Fast Distributed Construction of k-Dominating Sets and Applications

(Extended Abstract)

Shay Kutten * David Peleg †

Abstract

This paper presents 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 time complexity of the algorithm is $O(k \log^* n)$.

Small k-dominating sets have applications in a number of areas, including routing with sparse routing tables via the scheme of [PU], the design of distributed data structures [P2], and center selection in a distributed network (cf. [BKP]). The main application described in this paper concerns a fast distributed algorithm for constructing a minimum weight spanning tree (MST). On an n-vertex network of diameter d, the new algorithm constructs an MST in time $O(\sqrt{n} \log^* n + d)$, improving on the results of [GKP].

The new MST algorithm is conceptually simpler than the three-phase algorithm of [GKP]. In addition to exploiting small k-dominating sets, it uses a very simple convergecast protocol to inform a center about graph edges, that avoids forwarding messages about edges that close cycles. This convergecast protocol is similar to the one used in the third 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 the complicated second phase of the algorithm of [GKP].

1 Introduction

1.1 Motivation and Results

This paper concerns the fast construction of small k dominating sets, and its applications. A *k-dominating set* is a set D satisfying that for every node v in the graph, there is a node $w \in D$ at distance at most k from it.

*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.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
PODC 95 Ottawa Ontario CA © 1995 ACM 0-89791-710-3/95/08..$3.50

We 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 $n/(k+1)$ for $n \geq k+1$; it is easy to see that this is a lower bound). The algorithm also partitions the network into clusters, such that each cluster has a member of the k-dominating set as its center. The time complexity of the algorithm is $O(k \log^* n)$.

This new algorithm is useful for speeding up a number of distributed tasks. 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. Hence the new construction can serve to speed up the preprocessing stage of that routing scheme.

Efficient construction for k-dominating sets is 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. Likewise, such sets are useful for efficient selection of network centers for server placement, where it is desired to ensure that each node in the network is sufficiently close to some server (cf. [BKP]).

One application that we describe in detail in the paper concerns speeding up an algorithm for constructing a Minimum Spanning Tree (MST). Note that, informally speaking, MST can be thought of as a more "global" problem, as opposed to the k-Dominating Set problem which is more "local" in nature. We now present a formal notion of what we mean by a "fast" algorithm, which applies to both.

Linial [L] proved lower bounds on the time complexity of distributed algorithms, even when assuming a very strong computational model (not assumed here) where messages can be of arbitrary length. Yet sending a message to a distance d still takes time d in this model. Thus the lower bounds of [L] actually correspond only to the radius (around each node) from which information must be fetched in order to solve a given problem P. Let this radius be $R(P)$ for a given problem P. We term an algorithm for P *neighborhood optimal* if its complexity is $O(R(P))$. We use the word "fast" with respect to this notion, namely, with regard to algorithms aimed at achieving neighborhood optimality. We note that it can be shown that, assuming messages are of bounded size, there exist problems for which no neighborhood optimal algorithm exists, due to inherent congestion. (In particular, there exist problems P with constant $R(P)$ that require as much as $\Omega(n)$ time.)

Note that the inherent information radius of our k Dominating Set problem is k, hence our algorithm is neighborhood optimal up to a factor of $\log^* n$, which we term "fast".

A question raised in [GKP] is whether the MST problem can be solved in $O(Diam)$ time, where $Diam$ is the network diameter. Clearly $R(MST) = Diam$ in certain graphs. Thus in terms of the definition above one can rephrase the question of [GKP]: "does there exist a neighborhood optimal algorithm for MST". The algorithm presented in [GKP] has time complexity $O(n^{.614} + Diam)$, hence it is neighborhood optimal for the case that $Diam > n^{.614}$. Previous MST algorithms had running time $O(n \log n)$ [GHS], $O(n \log^* n)$[CT, G], and $O(n)$ [A2]. Using the fast k Dominating Set algorithm we manage to present an improved MST construction algorithm whose time complexity is $O(\sqrt{n} \log^* n + Diam(G))$. Thus the MST algorithm is now neighborhood optimal for all graphs with $Diam > n^{1/2} \log^* n$. Let us hint about how the improved MST algorithm was obtained. As noted in [GKP], the main obstacle for improving the time complexity is the congestion caused by the need to send the description of many edges. To (partially) overcome the congestion, the solution presented in [GKP] used several techniques to reduce the amount of information sent by combining it. 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. To improve the results of [GKP] we essentially replace the clustering technique used there by the clustering constructed by the k Dominating Set algorithm.

Another result presented in this paper is a new efficient pipelining technique, also aimed at overcoming congestion. This technique is motivated by the following complication in the solution of [GKP]. At a certain point in that algorithm, a center is informed about edges in the graph (those that connect clusters). Since there are "too many such edges", the algorithm of [GKP] employs a rather complex method for eliminating short cycles, and thus reducing the number of remaining edges. Instead, we use the new pipelining technique, which is very simple; the main novelty is in its analysis. As in [GKP], nodes on a tree forward the description of many edges to their parents on the tree, so that eventually the description reaches the tree root. However, here a node avoids forwarding the description of *cycle heavy* edges. That is, a node forwards the description of edges according to their weight, and does not forward the description of an edge that closes a cycle with edges whose description was already forwarded. This seems to suggest that at certain points in time a node might be forced to wait, due to not having an edge description that is eligible for forwarding. Thus it seems that the convergecast may not be pipelined, and hence may take a long time. We prove that this simple convergecast is fully pipelined, and thus its running time is the one required, eliminating the need of the complex cycle elimination stage in [GKP]. This pipelining proof may be of interest in itself.

