Improved Distributed Steiner Forest Construction

Extended Abstract

Christoph Lenzen*
Massachusetts Institute of Technology
32 Vassar Street, 02139 Cambridge, USA
clenzen@csail.mit.edu

Boaz Patt-Shamir†
Tel Aviv University
Tel Aviv 69978, Israel
boaz@eng.tau.ac.il

ABSTRACT

We present new distributed algorithms for constructing a Steiner Forest in the CONGEST model. Our deterministic algorithm finds, for any given constant $\varepsilon > 0$, a $(2+\varepsilon)$-approximation in $\tilde{\mathcal{O}}(sk + \sqrt{\min\{st, n\}})$ rounds, where s is the shortest path diameter, t is the number of terminals, k is the number of terminal components in the input, and n is the number of nodes. Our randomized algorithm finds, with high probability, an $\mathcal{O}(\log n)$-approximation in time $\tilde{\mathcal{O}}(k + \min\{s, \sqrt{n}\} + D)$, where D is the unweighted diameter of the network. We also prove a matching lower bound of $\tilde{\Omega}(k + \min\{s, \sqrt{n}\} + D)$ on the running time of any distributed approximation algorithm for the Steiner Forest problem. Previous algorithms were randomized, and obtained either an $\mathcal{O}(\log n)$-approximation in $\tilde{\mathcal{O}}(sk)$ time, or an $\mathcal{O}(1/\varepsilon)$-approximation in $\tilde{\mathcal{O}}((\sqrt{n}+t)^{1+\varepsilon} + D)$ time.

1. INTRODUCTION

Ever since the celebrated paper of Gallager, Humblet, and Spira [12], the task of constructing a minimum-weight spanning tree (MST) continues to be a rich source of difficulties and ideas that drive network algorithmics (see, e.g., [10, 13, 21, 23]). The *Steiner Forest* (SF) problem is a strict generalization of MST: We are given a network with edge weights and some disjoint node subsets called *input components*; the task is to find a minimum-weight edge set which makes each component connected. MST is a special case of SF, and so are the Steiner Tree and shortest s-t path problems. The general SF problem is well motivated by many practical situations involving the design of networks, be it physical (it was famously posed as a problem of railroad design), or virtual (e.g., VPNs or streaming multicast). The problem has attracted much attention in the classic algorithms community, as detailed on the dedicated website [14].

The first network algorithm for SF in the CONGEST model (where a link can deliver $\mathcal{O}(\log n)$ bits in a time unit—precise definitions in Section 2) was presented by Khan *et al.* [16]. It provides $\mathcal{O}(\log n)$-approximate solutions in time $\tilde{\mathcal{O}}(sk)$, where n is the number of nodes, k is the number of components, and s the *shortest path diameter* of the network, which is (see Section 2) the maximal number of edges in a weighted shortest path. Subsequently, in [19], it was shown that for any given $0 < \varepsilon \leq 1/2$, an $\mathcal{O}(\varepsilon^{-1})$-approximate solution to SF can be found in time $\tilde{\mathcal{O}}((\sqrt{n}+t)^{1+\varepsilon} + D)$, where D is the diameter of the unweighted version of the network, and t is the number of *terminals*, i.e., the total number of nodes in all input components. The algorithms in [16, 19] are both randomized.

Our Results. In this paper we improve the results for SF in the CONGEST model in two ways. First, we show that for any given constant $\varepsilon > 0$, a $(2+\varepsilon)$-approximate solution to SF can be computed by a deterministic network algorithm in time $\tilde{\mathcal{O}}(sk + \sqrt{\min\{st, n\}})$. Second, we show that an $\mathcal{O}(\log n)$-approximation can be attained by a randomized algorithm in time $\tilde{\mathcal{O}}(k + \min\{s, \sqrt{n}\} + D) \subseteq \tilde{\mathcal{O}}(s + k)$. On the other hand, we show that any algorithm in the CONGEST model that computes a solution to SF with non-trivial approximation ratio has running time in $\tilde{\Omega}(k + \min\{s, \sqrt{n}\} + D)$. If the input is not given by indicating to each terminal its input component, but rather by *connection requests* between terminal pairs, i.e., informing each terminal which terminals it must be connected to, an $\tilde{\Omega}(t + \min\{s, \sqrt{n}\} + D)$ lower bound holds. (It is easy to distributively transform connection requests into equivalent input components in $\mathcal{O}(t + D)$ rounds.)

