N-1$ edges participating in the MST of the fragment graph $\tilde{F}$, from among the edges it hears of from its children.

6. The root $r(B)$ then broadcasts (over the tree B) the resulting set of $N-1$ inter-fragment edges (constituting the MST for the fragment graph $\tilde{F}$) to all nodes in G.

4.2 Analysis

Our analysis hinges on two main properties of the procedure. First, the edges reported by each intermediate node to its parent in the tree are sent in nondecreasing weight order. Secondly, each intermediate node transmits edges upwards in the tree *continuously*, until it exhausts all the reportable edges from its subtree. Namely, once the set of candidates RC is empty, the node will learn of no more reportable edges.

Let us first make the following straightforward but crucial observation.

Lemma 4.1 *The edges reported by each intermediate node to its parent in the tree form a forest.*

Proof: This follows immediately from the rule used by the procedure to select the next edge to be transmitted upwards. ∎

Lemma 4.2 *Every node v starts sending messages upwards at pulse $\hat{L}(v)$, where $\hat{L}(v)$ is the* level *function defined as follows:*

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

Proof: By straightforward induction on the tree structure, from the leaves upwards. ∎

Our main technical lemma concerns the properties of a node on the tree in some round of the algorithm. Consider an intermediate node v at level L, that has still not terminated its participation in the algorithm, at the beginning of round t, for some $t \geq L$. Note that each of the children of v in the tree is of level $L-1$ or less, hence by Lemma 4.2 they started transmission at round $L-1$ or earlier. Call a child *active* if it has not terminated yet (i.e., it has upcast an edge to v on round $t-1$). Let

$$A_t(v) = \{v_1, \ldots, v_p\}$$

be the set of v's active children at round t.

Lemma 4.3 *(a) At the beginning of round t, the candidate set RC examined by v contains at least one candidate edge upcast by each of its active children from $A_t(v)$.*

(b) If v upcasts an edge of weight ω at round t, then the edge it was informed of on round $t-1$ by any of its active children, was of weight ω or greater.

(c) If v upcasts an edge of weight ω at round t, then any later edge it will learn of is of weight ω or greater.

(d) Node v upcasts edges in nondecreasing weight order.

Proof: We prove the lemma by induction on the structure of the tree, starting from the leaves upwards.

A *leaf* v has no (active or other) children, and therefore Claims (a), (b) and (c) hold vacuously. Claim (d) follows trivially from the rules of the procedure.

Let us now consider an intermediate node v, and assume that the claims hold for each of its children. We need to prove the four claims for v. We start with Claim (a).

Let U be the set of m edges upcast by v during the first $m = t - L$ rounds it has participated in (namely, rounds $L, \ldots, t-1$ if $t > L$). By Lemma 4.1, U forms a forest in G. Consequently, break U into the trees $U_1, \ldots, U_\ell$ in G, with $x_i = |U_i|$, where each such tree U_i has a vertex set $V(U_i)$ of exactly $x_i + 1$ vertices, and

$$\sum_i x_i = |U| = m. \tag{1}$$

Consider an active child u of v. Denote the set of edges upcast by u so far (up to and including round $t-1$) by D. Since u was still active on round $t-1$, it has transmitted continuously to v since round $\hat{L}(u)$, which as discussed before is at most $L-1$. Therefore, we conclude that

$$|D| \geq m+1. \tag{2}$$

Suppose, for the sake of contradiction, that none of the edges upcast by u is a candidate on round t. In other words, for each edge $e \in D$, either e was upcast by v earlier (namely, $e \in U$), or e closes a cycle with the edges of U (namely, $e \in Cyc(U)$).

Thus every such edge e has both of its endpoints in U. Furthermore, e cannot possibly connect endpoints that belong to two different trees U_i and U_j (since in that case, e would have been in neither U nor $Cyc(U)$).

This implies that the set D can be partitioned into sets $D_1, \ldots, D_\ell$ such that all the edges of D_i are restricted to vertices $V(U_i)$ of the tree U_i, for $1 \leq i \leq \ell$. Moreover, notice that each such set D_i is a forest, since the entire set D is a forest by lemma 4.1.

The last two facts combined imply that

$$|D_i| \leq |V(U_i)| - 1 = x_i. \tag{3}$$

Combining (1), (2) and (3) together, we get

$$m+1 \leq |D| = \sum_i |D_i| \leq \sum_i x_i = m,$$

contradiction. Hence Claim (a) must hold.

Next, we prove Claim (b) as follows. Consider some active child u of v. Let e be the edge upcast by u on round $t-1$. (Note that e does not necessarily have to be in RC.) Let e' be some edge that was upcast by u at some round $t' \leq t-1$ and is still in the candidate set RC on round t (such an edge must exist by Claim (a)). By the inductive hypothesis of Claim (d), $\omega(e) \geq \omega(e')$. By the edge selection rule of node v, $\omega(e') \geq \omega$. Claim (b) follows.