1.2 Model and Definitions

We focus on the problem of devising a time efficient algorithm to construct a small k Dominating Set. The network is modeled as an undirected graph $G = (V, E)$, where V is the set of nodes, and E is the set of communication links between them. The nodes can communicate only by sending and receiving messages over the communication links.

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). A small k Dominating Set is of size less than or equal to $max\{1, \lfloor \frac{n}{k+1} \rfloor\}$. Clearly one cannot do better than that in the general case.

We use the common assumption 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. One can do without (either) one of these two assumptions.

For every subgraph F of the network, let $Diam(F)$ denote F's diameter, i.e., the maximum distance on F between any two vertices of F (measuring distance in the unweighted sense, i.e., in number of hops). Similarly, let $Rad(F)$ denote the radius of F, i.e., the maximum distance on F 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 unessential 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; for example, we can use the simple synchronizer α of [A1] whose cost in an asynchronous network is one message over each edge in each direction per round of the synchronous algorithm.)

Still, we shall not adopt the extreme model employed in previous studies of locality issues [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 MST problem, for instance, 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 and more common 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 the previous algorithms that use edge weights or nodes identity, e.g. [GHS, A2, GKP].)

Let us now define the Minimum Spanning Tree task solved later on as an application of the k Dominating Set algorithm. 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.

1.3 Paper Structure

In the next section we describe some tools. We then use these tools in Section 3 for developing an algorithm for constructing a small k Dominating Set on a tree. Next, in Section 4 we use the previous algorithm to develop an algorithm for k-dominating set construction on a general network. The new MST algorithm is described in Section 5.

2 Basic Algorithms for Small k-Dominating Sets

As discussed next in Subsection 2.1, it is known that a "small" (size $n/(k+1)$) k-dominating set always exists in an n-node graph. The fast distributed algorithm for computing such a set is developed in stages. In this section we only present the existence proof and then (in Subsection 2.2) give a "slow" distributed procedure, computing a k-dominating set on a graph G in time $O(Diam(G))$. In the next section we give a fast algorithm for computing a k-dominating set on an n-node *tree* T in time $O(k \log^* n)$. In the following section we extend the result to a general graph.

2.1 Existence of Small k-Dominating Sets

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 (standard) proof outlined 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 Procedure `Initialize`, described in Fig. 1.

Next, we give (in Fig. 2) 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.

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 combine the above procedures into an algorithm named `Diam_DOM` (in Fig. 3) 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.

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 r, $Depth(v)$.
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 r to learn the largest distance label $M = \max_v\{Depth(v)\}$ (namely, the depth of the tree). Specifically, r broadcasts a request throughout the tree. Each leaf receiving the request echos back 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.

Figure 1: Procedure `Initialize`.

Code for a node v in $Depth(v) = i$, with children $u_1, \ldots, u_p$. 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 $counter(u_j, l)$ for $1 \leq j \leq p$.
2. At time $t_0 + (M - i)$ do:
   (a) Compute $Below = \sum_j counter(u_j, l)$.
   (b) Set $counter(v,l) = 1 + Below$ if $v \in D_l$, and $counter(v,l) = Below$ otherwise.
   (c) If not the root, then send $counter(v,l)$ to parent in the tree.

Figure 2: Procedure `Census`(l).

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.

Figure 3: Algorithm `Diam_DOM`.

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.

3 Distributed Computation of a Small k-Dominating Set on a Tree

In this section we concentrate on a fast algorithm for computing a small k-dominating set on a tree T. (We use the non-fast algorithm of 2.2 as a subroutine.) Let us start with an overview of the algorithm.

Algorithm `Diam_DOM` of the previous section requires time $O(Diam(G))$. Hence our approach to building a small k-dominating set is based on first partitioning the tree into a *collection* of subtrees (or clusters), such that the depth of each subtree is at most $O(k)$. Then Algorithm `Diam_DOM` can be applied on each subtree separately, and the resulting dominating sets can be merged to yield the dominating set for the entire graph.

A source of difficulty is that the subtrees we construct must not be too small: for a subtree of $l < k$ vertices, any dominating set must still contain at least one vertex, which does not meet the desired ratio of $1/(k+1)$ between dominators and dominated vertices. Hence our goal is to partition the tree into subtrees of size $k+1$ or more and depth $O(k)$.

Formally, let us introduce the following definition.

Definition 3.1 *A (σ, ρ)* spanning forest *of a graph $G(V,E)$ is a collection of disjoint trees $\{T_1, \ldots, T_m\}$ with the following properties:*

1. *the trees consist of edges of E and span all the nodes of V,*
2. *each tree contains at least σ nodes, and*
3. *the radius of each tree is at most ρ.*

In Subsection 3.1 we show a fast distributed algorithm for computing a 1-dominating set with certain useful properties. This is next used in Subsection 3.2 to construct a fast distributed algorithm for partitioning a tree into $O(k)$-depth, $\Omega(k)$-size clusters. Finally, in Subsection 3.3 we present a fast distributed algorithm that computes the desired "small" k-dominating set by first applying the fast partitioning algorithm of Subsection 3.2.3, and then running Procedure `Diam_DOM` of Subsection 2.2 inside each (low-diameter) cluster.