Related work. The Steiner Tree problem (the special case of SF where there is one input component) has a remarkable history, starting with Fermat, who posed the geometric 3-point on a plane problem circa 1643, including Gauss (1836), and culminating with a popularization in 1941 by Courant and Robbins in their book "What is Mathematics" [8]. An interesting account of these early developments is given in [3]. The contribution of Computer Science to the history of the problem apparently started with the inclusion of Steiner Tree as one of the original 21 problems proved NP-complete by Karp [15]. There are quite a few variants of the SF problem which are algorithmically interesting, such as Directed Steiner Tree, Prize-Collecting Steiner Tree,

*Supported in part by the National Science Foundation under Grant Nos. CCF-AF-0937274, 0939370-CCF, and CCF-1217506, the AFOSR under Contract No. AFOSR Award number FA9550-13-1-0042, the German Research Foundation (DFG, reference number Le 3107/1-1).

†Supported in part by a grant from the Israel Ministry of Science, Technology and Space and the French Ministry of Higher Education and Research.

PODC'14, July 15–18, 2014, Paris, France.

ACM 978-1-4503-2944-6/14/07 ...$15.00.
http://dx.doi.org/10.1145/2611462.2611464.

Group Steiner Tree, and more. The site [14] gives a continuously updated state of the art results for many variants. Let us mention results for the most common variants: For the Steiner Tree problem, the best (polynomial-time) approximation ratio known is $\ln 4 + \varepsilon \approx 1.386 + \varepsilon$ for any constant $\varepsilon > 0$ [4]. For Steiner Forest, the best approximation ratio known is $2 - 1/(t-k)$ [1]. It is also known that the approximation ratio of the Steiner Tree (or Forest) problem is at least 96/95, unless P=NP [6].

Regarding distributed algorithms, there are a few relevant results. First, the special case of minimum-weight spanning tree (MST) is known to have time complexity of $\tilde{\Theta}(D+\sqrt{n})$ in the CONGEST model [9, 10, 13, 18, 23]. In [5], a 2-approximation for the special case of Steiner Tree is presented, with time complexity $\tilde{\mathcal{O}}(n)$. The first distributed solution to the Steiner Forest problem was presented by Khan *et al.* [16], where a randomized algorithm is used to embed the instance in a virtual tree with $\mathcal{O}(\log n)$ distortion, then finding the optimal solution on the tree (which is just the minimal subforest connecting each input component), and finally mapping the selected tree edges back to corresponding paths in the original graph. The result is an $\mathcal{O}(\log n)$-approximation in time $\tilde{\mathcal{O}}(sk)$. Intuitively, s is the time required by the Bellman-Ford algorithm to compute distributed single-source shortest paths, and the virtual tree of [16] is computed in $\tilde{\mathcal{O}}(s)$ rounds. A second distributed algorithm for Steiner Forest is presented in [19]. Here, a sparse spanner for the metric induced on the set of terminals and a random sample of $\tilde{\Theta}(\sqrt{n})$ nodes is computed, on which the instance then is solved centrally. To get an $\mathcal{O}(\varepsilon^{-1})$-approximation, the algorithm runs for $\tilde{\mathcal{O}}(D + (\sqrt{n}+t)^{1+\varepsilon})$ rounds. For approximation ratio $\mathcal{O}(\log n)$, the running time is $\tilde{\mathcal{O}}(D + \sqrt{n} + t)$.