Next, we note that Claim (c) follows trivially from Claim (b). Finally, Claim (d) follows trivially from Claim (c) and the edge selection rule of the procedure. ∎

Finally, we need to argue that nodes do not terminate the algorithm prematurely.

Lemma 4.4 *After a node v has terminated its participation in the algorithm, it will learn of no more reportable edges.*

Proof: We need to argue that once the set RC becomes empty, no new candidate edges will become known to v. This follows directly from claim (a) of Lemma 4.3, which guarantees that if RC is empty on round t, none of v's children have upcast it an edge in round $t-1$, and hence all of them have already terminated. ∎

Lemma 4.5 *The running time of Procedure* `Pipeline` *is bounded by* $O(N + Diam(G))$*, and its output is an MST for* G*.*

Proof: The fact that the resulting tree is an MST follows from Lemma 3.1 and from the correctness of the "red rule" employed for edge elimination in the procedure (cf. [T], p. 71). (Essentially the red rule says the an edge that is the heaviest on any cycle is not a part of any MST.)

As for the running time, the bound is derived from the following facts. First, the root of the tree receives at most N edges from its children. Secondly, the children send these edges to the root in a fully pipelined fashion (namely, without stopping until exhausting all the edges they know of). Finally, the root starts getting such messages at time $Diam(G)$ at the latest. ∎

5 The Combined Algorithm

As explained in the introduction, Algorithm `Fast_MST` is obtained by combining the algorithms of the last two sections in sequence.

Algorithm `Fast_MST`

1. Perform Algorithm `Modified_Controlled_GHS`.
2. Perform Algorithm `Pipeline`.

Recall that by Lemmas 3.2 and 4.5, the complexity of Part 1 is $O(2^{\sqrt{\log n}}(2k+2)^I)$ and that of Part 2 is $O(Diam(G)+\frac{n}{(k+1)^I})$. The optimal I is thus obtained when $(2k+2)^I = \frac{n}{(k+1)^I}$, namely $I = \frac{\log n}{\log(2(k+1)^2)}$. For this I the combined complexity of Algorithm `Fast_MST` is $O(Diam(G) + 2^{\sqrt{\log n}}(2k+2)^I)$, where

$$(2k+2)^I = (2k+2)^{\frac{\log n}{\log(2(k+1)^2)}} = n^{\frac{\log(2(k+1))}{\log(2(k+1)^2)}} \equiv n^{\frac{1}{2}+\epsilon}$$

for

$$\epsilon = \epsilon(k) = \frac{1+\log(k+1)}{1+2\log(k+1)} - \frac{1}{2} = \frac{1}{2+4\log(k+1)} \quad .$$

Note that ϵ tends to 0 as k goes to infinity. That is, for every $\delta > 0$ it is possible to select a constant $k \geq 2^{\frac{1-2\delta}{4\delta}} - 1$ such that $\epsilon = \epsilon(k) \leq \delta$.

In summary, since $2^{\sqrt{\log n}}$ is $o(n^{\varepsilon'})$ for any $\varepsilon' > 0$, we have shown:

Theorem 5.1 *For every* $\epsilon > 0$*, there exists a distributed Minimum-weight Spanning Tree algorithm with time complexity* $O(n^{\frac{1}{2}+\epsilon} + Diam(G))$*.* ∎

References

[AK] S. Aggarwal and Shay Kutten, Time-Optimal Self Stabilizing Spanning Tree Algorithms, *Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, December 1993.

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

[AKMPV] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir and G. Varghese, Time-Optimal Self Stabilizing Synchronization, *Proc. 1993 ACM STOC*, San Diego, California, Pages 652–661, May 1993.

[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.

[CT] F. Chin and H. F. Ting, An Almost Linear Time and $O(n\log(n)+e)$ Messages Distributed Algorithm for Minimum-Weight Spanning Trees, *26th Symp. on Foundations of Computer Science*, Oct. 1985, pages 257–266.

[CGS] E.J. Cockayne, B. Gamble and B. Shepherd, An Upper Bound for the k-Domination Number of a Graph, *J. of Graph Theory*, Vol. 9, (1985), 533–534.

[CN] G.J. Chang and G.L. Nemhauser, The k-Domination and k-Stability Problems on Sun-Free Chordal Graphs, *SIAM J. Alg. & Disc. Meth.* Vol. 5, (1984), 332–345.

[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), (1983), 66–77.

[GKP] J. Garay, S. Kutten and D. Peleg, A Sub-Linear Time Distributed Algorithm for Minimum-Weight Spanning Trees, 34^{th} *IEEE Symp. on Foundations of Computer Science*, pages 659–668, November 1993.

[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.

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

[P2] D. Peleg, Distributed Data Structures: A Complexity Oriented View, *Proc. 4th Int. Workshop on Distributed Algorithms*, Bari, Italy, Sept. 1990, 71–89.

[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.

[PU] D. Peleg and E. Upfal, A tradeoff between size and efficiency for routing tables, *J. of the ACM*, Vol. 36, (1989), 510–530.

[T] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.