3.1 A Distributed Balanced Dominating Set Algorithm

Definition 3.1 *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) $|D| \leq \lceil \frac{n}{2} \rceil$.

(b) D is a dominating set.

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

Algorithm `Balanced_DOM` is a variant of the dominating set algorithm `Small-Dom-Set` in [GKP]. That algorithm computed an ordinary (not necessarily balanced) dominating set on a given tree, namely, a set satisfying properties (a) and (b) above, but not property (c). Algorithm `Small-Dom-Set` made use of an MIS procedure as a black box. Specifically, it used the deterministic distributed MIS algorithm of [PS]. Since Algorithm `Small-Dom-Set` uses the MIS procedure only for computing MIS on a tree, it is possible to replace the procedure of [PS] with the faster procedure of [GPS] (which computes an MIS on an n-vertex tree in time $O(\log^* n)$). Thus the following can be said about `Small-Dom-Set`, preparing to use it in Algorithm `Balanced_DOM`:

Lemma 3.2 [GKP] *There exists a distributed procedure* `Small-Dom-Set` *that, applied in a given n-vertex tree T for $n \geq 2$, computes (in a synchronous manner) a dominating set D of size at most $\lceil n/2 \rceil$ using $O(\log n)$-bit messages, and its time complexity is $O(\log^* n)$. Furthermore, the output of the procedure has the property that each node in D has some neighbor outside D.*

The property described in the last statement of the lemma is not argued in [GKP], but can easily be shown to hold.

Note that even with these changes, Algorithm `Small-Dom-Set` does not guarantee property (c). The modified variant, Procedure `Balanced_DOM` described in Fig. 4, will ensure also the additional property that every cluster has at least two nodes, namely, the partition P will not contain any singleton set.

Lemma 3.3 *On a tree of $n \geq 2$ vertices, Algorithm* `Balanced_DOM` *constructs a balanced dominating set and requires time $O(\log^* n)$.*

Proof: We first observe that step (2) can always be performed, by the last property in Lemma 3.2.

Next, we claim 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

1. Perform Algorithm `Small-Dom-Set` of [GKP] on the tree T. 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 $u \notin D$ 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.

Figure 4: Algorithm `Balanced_DOM`.

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). The time bound follows from Lemma 3.2, since all the additional steps require constant time. ∎

3.2 Fast distributed Partitioning of a tree

In this subsection we show how to partition a given tree of size $n \geq k+1$ into clusters of sufficient size (at least $k+1$) and small radius. For the sake of clarity we develop the algorithm in several steps. In Section 3.2.1 we start by giving a very simple algorithm that constructs a $(k+1, O(k^2))$ spanning forest, with time complexity which is $O(k^2 \log^* n)$. In Section 3.2.2 we show how to limit the growth of the trees in the forest, so that the output is a $(k+1, O(k))$ forest, with running time which is $O(k \log k \log^* n)$. Finally, in 3.2.3 we show how to improve the running time by a factor of $\log k$, using an idea similar to those used before e.g. in [AG, G, CT, A2, JM].

3.2.1 Constructing a $(k+1, O(k^2))$ spanning forest

The algorithm `DOM_Partition_1`(k) described in Figure 5 for constructing this partition operates on a tree T via repetitive applications of Procedure `Balanced_DOM`. Each application constructs a balanced dominating set and an associated partition on the tree, and then contracts each cluster in the partition into a single node, thus forming a (smaller) tree for the next iteration.

Let us remark that the distributed implementation of the contraction in the algorithm requires us to appoint for each

For $\lceil \log(k+1) \rceil$ 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.

End_For

Figure 5: Algorithm `DOM_Partition_1`(k).

cluster a center node, that 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 cluster centers.

For this simple algorithm, the properties of Algorithm `Balanced_DOM` enable us to prove the following.

Lemma 3.4 *Algorithm* `DOM_Partition_1`(k) *requires time* $O(k^2 \log^* n)$. *Every cluster* C *in the output partition* P *satisfies* $|C| \geq k+1$ *and* $Rad(C) \leq 4k^2$. ∎

3.2.2 Constructing a $(k+1, O(k))$ spanning forest

The straightforward variant of the algorithm presented in the previous subsection suffices to guarantee the requirements of cluster size $k+1$ or more and cluster radius $O(k^2)$. However, we would like to modify this algorithm so that it has a better running time and a better bound on the radius of each cluster in the partition, namely, $O(k)$. This is achieved by employing the following simple idea: while constructing the partition, clusters that have reached depth greater than k are erased from the tree, and hence stop growing.

The algorithm, named `DOM_Partition_2`(k), is complicated by the fact that the elimination of the large enough clusters from the tree causes it to split into a forest of subtrees. Subsequent invocations of Procedure `Balanced_DOM` are therefore applied to each of these subtrees separately. Hence we formally describe the algorithm as applied to a *forest* of trees $\mathcal{T}$ (initially containing the single tree T).

