Efficient Distributed Approximation Algorithms via Probabilistic Tree Embeddings *

Maleq Khan † Fabian Kuhn ‡ Dahlia Malkhi § Gopal Pandurangan ¶

Kunal Talwar §

Abstract

We present a uniform approach to design efficient distributed approximation algorithms for various network optimization problems. Our approach is randomized and based on a probabilistic tree embedding due to Fakcharoenphol, Rao, and Talwar [10] (FRT embedding). We show how to efficiently compute an (implicit) FRT embedding in a decentralized manner and how to use the embedding to obtain expected $O(\log n)$-approximate distributed algorithms for the generalized Steiner forest problem, the minimum routing cost spanning tree problem, and the k-source shortest paths problem in arbitrary networks. The time complexities of our algorithms are within a polylogarithmic factor of the optimum.

The distributed construction of the FRT embedding is based on the computation of least elements (LE) lists, a distributed data structure that might be of independent interest. Assuming a global order on the nodes of a network, the LE list of a node stores the smallest node (w.r.t. the given order) within every distance d (cf. Cohen [3], Cohen and Kaplan [4]). Assuming a random order on the nodes, we give an almost-optimal distributed algorithm for computing LE lists on weighted graphs. For unweighted graphs, our LE lists computation has asymptotically optimal time complexity $O(D)$, where D is the diameter of the network. As a byproduct, we get an improved synchronous leader election algorithm for general networks.

Keywords: Distributed Approximation, Generalized Steiner Forests, Least Element Lists, Metric Spaces, Network Optimization, Probabilistic Tree Embeddings

---

*This paper will appear in the 27th Annual ACM Symposium on Principles of Distributed Computing (**PODC**), 2008, Toronto, Canada.

†Network Dynamics and Simulation Science Laboratory, VBI, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA. E-mail: maleq@vbi.vt.edu

‡Institut f. Theoretische Informatik, ETH Zurich, Universitötstrasse 6, 8092 Zurich, Switzerland. E-mail: kuhn@inf.ethz.ch

§Microsoft Research, Mountain View, CA 94043, USA. E-mail: {dalia, kunal}@microsoft.com

¶Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA. E-mail: gopal@cs.purdue.edu

1 Introduction and Related Work

The emerging area of distributed approximation algorithms lies at the intersection of two well-established theoretical computer science areas: distributed computing and approximation algorithms. Distributed approximation algorithms trade-off optimality of the solution for the amount of resources (messages, time) consumed by the distributed algorithm. Besides a fundamental theoretical interest in understanding the algorithmic complexity of distributed approximation, there also is a practical motivation in studying distributed approximation algorithms. Emerging networking technologies such as wireless ad hoc and sensor networks or peer-to-peer networks operate under inherent resource constraints such as energy, bandwidth, etc. Moreover, the topology of such networks often changes dynamically. Hence it becomes critical to design efficient distributed algorithms for various network optimization problems that have low communication (message) and time complexity even if this comes at the cost of a reduced quality of the solution. For this reason, in the distributed context, devising approximation algorithms is well motivated even for network optimization problems that are not NP-hard. There is a large body of work on distributed approximation algorithms for classical graph optimization problems such as minimum spanning tree, shortest path, minimum dominating set and vertex cover, and maximum matching. [6, 9, 13, 14, 15, 16, 17, 18]. We also refer to the surveys by Elkin [7] and Dubhashi et al. [5] that summarize many results regarding efficient distributed approximation algorithms and the hardness of distributed approximation (in particular, lower bounds on the time-approximation trade-off) for various classical optimization problems.

When dealing with a certain class of metric optimization problems, a standard approach to obtain approximation algorithms is to use low-distortion metric embeddings. First, the given metric space is embedded into a host space with a simpler structure (e.g., an ℓ_p-metric or a tree metric). If one can approximately, or exactly solve the optimization problem on the host metric, one gets an approximation algorithm for the problem on the original metric. The approximation ratio suffers depending on the quality (i.e. the distortion) of the embedding. While using embeddings to obtain approximation algorithms is a standard approach in a non-distributed context, to the best of our knowledge, the technique has not been used in distributed approximation.

In this paper, we develop a distributed version of a probabilistic tree embedding due to Fakcharoenphol, Rao, and Talwar [10] (henceforth referred to as the FRT embedding). The FRT embedding embeds any n-point metric into a distribution over dominating tree metrics such that the expected stretch of any edge is $O(\log n)$. Moreover, in [10] it is shown how to efficiently sample a tree from this distribution. This directly leads to polylogarithmic approximation algorithms for a variety of optimization problems (see [10] and references therein for a detailed list). The stretch bound of the FRT embedding is existentially tight: There are metric spaces for which any probabilistic tree embedding has distortion $\Omega(\log n)$.

The FRT embedding algorithm of [10] is centralized. We show how to efficiently compute an (implicit) FRT embedding in a distributed manner and how to use the embedding to obtain distributed algorithms for the generalized Steiner forest problem, the minimum routing cost spanning tree problem, and the k-source shortest paths problem with expected approximation ratio $O(\log n)$. The time complexities of our algorithms are existentially optimal up to polylogarithmic factors. Our algorithms are the first known distributed approximation algorithms for the generalized Steiner forest and the minimum routing cost spanning tree problems. Our algorithm for k-source shortest paths is significantly more time-efficient than previous algorithms at the cost of giving a logarithmic approximation.

For our distributed construction of the FRT embedding, we need a distributed data structure called least elements (LE) lists that was first described by Cohen in [3] and by Cohen and Kaplan in [4]. Assuming a global order on the nodes of a network, the LE list of a node stores the smallest node (with respect to the given order) within every distance d. For arbitrary weighted graphs, we give an almost time-optimal

distributed algorithm for computing LE lists, assuming a random order on the nodes. As a byproduct, we get an improved leader election algorithm in general networks that is both time-optimal and, with high probability, almost message-optimal. Since LE lists are useful in various contexts ([3, 4]), we believe that this result can have other applications as well.

The main application of our distributed FRT embedding construction is a fast distributed approximation algorithm for the generalized Steiner forest (GSF) problem (cf. Section 4.1): Given a weighted graph $G = (V, E, w)$, and a collection of k disjoint subsets (groups) of V: $V_1, V_2, \ldots, V_k$, the problem is to find a minimum weight subgraph in which each pair of nodes belonging to the same group V_j is connected. The GSF problem is a generalization of the minimum Steiner tree problem (where $k = 1$) which in turn is a generalization of the minimum spanning tree (MST) problem, both of which are fundamental problems in distributed computing and communication networks. There is a long line of research on time-efficient distributed algorithms for the (exact) MST problem (see e.g. [11, 1, 19, 8]). The best known algorithms take $O(D + \sqrt{n} \log^* n)$ time [19, 8] (where D is the network diameter). In [22], it is shown that this is existentially tight up to polylogarithmic factors in the $\sqrt{n}$ term.

Distributed computations of approximate MST solutions were first considered by Elkin in [9]. He showed that approximating the MST within a ratio of H requires $\Omega(D + \sqrt{n/(HB \log n)})$ time (assuming B bits can be sent through each edge in one round). Note that this lower bound applies for the more general minimum Steiner tree and GSF problems as well. In [15], Khan and Pandurangan presented a fast distributed approximation algorithm that constructs an $O(\log n)$-approximate minimum spanning tree. Up to polylogarithmic factors, the time complexity of the algorithm of [15] is optimal (i.e., no $O(\log n)$-approximate algorithm can in general do better). The algorithm of [15] can be easily modified to yield an $O(\log n)$-approximation for the minimum Steiner tree problem as well. When applied to the more general GSF problem, the technique however only yields an $O(k \log n)$-approximation, where k is the number of disjoint sets. The question of getting a time optimal $O(\log n)$ distributed approximation was left open and this was a motivation for our new approach in this paper.

1.1 General Notations and Definitions

Throughout the paper, we use the following definitions and notations concerning an undirected weighted graph $G = (V, E, w)$ with $|V| = n$ nodes, $|E| = m$ edges, and non-negative edge-weights $w(e)$ for all edges $e \in E$.

- $Q(u, v)$: a path from node u to node v
- $|Q(u, v)|$, $|Q|$: number of edges on path $Q(u, v)$
- $w(Q(u, v))$, $w(Q)$: sum of the edge weights of $Q(u, v)$
- $P(u, v)$: a minimum weight path from u to v
- $d(u, v) = w(P(u, v))$: (weighted) *distance* between u and v
- $l(u, v) = |P(u, v)|$: number of edges on shortest path
- $D = \max_{u,v} \min_Q |Q(u, v)|$: *diameter* of G
- $\Delta = \max_{u,v} w(P(u, v))$: *weighted diameter* of G
- $\Gamma_\rho(v) = \{u \mid d(u, v) \leq \rho\}$: *ρ-neighborhood* of node v

We also call the number of edges $|Q|$ of a path Q, the *length* of Q. Similarly, $w(Q)$ is called the *weight* of the path Q. If there is more than one minimum weight path $P(u, v)$ between u and v, $l(u, v)$ denotes the length of the shortest (w.r.t. number of hops) such path.