Main Techniques. Our lower bounds are derived by the standard technique of reduction from results on 2-party communication complexity. Our deterministic algorithm is an adaptation of the "moat growing" algorithm of Agrawal, Klein, and Ravi [1] to the CONGEST model. It involves determining the times in which "significant events" occur (e.g., all terminals in an input component becoming connected by the currently selected edges) and extensive usage of pipelining. The algorithm generalizes the MST algorithm from [18]: for the special case of a Steiner Tree (i.e., $k = 1$), one can interpret the output as the edge set induced by an MST of the complete graph on the terminals with edge weights given by the terminal-terminal distances, yielding a factor-2 approximation; specializing further to the MST problem, the result is an exact MST and the running time becomes $\tilde{\mathcal{O}}(\sqrt{n}+D)$.

Our randomized algorithm is based on the embedding of the graph into a tree metric from [16], but we improve the complexity of finding a Steiner Forest. A key insight is that while the least-weight paths in the original graph corresponding to virtual tree edges might intersect, no node participates in more than $\mathcal{O}(\log n)$ distinct paths. Since the union of all least-weight paths ending at a specific node induces a tree, letting each node serve routing requests corresponding to different destinations in a round-robin fashion achieves a pipelining effect reducing the complexity to $\tilde{\mathcal{O}}(s+k)$. If $s > \sqrt{n}$, the virtual tree and the corresponding solution are constructed only partially, in time $\tilde{\mathcal{O}}(\sqrt{n}+k+D)$, and the partial result is used to create another instance with $\mathcal{O}(\sqrt{n})$ terminals that captures the remaining connectivity demands; we solve it using the algorithm from [19], obtaining an $\mathcal{O}(\log n)$-approximation.

Paper style. Due to the insufficient space allocated in the proceedings, we defer most details to the full paper, available online [20]. Here we give a high level description of algorithms and arguments used, together with precise definitions and statements to make the paper mathematically meaningful. In Section 2 we define the model, problem and basic concepts. Section 3 contains our lower bounds. In Section 4 and Section 5 we present our deterministic and randomized algorithms, respectively.

2. MODEL AND NOTATION

System Model. We consider the CONGEST($\log n$) or simply the CONGEST model as specified in [22], briefly described as follows. The distributed system is represented by a weighted graph $G = (V, E, W)$ of $n := |V|$ nodes. The weights $W : E \to \mathbb{N}$ are polynomially bounded in n (and therefore polynomial sums of weights can be encoded with $\mathcal{O}(\log n)$ bits). Each node initially knows its unique identifier of $\mathcal{O}(\log n)$ bits, the identifiers of its neighbors, the weight of its incident edges, and the local problem-specific input specified below. Algorithms proceed in synchronous rounds, where in each round, (i) nodes perform arbitrary, finite local computations,[1] (ii) may send, to each neighbor, a possibly distinct message of $\mathcal{O}(\log n)$ bits, and (iii) receive the messages sent by their neighbors. For randomized algorithms, each node has access to an unlimited supply of unbiased, independent random bits. Time complexity is measured by the number of rounds until all nodes (explicitly) terminate.

Notation. We use the following conventions and graph-theoretic notions.