One problem with this idea is that the "small" clusters may not be able to grow solely by joining other small clusters. In particular, it may be the case that a small cluster has no neighboring small cluster; all its neighboring clusters have grown beyond the specified bound and were consequently eliminated from the tree. On the other hand, if such a small cluster is allowed to join a neighboring large cluster, how can we control the growth of the large cluster, so that its diameter will not become much larger than k?

The solution is to attempt to merge small trees together as long as possible, and merge small trees to neighboring large trees only at the very last step of the algorithm (i.e., only once). This maintains a reasonable bound on the depth of the resulting trees.

Algorithm `DOM_Partition_2`(k) is described in detail in Fig. 6. In every intermediate iteration i, the algorithm associates with each node v a cluster $\mathcal{C}_i(v)$ consisting of the original nodes (of the input tree T) of the subtree that was

contracted into v. Also, the algorithm maintains a forest T_i of trees to be handled in the ith iteration.

1. Set $T_1 = \{T\}$ and $S = P_{out} = \emptyset$.
   /* P_{out} is the output partition.
   S collects small isolated trees */
2. For every vertex v set $\mathcal{C}_1(v) = \{v\}$.
3. **For** $i = 1$ to $\lceil \log(k+1) \rceil$ **do:**
   (a) /* Form the new forest T_{i+1}. */
   For every tree T' in the forest T_i **do:**
   - Perform Algorithm `Balanced_DOM` on T', yielding a partition P.
   - Contract each cluster C of P into a single node v, and set $\mathcal{C}_{i+1}(v) = \bigcup_{w \in C} \mathcal{C}_i(w)$.

   End_For
   $T' \leftarrow$ resulting forest of contracted trees.
   (b) /* Remove sufficiently large clusters. */
   For every tree T' in the forest T', and for every node v in T', if the tree spanning $\mathcal{C}_{i+1}(v)$ has depth $k+1$ or greater **then do:**
   - Erase v from the tree T' (thus splitting T' into a forest).
   - Make $\mathcal{C}_{i+1}(v)$ a cluster in the output partition P_{out}.

   End_For
   $T'' \leftarrow$ resulting forest.
   (c) /* Remove lone small clusters. */
   For every tree T'' in the forest T'', if T'' consists of a single node v then **do:**
   - Erase T'' from the forest T''.
   - Add the cluster $\mathcal{C}_{i+1}(v)$ to the set S.

   End_For
   $\mathcal{T}_{i+1} \leftarrow$ resulting forest.

   End_For
4. **For** each cluster $\mathcal{C}(v)$ in the set S **do:**
   (a) If $|\mathcal{C}(v)| > k$ then move it to P_{out} as is.
   (b) Else Merge $\mathcal{C}(v)$ into some neighboring cluster $\mathcal{C}'$ in P_{out}.

   End_For

Figure 6: Algorithm `DOM_Partition_2`(k).

Lemma 3.5 *The collection P_{out} output by Algorithm* `DOM_Partition_2` *is a partition of T.*

Proof: Note that whenever we erase a node v from some tree during iteration i, we make sure to move the corresponding set $\mathcal{C}(v)$ of original nodes to P_{out} or to S. Also note that in the final step (4), each cluster $\mathcal{C}(v)$ in S is either moved to P_{out} or merged into some cluster of P_{out}. In particular, note that in case $\mathcal{C}(v)$ is of size k or smaller, step (4b) always succeeds in finding a neighboring cluster $\mathcal{C}'$ for $\mathcal{C}(v)$. This is because v entered S in step (3c) as a result of remaining as a single node in some tree T'' in T''. Since the input tree was of size $k+1$ or larger, and v represents a cluster $\mathcal{C}(v)$ of fewer than $k+1$ nodes, $\mathcal{C}(v)$ must have some neighboring cluster $\mathcal{C}(w)$ that was already removed from the tree earlier (either in this iteration or in an earlier one). This $\mathcal{C}(w)$ could not have been moved to S, since at that time w still had v (or some "ancestor" of v) as a neighbor. Hence the cluster $\mathcal{C}(w)$ was moved to P_{out}, and thus it can be used in step (4b) of the procedure.

It remains to prove only that at the end of the main loop, the collection T is empty. This is proved as follows. Let s_i denote the minimum size of a cluster $\mathcal{C}_i(v)$ corresponding to a node v occurring in some tree $\tilde{T}$ in $\mathcal{T}_i$, namely,

$$s_i = \min_{\tilde{T} \in \mathcal{T}_i} \min_{v \in \tilde{T}} |\mathcal{C}_i(v)|.$$

By Lemma 3.3 and property (c) in Def. 3.1, s_i at least doubles in every iteration i. Since initially $s_i = 1$, at the end of iteration $i = \lceil \log(k+1) \rceil$ we have $s_{i+1} \geq k+1$, hence $\mathcal{T}_{i+1}$ must be empty. ∎

Lemma 3.6 *If the input tree T is of size $n \geq k+1$, then every cluster C in the output collection P_{out} has the following properties.*

(a) $|C| \geq k+1$.

(b) $Rad(C) \leq 5k+2$.

Proof: Let us first prove Property (a). For the set P_{out} obtained at the end of the main loop, the claim follows immediately by the rule of step (3b). For the final output, note that the claim must remain true since sets in P_{out} can only increase in the last step (through the merge of sets from S), and sets moved from S intact are sufficiently large too.