Definition 1.1 ***Shortest Path Diameter (SPD).*** *The SPD is denoted by $S(G, w)$ (or S for short) and defined as $S = \max_{u,v \in V} l(u, v)$. Note that $1 \leq D \leq S \leq n$ for every connected graph and $S = D$ for unweighted graphs.*

We use the term "with high probability" (WHP) to mean with probability at least $1 - 1/n^c$ for $c \geq 1$.

1.2 Distributed Computing Model

Without loss of generality, we assume that G is connected. We assume that each node has a unique identifier and that at the beginning of the computation, each node v accepts as input its own identifier and the weights of the edges adjacent to v. We assume a synchronous message passing model [21]. In one time step, a node v can send an arbitrary message of size at most $O(\log n)$ through each of its edges. The weights of the edges are assumed to be at most polynomial in the number of nodes n, and therefore, the weight of a single edge can be communicated in one time step (we assume that all weights are normalized to be at least 1). Such a restriction on the message size is quite natural as the bandwidth of single communication channels is typically bounded in practical applications. Moreover, note that without restricting the maximum message size, a single convergecast would suffice to send all input information to a single node and do all the computation locally at this node. The model we use is a standard model of distributed computation called the (synchronous) CONGEST($\log n$) model or simply the CONGEST model [21] and has been used by all previous distributed MST algorithms (e.g., [11, 12, 19, 1, 8]) as well as the distributed MST approximation algorithm of [15]. It would be straightforward to adapt our algorithms the a more general CONGEST(B) model where $O(B)$ bits can be transmitted in a single message. By using a *synchronizer*, our algorithms can also be made to work in an asynchronous system under the same time bounds, but at the cost of some increase in the message complexity [21]. Finally, although we assume that local computations (at a node) are for free, our algorithms only perform local operations that are polynomial in n.

1.3 Overview of the Results

Distributed Computation of LE Lists:
Assuming that the nodes are ranked according to a random order, we give an almost time-optimal distributed algorithm for computing LE lists (cf. Section 2). In weighted graphs, our algorithm terminates in $O(S \log n)$ time (i.e., all nodes will detect termination within this time) with a message complexity of $O(S|E|\log n)$ where S is the shortest path diameter. It is not hard to see that $\Omega(S)$ is a lower bound on the time complexity of any distributed algorithm that computes LE-lists (even when assuming a random node ordering) and hence the time complexity is optimal up to a logarithmic factor. For unweighted graphs, our algorithm terminates in $O(D)$ rounds and WHP, uses at most $O(|E| \min(D, \log n))$ messages. The algorithm for unweighted graphs can be used as a randomized, synchronous leader election algorithm by selecting the node with the smallest rank as leader. An important consequence is an improved leader election algorithm in arbitrary (synchronous) networks which improves over the previous best known result due to Peleg [20]. Peleg's algorithm takes $O(D)$ time and $O(D|E|)$ messages, while our algorithm takes $O(D)$ time (deterministically) and $O(|E| \min(D, \log n))$ messages with high probability. Peleg [20] raised an important open question of whether there exists an algorithm that achieves both optimal message complexity $\Omega(|E|+n \log n)$ and optimal time complexity $O(D)$ and our result makes progress on this question.

In prior work, Cohen [3] presented a centralized strategy for computing LE-lists that incurs $O(|E| \log n)$ memory operations. The strategy in [3] requires nodes to spread their rank values sequentially in ascending order. This is not trivial to emulate in a distributed setting. Cohen and Kaplan [4] compute LE list in a distributed setting. However, their protocol still assumes sequential spreading of rank values, only their order is determined in a distributed manner.

Distributed Approximation Algorithms:
We use LE lists to implicitly compute an FRT embedding. Using the embedding, we obtain distributed approximation algorithms for the following problems (cf. Section 4):

1. *Generalized Steiner Forest (GSF) Problem:* We give a distributed algorithm takes $O(S k \log^2 n)$ time and $O(S n \log n)$ messages and computes an (expected) $O(\log n)$-approximate Steiner subgraph. The parameter S can be shown to capture the hardness of distributed approximation quite precisely. Using the hardness results of Elkin [9], one can show that there exists a family of n-node graphs where $\Omega(S)$ time is needed by any distributed approximation algorithm to approximate the MST within an H-factor, for any $H \in [1, O(\log n)]$ (we refer to [15] for details). Since the MST problem is a special case of the GSF problem, our algorithm is existentially time-optimal (up to a factor of $O(k \log n)$). Note that S can be much smaller than $\sqrt{n}$. (Recall that $\tilde{\Omega}(\sqrt{n})$ is a lower bound on the time needed to compute exact MST or Steiner tree solutions.) For example, in networks where edge weights are chosen uniformly and independently from an arbitrary distribution, $S = O(D + \log n)$ with high probability [15].

No previous distributed approximation algorithm for the GSF problem is known. For the special case of the minimum Steiner tree problem ($k = 1$), there is a well-known centralized 2-approximation algorithm based on computing the MST on the node set V_1 [23]. Using this fact, an $O(n \log n)$-time 2-approximate distributed algorithm was designed in [2] based on the classical distributed MST algorithm due to Gallager et al [11]. This algorithm is not time optimal (since Gallager et al. also is not). Also, the approach cannot be generalized to the GSF problem.

2. *Minimum Routing Cost Spanning Tree Problem:* In this problem, the weight of an edge represents the cost of routing messages between its endpoints. The *routing cost* for a pair of nodes in a given spanning tree is the sum of the weights of the edges of the unique tree path connecting the two nodes. The routing cost of the tree then is the sum of the pairwise routing costs over all pairs of nodes. Finding a spanning tree with minimum routing cost is NP-hard for general weighted undirected graphs. We refer to [24] for a detailed background. We present an (expected) $O(\log n)$-distributed approximation algorithm that takes $O(S \log^2 n)$ time and $O(S n \log n)$ messages.

3. *k-Source Shortest Paths Problem:* Formally, given an undirected weighted graph $G = (V, E, w)$ and a subset $K \subseteq V$ of k nodes, the goal is to compute (approximate) shortest paths between all pairs of nodes in $V \times K$. We refer to [6] for a detailed background on this fundamental algorithmic problem. We give an algorithm that computes (expected) $O(\log n)$-approximate k-source shortest paths in $O(kD \log n)$ time using $O(|E|(\min[D, \log n] + k) + kn \log n)$ messages in an unweighted graph and in $O(kS \log n)$ time using $O(|E|(S \log n + k) + kn \log n)$ messages in a weighted graph.

In [6], Elkin presented a distributed algorithm that computes a path that is $(1 + \epsilon)$ times the shortest path plus an additive constant. On unweighted graphs, the algorithm runs in $O(kD + n^{1+\delta/2})$ time and uses $O(|E|n^{\rho} + kn^{1+\delta})$ messages (in the synchronous model) where ϵ, ρ, and δ are arbitrarily small positive constants. On weighted graphs, the algorithm runs in $O(w_{max} n^{1+\delta/2} + kD)$ time using $O(|E|n^{\rho} + kn^{1+\delta})$ messages and the additive error depends on w_{max}, the ratio between the largest and smallest weight in the network. Since the algorithm's time complexity depends on the edge weights, its running time can be quite large. Our result substantially improves the time complexity of Elkin's algorithm both for weighted and unweighted graphs. This however comes at the cost of an $O(\log n)$-factor in the quality of the solution. Our algorithm also improves over the communication complexity of exact algorithms (e.g. [6]).

2 Least Element Lists

At the core of our approximation techniques is a distributed protocol called *LE-Dist*. Given an undirected weighted network $G = (V, E, w)$, LE-Dist assigns to each node v a unique rank $\mathbb{R}(v)$ drawn uniformly at random, and computes Cohen's LE-Lists [3] for all nodes.