Let us now turn to Property (b). Consider first a cluster $\mathcal{C}_{i+1}(v)$ from the set P_{out} obtained at the end of the main loop. This cluster was constructed in step (3a). The operation that constructed it consisted of merging some clusters $\mathcal{C}_i(w)$ for $w \in C$. By property (b) in Def. 3.1, the radius of this cluster obeys the relation $Rad(\mathcal{C}_{i+1}(v)) \leq 3r+1$, where $r = \max_{w \in C} Rad(\mathcal{C}_i(w))$. Claim (b) now follows for $\mathcal{C}_{i+1}(v)$ by the fact that the clusters $\mathcal{C}_i(w)$ that formed it had radius k or smaller (since otherwise they would have been removed to P_{out} in some previous iteration), hence $Rad(\mathcal{C}_{i+1}(v)) \leq 3k+1$.

For the final output, note that the merges performed in the last step are shaped as "stars" composed of an original cluster $C \in P_{out}$ merged with a collection of neighboring clusters from S. Each such cluster from S has radius k or smaller (since, again, had its radius been $k+1$ or more, it would have found its way into P_{out}). Thus the radius of each set resulting from such a "star merge" is at most $(3k+1)+2k+1$. The claim follows. ∎

We have established that Algorithm `DOM_Partition_2`(k) constructs the desired $(k+1, O(k))$ spanning forest. However, the construction requires time $O(k \log k \log^* n)$. This is because each application of Procedure `Balanced_DOM` takes time $O(\log^* n)$, except that its distributed implementation on the ith iteration is slowed down by a factor proportional to the maximum diameter of clusters at that iteration. This diameter is bounded by k, hence the total time bound of $\sum_{i \leq \lceil \log(k+1) \rceil} O(k \log^* n) = O(k \log k \log^* n)$.

3.2.3 An $O(k)\log^* n$ time construction

In this section we show how to improve the running time of Algorithm DOM_Partition_2(k) by a factor of $\log k$. Let us first give an intuitive explanation for the extra $\log k$ factor in Algorithm DOM_Partition_2. Note that after the ith iteration the diameter of a cluster may be as high as 4^i (although no greater than k). Thus, after $\frac{1}{2}\log(k+1)$ iterations there may be a cluster whose diameter is about $\frac{k}{2}$. Each of the remaining $\Omega(\log k)$ iterations will thus last $\Omega(k)$ time, which brings the total complexity to $O(k \log k \log^* n)$. Note that many clusters may have grown at a slower pace, and thus $O(\log k)$ iterations may still be necessary.

The idea behind reducing the complexity is to execute each phase i only for $O(2^i)$ time. Clusters that are too large in phase i (namely, whose depth is larger than 2^i) will thus not participate in the phase (except for the 2^i steps required to determine that their depth is indeed too large).

We still face the problem of how to handle a small (hence participating) cluster all of whose neighboring clusters are large (hence non-participating). The solution we employ is to allow such a small cluster to merge onto its larger neighbor, but make sure that this type of merging cannot increase the depth of the formed cluster beyond k. Once a depth of $\Omega(k)$ is reached, the cluster is removed from the tree, as in the previous algorithm, and is not merged onto again until the very last step.

Ensuring the depth bound on the clusters is achieved by maintaining an accurate count of the depth at each node in the cluster. When a small cluster merges onto a large one, it will update the depth counters at each of its nodes, to reflect the depth w.r.t. the new root (which is the root of the large fragment). A leaf in a large cluster will permit other clusters to merge to it only so long as the depth it records is at most k.

We use the following notation. For a node v belonging to a cluster C, let $Depth(v)$ denote the depth of v in C, namely, its distance from the root. (When the algorithm starts, $Depth(v)$ is initialized to zero for every v.) We will use a set $\mathcal{W}$ to keep nodes marked as non-participating (and thus waiting) at a given iteration. Initially, we set $\mathcal{W} = \emptyset$. Algorithm DOM_Partition(k) is obtained by adding the following instructions to Algorithm DOM_Partition_2(k). Inside the main loop, just before step (3a), we add the steps listed in Fig. 7.

An additional modification to Algorithm DOM_Partition_2 is necessary due to the fact that the test performed in step (3b) for the radius of the cluster cannot be implemented directly, since it might cost $\Omega(k)$ time. Instead, this test is implemented by relying on the $Depth$ counters maintained at the nodes. Namely, each node $v \in C$ detecting $Depth(v) > k$ must notify the root of C of that fact. This will still cost $\Omega(k)$ time, but only once, as this cluster will perform step (3b) immediately after that. We do not bring here the pseudo code for this different implementation of the Depth test.

Lemma 3.7 *The collection P_{out} output by Algorithm* DOM_Partition(k) *is a partition of T. Furthermore, if T is of size $n \geq k+1$, then every cluster C in P_{out} has the following properties.*

(a) $|C| \geq k+1$.

(b) $Rad(C) \leq 5k+2$.