The *least element* in $\Gamma_{\rho}(v)$, denoted by $L_{\rho}(v)$, is a node $u \in \Gamma_{\rho}(v)$ such that for all $u' \in \Gamma_{\rho}(v)$ and $u' \neq u$, $\mathbb{R}(u) < \mathbb{R}(u')$. For every node $v \in V$, we want to compute the least element in $\Gamma_{\rho}(v)$ for every

distance $\rho \in [0, \Delta]$. These least elements are maintained as a list of ordered pairs, called the *least-element list (LE-list)*:

Definition 2.1 ***LE-list.*** *The LE-list of a node $v \in V$, $\mathbb{L}(v) = \{\langle u, \rho \rangle \mid \rho = d(v, u)$ and $u = L_\rho(v)\}$.*

Several properties follow from the definition of LE-lists. Let $\langle u_i, \rho_i \rangle$ be the ith element in the sorted order of the elements of $\mathbb{L}(v)$ in increasing order of ρ, i.e., $\rho_i < \rho_{i+1}$ for $1 \le i < |\mathbb{L}(v)|$. We have (a) $L_\rho(v) = u_i$ for any $\rho \in [\rho_i, \rho_{i+1})$, for $1 \le i \le |\mathbb{L}(v)|$ assuming $\rho_{|\mathbb{L}(v)|+1} = \Delta + \epsilon$ with any $\epsilon > 0$, (b) $\mathbb{R}(u_i) > \mathbb{R}(u_{i+1})$ for $1 \le i < |\mathbb{L}(v)|$, and (c) $u_{|\mathbb{L}(v)|}$ is the least element in V.

2.1 The LE-Dist Algorithm

We present a distributed algorithm called *LE-Dist* for computing the LE-lists of all nodes. The nodes first choose their ranks randomly. The basic strategy is to let each node flood its rank to the network. A node v needs to forward a rank message from w to its neighbors only if w is the lowest ranking node at distance $d(v, w)$ from v.

All nodes execute the algorithm simultaneously in phases, where a phase consists of several communication rounds. Initially, a node only knows its own rank and distance (0). In each phase, a node exchanges rank information with its neighbors. The node updates its knowledge of ranks in the network based on the messages it receives from neighbors. These changes are forwarded to neighbors in the next phase. It is easy to see that after phase k, a node has rank information regarding all nodes within hop-count k. Thus, after S phases (S is the shortest path diameter of G, defined in Section 1.1), every node has full rank information and can compute $\mathbb{L}(v)$.

Note that, in a weighted graph, at the time that node v gets w's rank, v may not have information about all the nodes within distance $d(v, w)$. So w's rank message may get forwarded by v, and later, v may receive a rank message from a lower ranking node within distance $d(v, w)$. This may result in having extraneous messages being forwarded, and also in extra delays, since only one message may be sent over each link per time step. The challenge is to keep the communication and completion time bounded. Subsequent sections address the time and message analysis.

More specifically, a node locally v maintains a data structure le_v which stores v's current view of $\mathbb{L}(v)$. Initially, it contains the LE-list of $\{v\}$, the node itself. At the beginning of round k, it stores the LE-list for v of all nodes at hop-distance $k - 1$ from v. For convenience, each entry in le_v contains, in addition to the pair $\langle u, R(u) \rangle$, the following information:

- ρ, the minimal distance that u's rank message traveled to reach v;
- u', the last node on the minimal-distance path from u to v;
- a new/old flag.

The details for the LE-Dist algorithm are given by Algorithm 1. In an initialization step, the nodes first choose random ranks. Choosing $p(v)$ from $\{1, \ldots, n^c\}$ guarantees that all nodes v choose different $p(v)$ WHP. Initially, le_v contains the single entry $\langle v, \mathbb{R}(v), 0, v, new \rangle$. After the initialization, the algorithm runs in phases. In phase k, node v exchanges all items marked *new* in its list with its neighbors, and incorporates their rank messages into le_v. At the end of a phase, a node v proceeds to the next phase as long as it has not received a control message notifying it of termination. The control messages needed for termination are omitted from the pseudo-code of the algorithm. We discuss termination detection in the following section.

Algorithm 1 LE-Dist Algorithm (at node v)

Initialization:

1: choose $p(v)$ uniformly at random from $\{1, \ldots, n^c\}$
2: rank $\mathbb{R}(v) := (p(v), \mathrm{ID}(v))$
3: *// use lexicographical order when comparing ranks*
4: insert $\langle v, \mathbb{R}(v), 0, v, new \rangle$ into le_v

Phases:

5: *// execute phase code until termination message received*
6: **for** phase $k := 1, 2, \ldots$ **do**
7:    **for** each *new* item $\langle u, \mathbb{R}(u), \rho, u', new \rangle$ in le_v **do**
8:       **send** *rank* msg. $\langle u, \mathbb{R}(u), \rho \rangle$ to all neighbors except u'
9:    **end for**
10:    **send** end-of-phase control msg. to all neighbors
11:    mark the *new* items in le_v old
12:    wait until end-of-phase msg. received from all neighbors
13:    **for all** neighbors u' **do**
14:       **for all** *rank* msg. $\langle u, \mathbb{R}(u), \rho \rangle$ rcv. from neighbor u' **do**
15:          $\rho' := \rho + w(u', v)$
16:          **for all** $\langle u_i, \mathbb{R}(u_i), \rho_i, u'_i, * \rangle \in \mathsf{le}_v$ **do**
17:             **if** $\rho_i \leq \rho'$ and $\mathbb{R}(u_i) \leq \mathbb{R}(u)$ **then**
18:                delete (ignore) *rank* message $\langle u, \mathbb{R}(u), \rho \rangle$
19:             **else**
20:                **for all** $\langle u_i, \mathbb{R}(u_i), \rho_i, u'_i, * \rangle \in \mathsf{le}_v$ **do**
21:                   **if** $\rho_i \geq \rho'$ and $\mathbb{R}(u_i) \geq \mathbb{R}(u)$ **then**
22:                      delete $\langle u_i, \mathbb{R}(u_i), \rho_i, u'_i, * \rangle \in \mathsf{le}_v$ from le_v
23:                   **end if**
24:                **end for**
25:                insert $\langle u, \mathbb{R}(u), \rho', u', new \rangle$ into le_v
26:             **end if**
27:          **end for**
28:       **end for**
29:    **end for**
30: **end for**

2.2 Detecting Termination

As we already argued above, after phase S, every node v has rank information from the entire network, and le_v does not change any more. Therefore, no more rank messages are generated after phase S ends. Since S is not known to the nodes initially, relying on the *rank* messages only, a node cannot determine whether there are any outstanding *rank* messages in the network and does not know when to terminate.

To detect termination, we introduce an additional mechanism. If v ignores a rank message it receives from u', it *echo*es it back. Otherwise, it waits for echoes from all neighbors other than u', and then echoes it back. Note that, if u' is v's only neighbor, then the waiting vacuously completes immediately, and v echoes back the message.

Every rank message is eventually echoed back, either when it cannot travel farther or when it is ignored. At the latest, after the end of Phase S, any remaining rank message is echoed. It may take up to S additional phases for echoes to reach their origins. After the end of Phase $2S$, every node has received echoes for its

own rank message from all of its neighbors. Still, nodes do not know when all of their peers have received their echoes.

Consider a node v whose rank is not the lowest. Its rank message is ignored at least by one other node (the lowest ranking). A node that ignores v's rank message forwards at the next phase information about a lower rank. That lower rank, or an even lower rank message that collides with it, reaches v at the latest one phase after the echo message reaches v. Therefore, a node w that receives its own echoes from all of its neighbors at phase r, and does not receive any lower rank message at phase $r+1$, knows that it is the lowest ranking node in the network. Node w then floods the network with a *leader* message. A node that receives a leader message stores the last (parent) hop, and forwards the leader message to all other neighbors. This builds a BFS tree rooted at the lowest ranking node w.

After having collected *echo* message for its rank message from all of its neighbors, a *leader* message, and *done* messages from every child neighbor (all but parent), a node sends a *done* message to the parent neighbor. The leader waits for *done* messages from all of its children, and then floods the network with a *termination* signal.

More precisely, in a phase, after having exchanged rank messages as described above, a node v exchanges messages with neighbors as follows.

Echo:

a) When v ignores a *rank* message $\langle u, \mathbb{R}(u), \rho \rangle$ received from a neighbor u' (see Step 2(e)i. above), it sends an *echo* message, for this *rank* message, back to u'.

b) If v forwards the *rank* message $\langle u, \mathbb{R}(u), \rho \rangle$ (Step 2a), it waits until it receives the *echo* messages for this *rank* message from its neighbors (except u', whom v did not forward the *rank* message to). Once v receives these *echo* messages, if v did not send an echo u' already, v sends an *echo* message to u'. If v does not have any neighbor other than u', i.e., there is no neighbor to forward $\langle u, \mathbb{R}(u), \rho \rangle$ to, v sends the *echo* immediately to u'.

c) If an item $\langle u_i, \mathbb{R}(u_i), \rho_i, u'_i, * \rangle$ gets deleted from le_v (Step 2(e)ii), and if v did not send the echo to u'_i for this *rank* message yet, v sends the echo to u'_i.

One full phase after having received echo messages for its own rank from all of its neighbors, a node v starts the following termination protocol.

Termination:

a) If there are no lower rank items in le_v, then v knows it is the leader and sends *leader* message to all its neighbors. It waits for *done* messages back from them, and then sends a *termination* signal to all of them.

b) If v receives a leader message from u, and $parent_v$ is empty, then v sets $parent_v$ to u and forwards the leader message to every neighbor except u.

c) If $parent_v$ is not empty, and v received *done* messages from every neighbor except $parent_v$ (vacuously holds if $parent_v$ is the only neighbor), then v sends a done message to $parent_v$.

2.3 Correctness

By applying the following lemma recursively, one can show that using the LE-lists as "routing tables", a node v can trace back a path (which is the weighted shortest path) to the lowest ranked node in $\Gamma_\rho(v)$ for any ρ.

Lemma 2.1 *At the end of the LE-Dist algorithm (Algorithm 1), if* $\langle u, \mathbb{R}(u), \rho, u', *\rangle \in \mathsf{le}_v$*, then* $\langle u, \mathbb{R}(u), \rho - w(u', v), u'', *\rangle \in \mathsf{le}_{u'}$ *for some* u'' *such that* $u'' \neq v$ *and* $(u', u'') \in E$.

Proof: $\langle u, \mathbb{R}(u), \rho, u'\rangle, *$ can be inserted in le_v only after v receives *rank* message $\langle u, \mathbb{R}(u), \rho - w(u', v)\rangle$ from neighbor u', and at that time $\langle u, \mathbb{R}(u), \rho - w(u', v), u''\rangle, *$ is in le'_u. Clearly $u'' \neq v$, because if u' receives this message from v, u' would not forward it to v. Later $\langle u, \mathbb{R}(u), \rho - w(u', v), u'', *\rangle$ can be deleted from le'_u only if u' receives another message $\langle u_1, \mathbb{R}(u_1), \rho_1\rangle$ from u'_1 with $\mathbb{R}(u_1) \leq \mathbb{R}(u)$ and $\rho - w(u', v) \leq \rho_1 + w(u', u'_1)$. If $u'_1 \neq v$, u' would forward this message to v leading to the deletion of $\langle u, \mathbb{R}(u), \rho, u', *\rangle$ from le_v, and if $u'_1 = v$, v would delete it before forwarding the message to u'. □

Lemma 2.2 *By the time, a node u receives the echoes of the rank message originated by itself from all of its neighbors, the rank message* $\langle u, \mathbb{R}(u), \rho\rangle$ *has reached every node v for which* $\langle u, \mathbb{R}(u), \rho, *, *\rangle$ *is supposed to be in* le_v.

Proof: In Steps 3a and 3b, the *echo* for a *rank* message is generated only when the *rank* messages is not forwarded, and if it is forwarded (by v), v waits until the *echo* messages comes back from the neighbors to which v forwarded the message to. Thus, clearly, the *rank* messages (originated by u) complete their journey before u receives the echoes back. In Step 3c, v sends an echo early if an item $\langle u, \mathbb{R}(u), \rho, *, *\rangle$ is deleted from le_v. Consider any node v' such that the *rank* message $\langle u, \mathbb{R}(u), *\rangle$ travels from u to v' via v. Since $\langle u, \mathbb{R}(u), \rho, *, *\rangle$ is deleted from le_v, following Lemma 2.1, the item $\langle u, \mathbb{R}(u), \rho, *, *\rangle$ corresponding to the *rank* message $\langle u, \mathbb{R}(u), \rho\rangle$ cannot be in the "final" LE-list of v'. Thus, the lemma follows. □

Lemma 2.3 *If a node u receives the echoes of its own rank message from all of its neighbors by the end of phase k, then by the end of phase* $k + 1$ *it has a lower ranking entry in* le_v *if there exists a lower rank in the graph.*

Proof: Consider a node v whose rank is not the lowest. Its rank message is ignored at least by one other node (the lowest ranking). A node that ignores v's rank message forwards at the next phase information about a lower rank. That lower rank, or a lower rank message that collides with it, reaches v at the latest one phase after the echo message reaches v. □

Lemma 2.4 *At the end of Algorithm 1,* $\langle u, \mathbb{R}(u), \rho, u', *\rangle \in \mathsf{le}_v$ *if and only if* $\rho = d(v, u)$ *and* $u = L_\rho(v)$.

Proof: When the algorithm terminates, every node v will have received echoes for its own rank message from all its neighbors. Therefore, by Lemma 2.2, for every node u, such that u in le_v, u's rank message has arrived at v. It remains to show that le_v is computed correctly from all such messages. This follows from lines 16–27 of the algorithm. Notice that a *rank* message, originated by $u = L_\rho(v)$, that follows the shortest path $P(u, v)$, meaning $\rho = d(v, u)$, cannot be ignored or deleted by an intermediate node in path $P(u, v)$ or by v itself. On the other hand, if it does not follow $P(u, v)$ (i.e., $\rho > d(v, u)$), its deletion is caused, if it is not deleted yet, by another *rank* message originated by u that follows $P(u, v)$. If $\rho = d(v, u)$ but $u \neq L_\rho(v)$, i.e., there exist $u_1 \in \Gamma_\rho(v)$ with $\mathbb{R}(u_1) < \mathbb{R}(u)$, the deletion is caused by the *rank* message originated by u_1 that follows $P(u_1, v)$. □

Theorem 2.5 *The LE-Dist algorithm correctly computes the LE-list of each node.*

2.4 Analysis for Weighted Graphs

In this section, we provide an analysis of the number of rounds and messages needed for completion. We will make use of the following notation in our proofs: $\mathsf{le}_v(k)$ denotes the LE-list le_v at the end of phase k.

Lemma 2.6 *$E[|\mathsf{le}_v(k)|] \leq \log n$, and simultaneously for all $v \in V$ and all phases k, $|\mathsf{le}_v(k)| \leq O(\log n)$ WHP.*

Proof: Let $H_k(v)$ be the set of nodes within k-hop distance from v, i.e., $H_k(v) = \{u \mid \exists Q(v,u) \text{ s.t. } |Q(v,u)| \leq k\}$. The graph G restricted to nodes in $H_k(v)$ and edges connecting them is denoted $G_k(v)$. Let $M_k(v)$ be a set of ordered pairs defined as $M_k(v) = \{\langle u, \rho\rangle \mid u \in H_k(v) \text{ and } \rho = \min_{|Q|\leq k} w(Q(v,u))\}$. We have $|M_k(v)| = |H_k(v)| \leq n$. Now, consider a sorted order of the elements in $M_k(v)$ in non-decreasing order of ρ, and let $\langle u_i, \rho_i\rangle$ be the ith element in this sorted order, where $1 \leq i \leq |M_k(v)|$. It is easy to see that the LE-list of v on the restricted graph G_k is equal to the set $\mathbb{L}_{M_k}(v) = \{\langle u_i, \rho_i\rangle \mid \mathbb{R}(u_j) > \mathbb{R}(u_i) \text{ for all } j < i\}$. The list constructed by the algorithm at the end of phase k at node v is the LE-list of v on G_k. Hence, $\mathsf{le}_v(k)$ is $\mathbb{L}_{M_k}(v)$.

Let ξ_i be the event that $\langle u_i, \rho_i\rangle \in \mathbb{L}_{M_k}(v)$. Event ξ_i occurs if and only if $\mathbb{R}(u_i)$ is the lowest among $\mathbb{R}(u_1), \mathbb{R}(u_2), \ldots, \mathbb{R}(u_i)$. Since the ranks are chosen by the nodes independently at random, we have $\Pr\{\xi_i\} = \frac{1}{i}$, and thus, $E[|\mathbb{L}_{M_k}(v)|] = \sum_{i=1}^{|M_k(v)|} \frac{1}{i} \leq \log|M_k(v)| \leq \log n$.

Notice that the events ξ_is, $1 \leq i \leq |M_k(v)|$, are independent. By using the Chernoff bound, we can show that for a particular v, $|\mathbb{L}(M_k(v))| < 5\log n$ with probability at least $1 - \frac{1}{n^3}$. Since $k < n$, by using the union bound, we have $|\mathbb{L}_{M_k}(v)| < 5\log n$ for all v and k simultaneously with probability at least $1 - \frac{n^2}{n^3} \geq 1 - \frac{1}{n}$. This completes the proof. □

Lemma 2.7 *In every phase k, a node v exchanges $O(\log n)$ messages with each of its neighbors WHP.*

Proof: Nodes exchange three types of messages during a phase: Rank, echo, and termination-related (leader, done, termination).

By Lemma 2.6, each node sends its neighbors $O(\log(n))$ rank messages, and likewise, receives $O(\log(n))$ rank messages from each neighbor.