(3-I): Return the nodes in the set $\mathcal{W}$ to the forest $\mathcal{T}_i$.
/* This may reunite trees split previously */
(3-II): For every cluster $\mathcal{C}_i(v)$, check whether the cluster's radius is larger than $2 \cdot 2^i$.
If not, then notify all nodes in $\mathcal{C}_i(v)$ that the cluster does participate in iteration i. /* Within $O(2^i)$ time every node in $\mathcal{C}_i(v)$ knows whether $\mathcal{C}_i(v)$ is a participating or a non-participating cluster. */
(3-III): Remove from the trees of $\mathcal{T}_i$ all the nodes u representing non-participating clusters $\mathcal{C}_i(u)$, and place them in the set $\mathcal{W}$.
(3-IV): For every tree T' in $\mathcal{T}_i$, if T' consists of a single node v (representing a lone participating cluster $C' = \mathcal{C}_i(v)$) then do:

(i) Choose a node $u \in \mathcal{W}$ (representing a non-participating cluster $C'' = \mathcal{C}_i(u)$) such that C'' contains a node w that neighbors C', and is of depth $Depth(w) \leq k$ in C''.
/* This node will become the parent of the nodes of C' if C' joins C''. */
If no such cluster C'' exists then erase T' from the forest $\mathcal{T}_i$ and add cluster $\mathcal{C}_i(v)$ to the set S.

(ii) Erase T' (and the node v) from the forest $\mathcal{T}_i$ and move the nodes of C' to the cluster C''.
/* u represents a cluster $\mathcal{C}_i(u) = C' \cup C''$. */

(iii) Use $Depth(w)$ to assign the correct $Depth$ values to every node z in C' (i.e., their new depth in the combined cluster $\mathcal{C}_i(u)$).

Figure 7: Additional section for Algorithm DOM_Partition(k).

Proof: The proof is similar to that of Lemmas 3.6 and 3.5. Note that a node that does not participate in phase i has a size that is at least 2^i. Thus a cluster which reaches size $k+1$ is eventually put in P_{out} as in Lemma 3.5. For clusters moved to S in the additional step (3-IV)(ii), note that by arguments similar to those in the proof of Lemma 3.5, such a cluster must have a large cluster to merge with in P_{out}, since at the point it was moved to S, all the clusters that touched it were either already in P_{out} or non-participating with depth $k+1$ or greater. We conclude that the output is indeed a partition.

Similarly, property (a) follows either as in Lemma 3.6 for a cluster that is put in P_{out} in step (3b), or else it follows immediately by the rule of step (3-IV) parts (ii) to (iv). As for Property (b), note the rule in step (3-IV(ii)). ∎

Lemma 3.8 *Algorithm* DOM_Partition(k) *requires time $O(k \log^* n)$.*

Proof: This is clear from step (3-I), which ensures that iteration i takes only $O(2^i)$ time. ∎

3.3 Fast Distributed Computation of Small k-dominating sets

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

Algorithm Fast_DOM_T

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

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

Proof: By Lemma 2.1 and part (a) of Lemma 3.7, $|D_i| \leq \lfloor \frac{|C_i|}{k+1} \rfloor$. 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 3.10 *The time complexity of Algorithm* Fast_DOM_T *is* $O(k \log^* n)$.

Proof: The complexity of Algorithm Diam_DOM on each cluster C_i is of the order of the diameter of C_i, which is $O(k)$ by property (b) of Lemma 3.7. Hence the complexity of Algorithm DOM_Partition(k) (which is $O(k \log^* n)$ by Lemma 3.8) dominates the total complexity of Algorithm Fast_DOM_T. ∎

Theorem 3.2 *There exists a distributed algorithm for computing a k-dominating set of size at most $n/(k+1)$ on a tree, with time complexity $O(k \log^* n)$.*

4 Distributed Computation of a Small k-Dominating Set on a Graph

In this section we describe how to extend the solution to general graphs. The easy solution of constructing a BFS spanning tree T of the graph G and then computing a small k-dominating set on the tree T has the disadvantage that constructing the spanning tree requires time $O(Diam(G))$. We circumvent this problem by constructing in time $O(k)$ a spanning forest of the graph, composed of large trees, of size $k+1$ or more, or formally, a $(k+1, n)$ spanning forest. Then the solution of the previous section can be applied to each tree separately.

4.1 Constructing a $(k+1, n)$ spanning forest

To construct a $(k+1, n)$ spanning forest for a given graph G, we use a procedure Simple_MST, which we describe below. This procedure is basically a simplified version of the algorithms of [GHS, A2, G]. The minor modifications necessary for the algorithm of [A2] result from the fact that we need to stop the algorithm after $\log(k+1)$ phases, but to ensure that these phases do not last more than $O(k)$ time, while in the [A2] algorithm $O(n)$ time is allowed for $O(\log n)$ phases. Rather than describing the somewhat involved algorithm of [A2] in its entirety along with our modifications, we present a version that is significantly simplified, due to our assumptions that the network is synchronized and that message complexity is ignored. (The number of messages in our algorithm can be reduced, at the expense of simplicity). Algorithms using similar ideas have appeared in the literature also in different contexts (cf. [JM]).

4.2 A High Level description

The high level description of the Simple_MST procedure we use here is similar to the algorithms of [GHS, A2], and the differences concern mainly the termination rule and some implementation details[1].

Nodes group themselves into *fragments* of increasing size. Initially, all nodes are in singleton fragments. Nodes in each fragment F are connected by edges that form a rooted 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 $T(F)$.)