We now bound echo messages. During phase k, a node v echoes back to a neighbor u either rank messages received (and ignored) from u during the phase, or rank messages that were previously received from u. The latter must correspond to entries that exist at the beginning of the phase in $\mathsf{le}_v(k-1)$, and are either deleted at phase k, or remain in $\mathsf{le}_v(k)$ (and echoes for them arrived). By Lemma 2.6, the number of messages arriving in phase k from u is bounded by $O(\log n)$ WHP, and by the same lemma, so is the size of $\mathsf{le}_v(k-1)$.

Finally, in each phase a node may exchange with each neighbor at most one of each, *leader*, *done*, *termination*, messages. Hence, the lemma follows. □

Theorem 2.8 *The LE-Dist algorithm terminates in $O(S \log n)$ rounds using $O(|E|S \log n)$ messages WHP.*

Proof: Each phase takes $O(\log(n))$ communication rounds WHP, by Lemma 2.7. When all nodes complete $S + 1$ phases, no new rank messages are generated. Within additional S phases, echo messages travel back to all nodes. Finally, constructing the termination tree by flooding a *leader* notification takes $O(D)$ phases, where D is the unweighted (hop) diameter. The resulting BFS tree has height D. Hence, a converge-cast of *done* messages and a broadcast of the *termination* signal take altogether $O(D)$ phases. In total, since $D \leq S$, the algorithm terminates in $O(S \log n)$ rounds and uses $O(|E|S \log n)$ messages. □

Tightness of Analysis: The analysis of the LE-Dist Algorithm is tight up to constant factors as stated in the following theorem (the proof is omitted).

Theorem 2.9 *For arbitrary constants ϵ, δ ¿ 0 such that $\epsilon + \delta < 1/2$, it is possible to construct a weighted n-node graph G with $S = \Theta(n^{\delta})$ and $|E| = \Theta(n^{2-2\epsilon})$ such that the expected time and message complexities of the LE-Dist algorithm are $\Omega(\epsilon S \log n)$ and $\Omega(\epsilon |E| S \log n)$.*

2.5 Analysis for Unweighted Graphs

In an unweighted graph, the LE-Dist algorithm can be simplified as described below. On unweighted graphs, the distance ρ traveled by a message is simply the number of edges on the path that the message followed. Initially, at phase $k = 1$, v forwards its own rank to all of its neighbors. By induction on k, we have that i) the messages arriving at v in phase k, have larger ρ values than that of all messages arriving in phases $k' < k$, ii) all messages arriving at v in phase k, have $\rho = k$. As a result, in phase $k + 1$, we only need to forward the message with the lowest-ranked originator among the messages received in phase k and we can ignore all other messages. Hence, a message is never delayed, it is either ignored or forwarded. As a result, each phase boils down to a single round. We no longer need to use control messages to mark the end of a phase and we do not need to keep the new/old flag. For the message complexity, we therefore only need to count the number of *rank* messages. Note that the number of *echo* messages is exactly equal to the number of *rank* messages.

Theorem 2.10 *For unweighted graphs, LE-lists can be computed in $O(D)$ rounds (for any ordering of the nodes). If ranks are chosen randomly, the message complexity is $O(|E| \min(D, \log n))$ WHP.*

Proof: Each phase takes one communication round (i.e., there is no delay). Hence the algorithm terminates in $O(S) = O(D)$ rounds.

As we observed, the messages that arrive at node v in phase k have a larger ρ value than that of a message that arrives in a phase $t' < t$. Thus an item in le_v never gets deleted once the item is inserted into le_v. Further, whenever v forwards a message, it inserts the corresponding item in le_v. Thus the total number of messages forwarded by v is bounded by $|\mathbb{L}(v)|$. Node v forwards each message through at most $\deg(v)$ adjacent edges, where $\deg(v)$ is the degree of v. Now, $|\mathsf{le}_v| = O(\log n)$ WHP (Lemma 2.6) and also $|\mathsf{le}_v| \leq D$ since in each round, at most one item is inserted in le_v. Thus, the message complexity is $\sum_{v \in V} \deg(v) \cdot O(|\mathsf{le}_v|) \leq \sum_{v \in V} \deg(v) \cdot O(\min\{D, \log n\}) = O(|E| \cdot \min\{D, \log n\})$. □

Leader Election: The LE-Dist algorithm for unweighted graphs is a leader election algorithm since the lowest ranked node can be elected as the leader and the all the rest know it (cf. Section 2.2). Hence the algorithm for unweighted graphs can be used as a synchronous, leader election algorithm for arbitrary graphs. By Theorem 2.10, the time complexity is $O(D)$ (deterministically) and the message complexity is $O(|E| \min(D, \log n))$ WHP.

3 Probabilistic Tree Embedding

We next show how to compute an FRT embedding as described in [10], i.e., we give a randomized algorithm that maps the given graph metric into a tree such that every pair of nodes has good expected stretch, where the expectation is taken over the coin tosses of the algorithm.

1. Construction of LE-lists: Construct the LE-lists of the nodes using the LE-Dist algorithm.

2. Construction of β-lists: Given a parameter $\beta \in [1, 2]$, let $\beta_i = 2^{i-1}\beta$ for $i = 0, 1, 2, \ldots, \delta = \lceil \log \Delta \rceil$. The β-list of a node v, denoted by $\beta(v)$, is a list of pairs $\langle \beta_i, u_i \rangle$ such that u_i is the least element in $\Gamma_{\beta_i}(v)$. The leader selects a real number β uniformly at random in $[1, 2]$ and broadcasts β to the other nodes using the

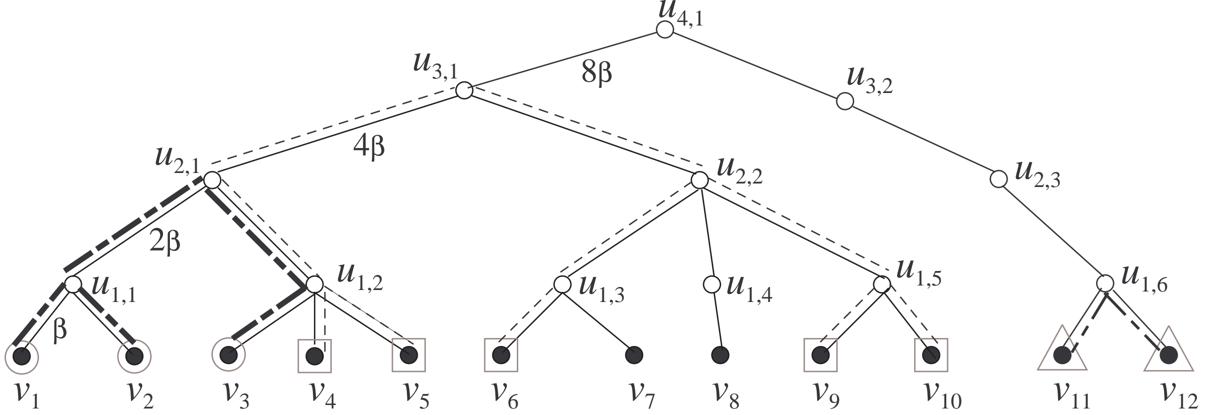

Figure 1: Construction of GSF in an FRT tree. $V = \{v_1, v_2, \ldots, v_{12}\}$, $\delta = 4, V_1 = \{v_1, v_2, v_3\}$, $V_2 = \{v_4, v_5, v_6, v_9, v_{10}\}$, $V_3 = \{v_{11}, v_{12}\}$. The solid black nodes are in V. The white nodes are internal nodes of an FRT tree. Nodes in V_1, V_2 and V_3 are marked by circles, squares, and triangles, respectively. The solid lines are the edges in an FRT tree, and the dotted lines are the edges in the Steiner forest.

BFS tree computed during the LE lists computation. This needs $O(D)$ time and $O(n)$ messages. Once a node gets β, it can compute its β-list from its LE-list.

3. Tree Embedding: The β-lists of the nodes define a hierarchical clustering as follows: The top level cluster consists of all nodes in V and we define its level to be δ. Each level-i cluster $C_{i,j}$ is decomposed into level-$(i-1)$ clusters. We define u_i to be the level i cluster center for v if $\langle \beta_i, u_i \rangle \in \beta(v)$. All nodes in $C_{i,j}$ with the same level-$(i-1)$ cluster center form one level-$(i-1)$ cluster. Thus v itself is its level-0 cluster center, i.e., $u_0 = v$, and the center of the root cluster is the least element in V. Note that a particular node may be the center of several different clusters at the same level and at different levels. Also observe that each cluster at level i has diameter at most $2^i\beta$.

Given the hierarchical clustering, one can naturally define a tree as follows: each cluster corresponds to a node in the tree, and we have an edge from cluster $C_{i,j}$ to $C_{i-1,j'}$ iff $C_{i-1,j'}$ is a sub-cluster of $C_{i,j}$. The level-0 clusters that form the leaves of this tree are singleton clusters that correspond to the nodes of V. Thus each $v \in V$ forms a leaf node in the tree (the black nodes in Figure 1). The edge from a level i cluster to its parent has weight $2^i\beta$. It is easy to verify that for any edge in the tree, the graph distance between the corresponding cluster centers is no larger than twice the weight of the edge. Clusters have radii at most $2^{i-1}\beta$. It is shown in [10] that for any $v, w \in V$, the tree distance between v and w is (deterministically) at least as large as the original graph distance, and that the expected tree distance between them is $O(\log n)$ times their original distance.

In a distributed setting, we do not explicitly construct this FRT tree. Instead, the β-lists implicitly define the tree. We will think of each cluster as being represented by its center.

4 Distributed Approximation Algorithms

4.1 Generalized Steiner Forests

The tree embedding inspires a natural centralized algorithm for the GSF problem (defined in Section 1): Observe that the problem can be solved optimally on a tree by connecting each pair of nodes belonging to

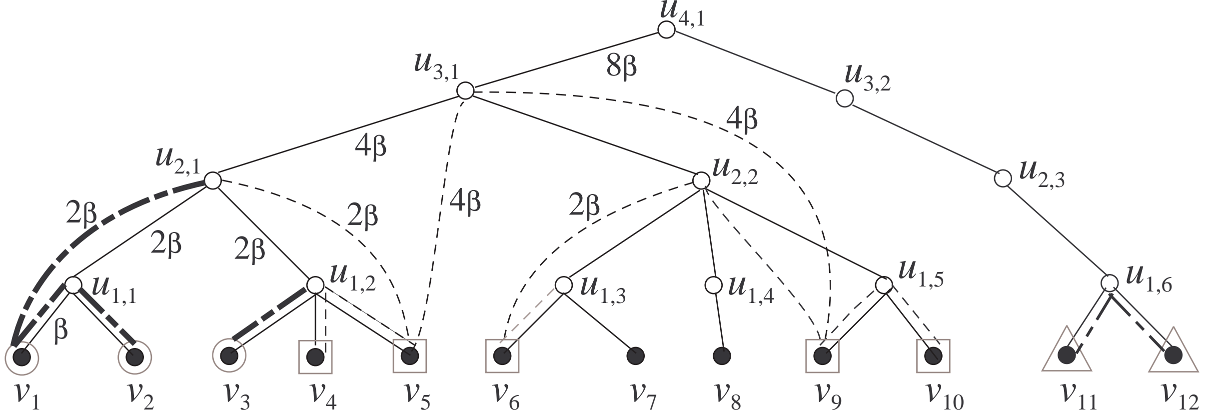

Figure 2: Distributed GST construction. The dotted lines are the paths that are included in the Steiner subgraph. The light-colored dotted lines, e.g., $P(v_6, u_{1,3})$, may or may not be included in the Steiner subgraph.

the same group V_j (for all $1 \leq j \leq k$) via the nearest common ancestors (Figure 1) and taking the union of all selected edges for all sets V_j, $1 \leq j \leq k$. This solution, applied to the tree resulting from an embedding, can be converted to a solution for the original graph by including, for each edge in the tree solution, the shortest path in the graph between the corresponding cluster centers (note that the resulting subgraph may not be forest). The bound on the expected distortion of the FRT embedding implies that in expectation, the cost of OPT on the tree is no larger than $O(\log n)$ times the optimal solution to the original instance. The fact that the distances in the tree are always larger than corresponding distances in the graph implies that the cost of the solution induced on the graph is no larger than the cost of the solution on the tree. Thus, this algorithm gives an $O(\log n)$-approximation to the problem in expectation. Next, we describe a fast distributed approximation algorithm based on this approach.

Distributed Algorithm

We assume that we have an implicit representation of the FRT tree; i.e., each node has its LE-list and β-list. Our distributed algorithm constructs an (explicit) $O(\log n)$-approximate Steiner subgraph. In other words, at the end of the algorithm, each node knows which of its incident edges belong to the Steiner subgraph. We focus on finding an $O(\log n)$-approximate Steiner subgraph, not necessarily a forest. However, a Steiner forest can be constructed from the Steiner subgraph, with the same time and message complexities, by a breadth first search.

Before we detail the algorithm, we illustrate the main ideas with an example. In a distributed setting, the internal nodes (the white nodes in Figure 1) of an FRT tree are represented by the centers of the corresponding clusters. An edge of an FRT tree is replaced by the shortest path between the corresponding cluster centers. However, replacing an edge between two white nodes in an FRT tree is not straightforward: using LE-lists as routing tables (Lemma 2.1), we can find the shortest path from a black node to any of its ancestor white nodes but not between two white nodes. To resolve this problem, we pick an arbitrary black descendant on a white node and connect it to the higher level white node. Thus in Figure 1, edge $(u_{1,2}, u_{2,1})$ of an FRT tree can be replaced with the shortest path $P(v_5, u_{2,1})$ in G.

For a group V_j, let $\ell(V_j)$ denote the level of the lowest common ancestor of the nodes in V_j, in the tree; i.e. the smallest i such that all nodes in V_j belong to a single level i cluster. For a node v in group V_j, we define $\ell(v)$ to be $\ell(V_j)$ (recall that the groups are disjoint).

The algorithm consists of two parts: the first is a *discovery* part, where each node v computes $\ell(v)$, and the second is a *construction* part where the Steiner subgraph is constructed using this information.

Discovery Part

We start by describing a simplified version of the discovery part of the algorithm, which we will then refine to avoid congestion. We describe the algorithm for a single group V_j; the k groups run the algorithm in parallel. The algorithm consists of $\delta = \lceil \log \Delta \rceil$ synchronized phases. The nodes begin $(i+1)^{st}$ phase after all nodes finish their i^{th} phase. (We describe later how to synchronize the phases.) Each node v computes $\ell(v)$ by sending messages to its level i ancestor u_i, for each i, and determining whether the cluster defined by its level i ancestor contains all of V_j, starting from the top down.

We next describe the ith phase. Let $\bar{i} = \delta - i + 1$. Each node v that hasn't yet determined $\ell(v)$ sends a *find-lca* message to its level $\bar{i}$ cluster center $u_{\bar{i}}$, containing its own identifier, its group identifier, and its level $\bar{i} - 1$ cluster center. The level $\bar{i}$ cluster center looks at all the messages it receives, and if it receives messages containing at least two of its children as level $(\bar{i} - 1)$ cluster centers, it declares itself to be the least common ancestor and sends a YES reply to each node it received a message from. On the other hand, if all messages agree on the level $(\bar{i} - 1)$ cluster center, the level $\bar{i}$ cluster center sends a NO message, communicating that the least common ancestor is lower down the tree. A node $v \in V_j$ stops as soon as it receives a YES message and sets $\ell(v)$ to $\bar{i}$.

Lemma 4.1 *At the end of δ phases, each node v in each group will determine the correct value of $\ell(v)$.*

Proof: The lemma would be immediate if instead of cluster center $u_{\bar{i}-1}$, we sent a unique identifier for the cluster to the parent cluster center. However, one node u can be the cluster center for several clusters at the same level. We next argue that from the point of view of this algorithm, the cluster centers are in fact unique identifiers.

Let C_{i^*} be the cluster corresponding to the true lca with center u_{i^*}. Thus $V_j \subseteq C_{i^*}$, so that messages sent to centers u_i for $i \geq i^*$ all agree on their level $i - 1$ cluster and hence on the cluster center. The tree construction is easily seen to have the property that all *children* of a cluster C have distinct cluster centers. Thus the messages to the level i^* cluster center, which contain centers of at least two distinct clusters, must in fact contain at least two distinct cluster centers. Thus the response from u_{i^*} will in fact be YES and the nodes in V_j will stop with the right value of $\ell(v)$. □

We next mention some implementation details and refinements.

Routing: Lemma 2.1 implies that for every $v \in V$, routing from v to its level i cluster center u_i, for any i, can be done using the LE lists as routing tables. (The route taken will be the weighted shortest path to u_i.) Additionally, whenever a node w gets a message originating from v along edge e, destined for a node u_i, w stores a routing table entry consisting of v and the edge e. These routing table entries enable routing from u_i to v. Thus all messages can be routed without redundant transmissions.

Controlling Congestion: In the above protocol, several nodes can send a message to their level i cluster center simultaneously. Since the paths from these nodes to their parents can intersect, this can lead to congestion on edges. We modify the protocol so as to merge messages belonging to the same group and destined for the same cluster center, according to the following rules:

1) If in any round, a node w gets at least two simple find-lca messages all agreeing in $u_{\bar{i}}$ and $u_{\bar{i}-1}$ and j: w picks one of the messages $\langle v, u_{\bar{i}}, u_{\bar{i}-1}, j \rangle$ arbitrarily and forwards it.