Active fragments grow through merging onto neighboring fragments. A fragment stops participating in this process and becomes *terminated* once it detects that its depth has reached $k+1$. Such a fragment will stop trying to merge onto other fragments, although it will still allow other (smaller, active) fragments to merge onto it.

A fragment F is identified as follows. An active fragment (whose depth is still k or smaller) is identified by the id of its root node. Each node in the fragment knows this identity, as well as the MST edges that touch it. In a *terminated* fragment F (whose depth is $k+1$ or larger), on the other hand, identity is less clear. Nodes close to the root (i.e., within distance k from it) still know its identity. However, more remote nodes know only three things: (a) the MST edges touching them, (b) the fact that the fragment F is terminated, and (c) an outdated fragment id (identifying some small fragment F' they belonged to before F' was consolidated through merging into F), whose main useful property is that it is *different* from the id of any other fragment F'' that is still active in the system.

Fragment merging is performed as follows. The process is carried in phases. In the beginning of a phase, within each active 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 a broadcast (initiated by the fragment root) 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 an active 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 F'.

The final step in the phase involves deciding whether the resulting fragment F' will be active or terminated in the next phase. This is decided by the root of F', which is chosen in a way described later on. The decision is made by performing a process of "broadcast and echo" to depth $k+1$ over the tree, namely, using a *hop counter* in the broadcast message, which is decremented with each forwarding of the message. This allows us to check whether the entire tree

[1] Note that while this procedure superficially resembles also the partitioning algorithm DOM_Partition of the previous section, there are a number of major differences due to the different tasks performed by these two algorithms; for example, here we do not need an upper bound on the size of a fragment.

was covered by the broadcast; if the broadcast reaches a non-leaf node when the hop counter is zero, then the root must be notified that the tree is too deep. This information enables the root to decide whether F' should be active or terminated. At the end of this process, the root broadcasts again to depth k, informing the nodes of the outcome. Nodes not receiving these messages learn that they belong to a terminated fragment.

The process as outlined here requires $O(k \log k)$ time. This is because each phase requires $O(k)$ time for the various broadcast and convergecast processes performed over the fragment trees. In the next subsection we give a more precise description of Procedure `Simple_MST`, resulting in an improved time complexity of $O(k)$, again using an idea similar to [AG, G, CT, A2, JM].

4.3 Procedure `Simple_MST`

The (synchronous) algorithm operates in phases $1, 2, \ldots \log k+1$, where phase i lasts exactly $5 \cdot 2^i + 2$ time units.

In phase i the root first finds out whether the depth of its tree is larger than 2^i. (This is done using a "broadcast and echo" to depth 2^i over the tree, just as outlined in the previous subsection.) If the depth is larger than 2^i then the fragment *halts* and does nothing more in this phase, but it does not terminate, i.e., it may resume its active status in later phases. (We see below that some of the fragment's nodes may receive messages from neighbors not in the fragment and answer them.)

If the root finds that the tree depth is not larger than 2^i (hence the fragment should remain active during this phase), then it broadcasts its identity over the tree. (This can be combined with the previous operation.) Exactly after $2^i + 1$ time units after the beginning of the identity broadcast, each node knows whether its fragment is active or not in the current phase. At this point, each node belonging to an active fragment transmits the identity of its root over all of its edges. By the next time unit each node v receives over some of its edges identities of the corresponding roots. Those edges over which it received an identity equal to that of its own fragment are recognized as *internal* edges, namely, they lead to other nodes in the same fragment. The remaining edges are *outgoing* edges.

Each node (of an active fragment) now selects its minimum (weight) outgoing edge, and convergecasts its weight to the root. During the convergecast, a message containing an edge weight is discarded once a lower weight edge is known. Thus the root learns the minimum weight outgoing edge of the tree 2^i time units after the convergecast started. During the convergecast, each node also remembers the child from which it received each message. Thus the route from the root to the minimum weight outgoing edge can be reconstructed.

Exactly $2^i + 1$ time units after the convergecast started, the root starts a process that transfers the "rootship" to the endpoint of the minimum outgoing edge. Exactly 2^i time units later that endpoint is already the root. The new root then notifies the other endpoint of the minimum outgoing edge that it wishes to connect.

If no message from the other endpoint is received by the next time unit then the fragment became a part of the fragment of the other endpoint. Its root thus ceases to be a root. Otherwise, the other endpoint also notified that it wishes to connect. The endpoint with the highest identity becomes the root of the combined fragment.

4.4 Analysis of the `Simple_MST` Procedure

Since each phase i lasts $O(2^i)$ time, we have

Lemma 4.1 *The* `Simple_MST` *procedure terminates after* $O(k)$ *time.*

Lemma 4.2 *When Procedure* `Simple_MST` *terminates, the collection of resulting fragments forms a* $(k+1, n)$ *spanning forest for the graph. Moreover, each tree of this forest is a fragment of the MST of the graph.*

Proof: The proof is just a simplified version of the proofs of the previous algorithms, e.g. [GHS, A2]. The lower bound of $k+1$ on the size of fragments can be proved inductively, arguing that after phase i, every fragment is of size at least 2^i. This is based on the observations that a fragment at least doubles its size in every iteration in which it is active, and that if some fragment is halted in phase i then its size was greater than 2^i to begin with. The claim follows since there are $\log(k+1)$ iterations. ∎

The properties of our simple MST algorithm are summarized in the following lemma.