2) If in any round, a node w gets at least two simple find-lca messages $\langle v, u_{\bar{i}}, u_{\bar{i}-1}, j\rangle$ and $\langle v', u'_{\bar{i}}, u'_{\bar{i}-1}, j\rangle$ such that $u_{\bar{i}} = u'_{\bar{i}}$ but $u_{\bar{i}-1} \neq u_{\bar{i}-1}$: w concatenates two of the messages and forwards a concatenated message $\langle v, u_{\bar{i}}, u_{\bar{i}-1}, j, v'u'_{\bar{i}-1}\rangle$.

3) If in any round, a node w receives more than one message agreeing on $u_{\bar{i}}$ and j, including at least one concatenated message: w forwards a concatenated message.

4) Messages agreeing on the group j but meant for different cluster centers are not combined.

5) Whenever w drops or concatenates messages, it makes a record locally so that when the reply arrives, it can forward it to all the relevant nodes.

Note that now a cluster center u_i learns of only a subset of the nodes in V_j. However, the above rules ensure that if there are at least two nodes u and u' with different level $(\bar{i} - 1)$ cluster centers, then u_i gets messages for at least two such nodes. Thus the correctness of the algorithm (i.e., correctness of ℓ values) continues to hold.

We next prove a lemma to help bound the congestion.

Lemma 4.2 *Suppose nodes in a group V_j send a message in phase i. Then they all have the same level $\bar{i}$ ancestor $u_{\bar{i}}$.*

Proof: We show this by induction on i. In the first phase, clearly all messages are meant for the root so the base case holds. Suppose the claim holds for all $i - 1$. Since a message is sent in phase i, all the previous responses must be NO . In particular, $u_{\bar{i}+1}$ sent a NO message which implies that all nodes agree on $u_{\bar{i}}$. The claim follows. □

Thus rule 4) above never applies and at most one message is sent per round on each edge.

Synchronizing the Phases: In each phase, a cluster center waits Sk rounds so that it receives all of the *find-lca* messages destined to it before responding to any of them. Note that S and k can easily be computed during the execution of LE-dist algorithm.

Before starting the next phase, each node waits for another Sk rounds to make sure all nodes receives the responses of their *find-lca* messages. The correctness of this synchronization follows from the following lemma.

Lemma 4.3 *Any message in the first part of the GST algorithm takes at most Sk time to reach its destination.*

Proof: A message travels through at most S edges and congestion at any edge is at most k. The claim follows. □

Construction Part

The construction part of the algorithm also runs in synchronized phases. Each node $v \in \cup V_j$, tries to construct paths that correspond to the tree path from v to its level $\ell(v)$ ancestor. The definition of ℓ implies that each group V_j is connected in the resulting subgraph.

In phase i, each node v with $\ell(v) \geq i$ ensures that its level $(i - 1)$ cluster center is connected to its level i cluster center. This is achieved by picking a node w from each relevant level $(i - 1)$ cluster, and connecting w both to its level $i-1$ cluster center and to its level i cluster center. Thus u_{i-1} and u_i get connected via some node w in u_i's cluster, and the tree construction ensures that the expected cost of this subgraph is within $O(\log n)$ of OPT.

However constructing the above subgraph may require too many messages, as each node u_i may represent several cluster centers, and hence may need to connect to several level $(i+1)$ cluster centers in one phase. To alleviate this, we will actually construct only a subset of these edges. More precisely, if there are several different nodes $v_1, \ldots, v_s$ belonging to the same group V_j that want to connect their level i cluster center u_i (possibly representing different clusters), to their level $(i+1)$ cluster centers $u_{i+1}^{(1)}, \ldots, u_{i+1}^{(s)}$ respectively, we select one arbitrarily and connect u_i to u_{i+1}. For concreteness, suppose that we always select the v_x with the lowest rank; we say that $v_y, x \neq y$ is *overruled* by v_x. Let E' be the corresponding subset of pairs (u_i, u_{i+1}). For a pair $(u_i, u_{i+1}) \in E'$, define $w(u_i, u_{i+1})$ to be $2^{i+2}\beta$.

Before we show how this is implemented in the distributed setting, we show that this optimization preserves correctness. It is easy to see that this builds only a subset of edges and hence the approximation guarantee is preserved. We next argue that this still ensures that each V_j is connected in the resulting subgraph.

Lemma 4.4 *Let the subgraph E' be constructed as above. Then each V_j is connected in E'.*

Proof: Consider a particular group V_j, with $\ell(v) = l$ for each v in V_j. Thus each $v \in V_j$ has the same level l ancestor u_l. We show that E' has a path from u_l to every $v \in V_j$, by induction on the rank of v.

Clearly the lowest ranked node in V_j never gets over-ruled, and hence constructs a path from its level i ancestor to its level $(i+1)$ ancestor, for every $i : 0 \leq i \leq (l-1)$. This establishes the base case.

Consider a node $v \in V_j$ and assume inductively that all lower ranked nodes have a path to u_l. If v is never overruled, it clearly has a path to u_l. Else it first gets overruled at some step i, say by a node v'. In this case, v has a path to u_i, and v' has a path to u_i as well. Moreover, since v' has a lower rank, it inductively has a path to u_l. Hence v is connected to u_l as well, and the claim follows. □

We next describe how this set E' is constructed by the distributed algorithm. Nodes that have not yet been overruled are considered *selected*, and these nodes send messages to their level i ancestors in phase i. These messages help u_i determine the next level cluster centers that it needs to connect to. For each such u_{i+1}, it picks an arbitrary descendant from the corresponding cluster and asks it to connect to u_{i+1}. Initially, each node v considers itself *selected* for phase 1 if $\ell(v) \neq 0$. Only the *selected* nodes participate in the next phase.

In phase $i = 0, 1, 2, \ldots, \delta$:

- Each *selected* node v belonging to a group V_j sends a *want connected* message $\langle v, u_i, u_{i+1}, j \rangle$ to its level-i cluster center u_i.
- After the cluster center u_i receives *want connected* messages, if any, from its *selected* descendants in group V_j, u_i picks the one with the lowest rank and selects it. Then u_i sends a *selected-for-next-phase* message to it. Additionally, let U_{i+1} be the set of level $(i+1)$ cluster centers that are received from the selected nodes. For each $u_{i+1} \in U_{i+1}$, u_i picks an arbitrary sender v of that request, and sends v a *chosen to connect* message consisting of u_i and u_{i+1}.
- If v gets a *chosen to connect* message from u_i consisting of u_i and u_{i+1}, it sends a *connect* message to u_{i+1}. When a *connect* message passes through an edge (v_1, v_2), both v_1 and v_2 mark this edge to be included in the Steiner subgraph.
- After v' gets the *selected-for-next-phase* message, v' considers itself *selected* for the next phase if $i < \ell(v')$.

Note that the *selected-for-next-phase*, *chosen to connect*, and *connect* messages correspond uniquely to some *want connected* message, and hence it suffices to analyze the time and message complexity of the *want connected* messages. We next show the following invariants.

Lemma 4.5 *If v sends a* want connected *message to u_i, then v and u_i are connected in the Steiner subgraph.*

Proof: The proof is by induction on i. The base case is trivial as u is connected to itself. Suppose that v is selected in phase i so that it sends a *want connected* message to u_{i+1}. Then $u_{i+1} \in U_{i+1}$ so that there is v' which receives a *chosen to connect* message from u_i consisting of u_{i+1}. Thus at the end of this phase, v' and u_{i+1} are connected in the Steiner subgraph. Inductively, both v and v' were connected to u_i in the Steiner subgraph. The claim follows. □

Lemma 4.6 *If u_i is connected to u_{i+1} in E' above, then the Steiner subgraph built by the distributed algorithm has a u_i—u_{i+1} path. Moreover, the Steiner subgraph has cost no more than $w(E')$.*

Proof: The set of edges incident on u_i and a higher level cluster center in E' consists exactly of the level $i+1$ ancestors of nodes that are minimal ranked in $V_j \cap desc(u_i)$ for some j. This is exactly the set U_{i+1} and the *chosen to connect* messages ensure that the Steiner subgraph has a path from u_i to u_{i+1}, for every $u_{i+1} \in U_{i+1}$.

The fact that a unique v' is chosen to connect ensures that bound on the weight of the subgraph. □

These imply that running the above algorithm will give subgraph that connects each of the V_j's and has expected cost at most $O(\log(n) \cdot OPT)$.

Routing: Since messages go from nodes to their cluster centers, and back, routing is easily done as in the first part.

Controlling Congestion: Similar to the discovery part of the algorithm congestion is controlled by combining the *want connected* messages if they meet at some intermediate node u on their way and have the same destination:

- If two or more *want connected* messages belonging to the same group (e.g. messages originated by nodes $x_1, x_2, \ldots, x_r$) with the same destination u_i arrive at u at the same time, u picks the lowest ranked one and forwards it towards u_i.
- If u_i sends a *selected-for-next-phase* or a *chosen to connect* message destined for x_j, u simply forwards it to x_j.

It is easy to verify that while this pruning allows fewer *want connected* messages from reaching u_i, it still ensures that for each node that would get selected in the unpruned run, its *want connected* message is not pruned, and hence the same set of nodes gets selected. Thus the set of edges in the Steiner subgraph is unchanged.

Synchronizing the Phases: In each phase, a cluster center waits $kSL_{\max}$ rounds so that it receives all of the *connect* messages destined for it, where $L_{\max} = \max_{v \in V} |\mathbb{L}(v)|$. The leader can find k and $L_{\max}$ by using BFS tree as an aggregation tree and broadcasting $L_{\max}$ to other nodes. This takes $O(D)$ time and $O(n)$ messages. Before starting the next phase, a *selected* node waits for another $kSL_{\max}$ rounds. The correctness of this synchronization follows from the following lemma.

Lemma 4.7 *In the second part of the GSF algorithm, congestion at any edge adjacent to v is at most $k|\mathbb{L}(v)|$.*

Proof: A node v' sends a message to one of its cluster centers u_i through the shortest path $P(v', u_i)$. If v is on path $P(v', u_i)$, there must be an entry $\langle u_i, *\rangle$ in $\mathbb{L}(v)$ (cf. Lemma 2.1 and 2.4). Thus, v can be on the shortest path tree for at most $|\mathbb{L}(v)|$ different destinations. The rules above for controlling congestion ensure that at most one message is forwarded per group to each cluster center, and hence the congestion at any edge (v, u') is at most $k|\mathbb{L}(v)|$. □

Theorem 4.8 *The above distributed GSF algorithm gives an expected $O(\log n)$ approximation and takes $O(S\,k \log^2 n)$ time and at most $O(S\,n \log n)$ messages.*

Proof: The expected $O(logn)$ approximation follows from
Lemma 4.6, and the fact that the congestion-related pruning does not affect the set of edges added to the Steiner subgraph.

The time complexity follows from the fact that there are δ phases and each phase takes $2S\,k$ time in the first part and $O(S\,kL_{\max})$ time in the second part of the algorithm. For the message complexity, notice that in each phase, each node generates at most one *find-lca* or one *want connected* message and receives its response. Since each message travels through at most S edges, the number of messages is at most $4S\,n\delta$. □

Unweighted Graphs: In an unweighted graph, the algorithm works in the same way except for the following modification of the synchronization of the phases. In the first part, in phase i, a cluster center waits $\lceil\beta_i\rceil k$ rounds to receive all *find-lca* messages, and a node v waits for $2\lceil\beta_i\rceil k$ rounds to begin the next phase. In the second part, in phase i, a cluster center waits $k\lceil\beta_i\rceil L_{\max}$ rounds to receive the *want connected* messages, and v waits for $2k\lceil\beta_i\rceil L_{\max}$ rounds to begin the next phase, which leads to the following time and message complexity.

Theorem 4.9 *In an unweighted graph, the distributed GSF algorithm time and message complexities $O\left(kD \log n + D \log^2 n\right)$ and $O(nD \log n)$, respectively.*

4.2 Routing Cost Spanning Trees

The problem (defined in Section 1.3) is trivially solvable in trees. Thus, using the FRT embedding, we can obtain an (expected) $O(\log n)$ approximation for the problem. Using a similar approach to GSF (with $k = 1$) we obtain the following result.

Theorem 4.10 *There is an (expected) $O(\log n)$-distributed approximation algorithm for the routing cost spanning tree problem that takes $O(S \log^2 n)$ time and $O(S\,n \log n)$ messages.*

4.3 *k*-Source Shortest Paths

We present a distributed approximation algorithm for k-source shortest paths (defined formally in Section 1.3) in an arbitrary weighted network. FRT embedding gives (expected) $O(\log n)$-approximate paths between any pair of nodes. Below is a brief description of the algorithm.

1. All $v \in V$ find their LE-lists using LE-Dist algorithm and compute the β-lists.

2. Each source $s_j \in K$ constructs a BFS tree BFS_j rooted at s_j by initiating a breadth-first search and broadcasts its LE-list $\mathbb{L}(s_j)$ to all $v \in V$ using using the edges of BFS_j. This process takes $O(kD \log n)$ rounds and $O(k|E| + kn \log n)$ messages for all k sources.

3. Each $v \in V$ computes $\beta(s_j)$, $1 \le j \le k$, from $\mathbb{L}(s_j)$. Now v can determine the lowest common ancestor of $\{v, s_j\}$ from the β-lists $\beta(v)$ and $\beta(s_j)$. A path from v to s_j is constructed by concatenating the shortest paths $P(v, u_i)$ and $P(u_i, s_j)$. The shortest path from any node to any of its cluster center can be found using the the LE-lists as routing tables (Lemma 2.1). Thus, $P(v, u_i)$ can be constructed correctly, but we can construct $P(u_i, s_j)$ only in reverse direction. To solve this problem, each source s_j sends a *dummy* message to all of its cluster centers u_i so that the nodes in this path can track their predecessors for the purpose of constructing the routing table for the reverse paths toward the sources. This step takes $O(kS \log n)$ time and $O(kS \log n)$ messages.

Theorem 4.11 *The above algorithm computes an (expected) $O(\log n)$-approximate k-source shortest paths in $O(kD \log n)$ time using $O(|E|(\min[D, \log n] + k) + kn \log n)$ messages in an unweighted graph and in $O(kS \log n)$ time using $O(|E|(S \log n + k) + kn \log n)$ messages in a weighted graph.*

5 Concluding Remarks

It would be interesting to improve the time and message complexities of our distributed approximation algorithms, especially for weighted graphs. The key in doing this is to improve the complexities for computing LE-lists. It might also be possible to extend our approach to design $O(\log n)$ (or polylogarithmic) distributed approximation algorithms for other network optimization problems that are easily solvable on trees.

References

[1] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *19th STOC*, 1987.

[2] P. Chalermsook and J. Fakcharoenphol. Simple distributed algorithms for approximating minimum steiner trees. In *11th COCOON*, 2005.

[3] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *JCSS*, 55(3):441–453, 1997.

[4] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network. *JCSS*, 73(3):265–288, 2007.

[5] D. Dubhashi, F. Grandioni, and A. Panconesi. Distributed algorithms via lp duality and randomization. In *Handbook of Approximation Algorithms and Metaheuristics*. 2007.

[6] M. Elkin. Computing almost shortest paths. In *20th PODC*, 2001.

[7] M. Elkin. Distributed approximation - a survey. *ACM SIGACT News*, 35(4):40–57, 2004.

[8] M. Elkin. A faster distributed protocol for constructing minimum spanning tree. In *15th SODA*, 2004.

[9] M. Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *36th STOC*, 2004.

[10] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *JCSS*, 69(3):485–497, 2004.

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

[12] J. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. on Computing*, 27:302–316, 1998.

[13] F. Grandoni, J. Könemann, A. Panconesi, and M. Sozio. Primal-dual based distributed algorithms for vertex cover with semi-hard capacities. In *24th PODC*, 2005.

[14] L. Jia, R. Rajaraman, and R. Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193–205, 2002.

[15] M. Khan and G. Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. *Distributed Computing*, 20:391–402, 2008.

[16] F. Kuhn and T. Moscibroda. Distributed Approximation of Capacitated Dominating Sets. In *19th SPAA*, 2007.

[17] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What Cannot Be Computed Locally! In *23rd PODC*, 2004.

[18] F. Kuhn, T. Moscibroda, and R. Wattenhofer. The Price of Being Near-Sighted. In *17th SODA*, 2006.

[19] S. Kutten and D. Peleg. Fast distributed construction of k-dominating sets and applications. *J. Algorithms*, 28:40–66, 1998.

[20] D. Peleg. A time optimal leader election algorithm in general networks. *J. Parallel and Distributed Computing*, 8:96–99, 1990.

[21] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

[22] D. Peleg and V. Rabinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *40th FOCS*, 1999.

[23] V. Vazirani. *Approximation Algorithms*. Springer Verlag, 2004.

[24] B. Wu, G. Lancia, V. Bafna, K. Chao, R. Ravi, and C. Tang. A polynomial time approximation scheme for minimum routing cost spanning trees. In *9th SODA*, 1998.