Lemma 4.3 *Procedure* `Simple_MST` *terminates after* $O(k)$ *time. Upon termination, the collection of resulting fragments forms a* $(k+1, n)$ *spanning forest for the graph. Moreover, each tree of this forest is a fragment of the MST of the graph.*

4.5 Computing a Small k-dominating Set on a Graph

The fast algorithm `Fast_DOM_G` is now obtained as the combination of the partitioning algorithm `Simple_MST` of Subsection 4.3 and the k-dominating set algorithm `Fast_DOM_T` of Section 3.3. Namely:

Algorithm `Fast_DOM_G`

1. Perform Algorithm `Simple_MST`, yielding a $(k+1, n)$ spanning forest F for G.
2. Perform Algorithm `Fast_DOM_T` on each tree T_i in the spanning forest F.

Theorem 4.4 *There exists a distributed algorithm for computing a k-dominating set of size at most $n/(k+1)$ on a graph, with time complexity $O(k \log^* n)$.* ∎

5 An Application: a Faster MST Algorithm

Our new MST algorithm is composed of two parts. The first part involves simply invoking Algorithm `Fast_DOM_G` of the previous section with parameter $k = \sqrt{n}$. Recall that this algorithm itself is composed of the following three stages: First, execute Procedure `Simple_MST` of Subsection 4.3, constructing a $(\sqrt{n}+1, n)$ spanning forest for G. Each tree in this forest is a fragment of the MST of the graph. Next, apply Procedure `DOM_Partition`$(\sqrt{n})$ on each fragment separately, thus yielding a $(\sqrt{n}+1, O(\sqrt{n}))$ spanning forest for G, with each tree in this forest still being a fragment of the

MST. Finally, apply Procedure Diam_DOM on each fragment separately[2].

The output of Algorithm Fast_DOM_G is a $\sqrt{n}$-dominating set D of size at most $\sqrt{n}$, and an accompanying partition P of the graph into clusters of radius at most $\sqrt{n}$ around each of the nodes in D. However, note that each of these clusters is still spanned by a fragment of the MST, and these fragments were constructed as a byproduct of the algorithm. In particular, each node belonging to some cluster C in P knows which edges belong to the MST fragment spanning C, and also knows the identity of the fragment's root.

Consider the *fragment graph* $\tilde{F}$, whose vertex set $V(\tilde{F})$ is the collection of clusters in P, containing $N = O(\sqrt{n})$ 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 describe the second part of our MST algorithm, whose role is to reduce the total number of remaining inter-fragment edges to the necessary $N-1$.

5.1 Global Edge Elimination by Pipelining

5.1.1 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.

Our pipelined procedure is given in Fig. 8.

5.1.2 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 5.1 *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 5.2 *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 round t, for some $t \geq L$. Note that each of the children of v in the tree is of level $L-1$ or lower, hence by Lemma 5.2 they started

[2]This stage is not essential for the purposes of the current section.

1. Build a BFS tree B on G, with a root $r(B)$.
2. Throughout the execution, each node v in B maintains a set Q of all 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 its adjacent inter-fragment edges. It also maintains a set U (initially empty) of all 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, v 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.
It then broadcasts the resulting set S (over B) to all nodes in G.

Figure 8: Procedure Pipeline.

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 5.3 *(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 5.1, 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 5.1.

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

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

which is a 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 5.4 *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. We prove this fact by induction on the structure of the tree, starting from the leaves upwards. The inductive step follows directly from claim (a) of Lemma 5.3, 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 5.5 *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 the fact that the trees constructed in the first stage were fragments of the MST, 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. ∎

5.2 Algorithm `Fast_MST`

Combining the two parts, we get the following distributed algorithm for MST.

1. Perform Algorithm `DOM_G` for $k = \sqrt{n}$.
2. Perform Algorithm `Pipeline`.

Theorem 5.6 *There exists a distributed Minimum-weight Spanning Tree algorithm with time complexity $O(\sqrt{n}\log^* n + Diam(G))$.* ∎

Proof: By Theorem 4.4 and the choice of $k = \sqrt{n}$, the execution time of the first stage of Algorithm `Fast_MST` is $O(\sqrt{n}\log^* n)$. By Lemma 5.5 the time complexity of the second stage of Algorithm `Fast_MST` is bounded by $O(\sqrt{n}) + Diam(G)$. The claim follows. ∎

References

[AG] Y. Afek and E. Gafni, Time and message bounds for election in synchronous and asynchronous complete networks, *Proc. 4th Symp. on Principles of Distributed Computing*, 1985, 186–195.

[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.

[A1] Baruch Awerbuch, Complexity of network synchronization, *J. ACM*, Vol. 32, (1985), 804–823.

[A2] 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. 25th Symp. on Theory of Computing*, San Diego, California, pp. 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.

[BKP] J. Bar-Ilan, G. Kortsarz and D. Peleg, How to Allocate Network Centers, *J. of Algorithms*, Vol. 15, (1993), 385–415.

[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, *34th 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*, 1987.

[JM] D.B. Johnson and P. Metaxas, Connected components in $O(\lg^{3/2}|V|)$ parallel time for the CREW PRAM, *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991, 688–697.

[L] N. Linial, Distributive graph algorithms - global solutions from local data, *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987, pp. 331–335.

[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.