- The number of *hops* of a path $p = (v_0, \ldots, v_{\ell(p)})$ in G is $\ell(p)$.
- The weight of path p is $W(p) := \sum_{i=1}^{\ell(p)} W(v_i, v_{i-1})$. For notational convenience, we assume w.l.o.g. that different paths have different weight (ties broken lexicographically).
- By $\mathcal{P}(v, w)$ we denote the set of all paths between $v, w \in V$ in G, i.e., $v_0 = v$ and $v_{\ell(p)} = w$.
- The (unweighted) *diameter* of G is defined as $D := \max_{v,w \in V}\{\min_{p \in \mathcal{P}(v,w)}\{\ell(p)\}\}$.
- The (weighted) *distance* of v and w in G is $\mathrm{wd}(v, w) := \min_{p \in \mathcal{P}(v,w)}\{W(p)\}$.
- $\mathrm{WD} := \max_{v,w \in V}\{\mathrm{wd}(v, w)\}$ is the *weighted diameter* of G.
- The *shortest-path-diameter* of G is defined as $s := \max_{v,w \in V}\{\min\{\ell(p) \,|\, p \in \mathcal{P}(v, w) \wedge W(p) = \mathrm{wd}(v, w)\}\}$.
- For $v \in V$ and $r \in \mathbb{R}_0^+$, we use $B_G(v, r)$ to denote the ball of radius r around v in G, which includes all nodes and edges at weighted distance at most r from v. The ball may contain edge fractions: for an edge $\{w, u\}$ for which only w is in $B_G(v, r)$, the $(r - \mathrm{wd}(v, w))/\mathrm{wd}(v, w)$ fraction of the edge closer to w is considered to be within $B_G(v, r)$, and the remainder is considered outside $B_G(v, r)$.

We use "soft" asymptotic notation. Formally, given functions f and g, define (i) $f \in \tilde{\mathcal{O}}(g)$ iff there is some $h \in \mathrm{polylog}\, n$ so that $f \in \mathcal{O}(gh)$; (ii) $f \in \tilde{\Omega}(g)$ iff $g \in \tilde{\mathcal{O}}(f)$; and (iii) $f \in \tilde{\Theta}(g)$ iff $f \in \tilde{\mathcal{O}}(g) \cap \tilde{\Omega}(g)$.

The abbreviation *w.h.p.* means *with probability* $1 - n^{-\Omega(1)}$, for a sufficiently large constant in the $\Omega(1)$ term.

[1] All our algorithms require polynomial computations only.

The Distributed Steiner Forest Problem (SF). In the Steiner Forest problem, the output is a set of edges. We require that the output edge set F is represented distributively, i.e., each node can locally answer which of its adjacent edges are in the output. The input may be represented by two alternative methods, which are both justified and common in the literature. We give the two definitions.

DEFINITION 2.1 (SF CONNECTION REQ. (DSF-CR)).
Input: *At each* $v \in V$*, a set of* connection requests $R_v \subseteq V$.
Output: *An edge set* $F \subseteq E$ *such that for each connection request* $w \in R_v$*,* v *and* w *are connected by* F.
Goal: *Minimize* $W(F) = \sum_{e \in F} W(e)$.

The set of *terminals* is $T = \{w \,|\, w \in R_v \text{ for some } v \in V\} \cup \{v \mid R_v \neq \emptyset\}$, i.e., the set of nodes v for which there is some connection request $\{v, w\}$.

DEFINITION 2.2 (SF INPUT COMPONENTS (DSF-IC)).
Input: *At each node* v*,* $\lambda(v) \in \Lambda \cup \{\bot\}$*, where* Λ *is the set of* component identifiers*. The set of* terminals *is* $T := \{v \in V \mid \lambda(v) \neq \bot\}$*. An* input component C_λ *for* $\lambda \neq \bot$ *is the set of terminals with label* λ.
Output: *An edge set* $F \subseteq E$ *such that all terminals in each input component are connected by* F.
Goal: *Minimize* $W(F) = \sum_{e \in F} W(e)$.

An instance of DSF-IC is *minimal*, if $|C_\lambda| \neq 1$ for all $\lambda \in \Lambda$. We assume that the labels $\lambda \in \Lambda$ are encoded using $\mathcal{O}(\log n)$ bits. We define $t := |T|$ and $k := |\Lambda| \leq t$, i.e., the number of terminals and input components, respectively.

We say that any two instances of the above problems on the same weighted graph, regardless of the way the input is given, are *equivalent* if the set of feasible outputs for the two instances is identical.

LEMMA 2.3. *Any instance of* DSF-CR *can be transformed into an equivalent instance of* DSF-IC *in* $\mathcal{O}(D + t)$ *rounds.*

LEMMA 2.4. *Any instance of* DSF-IC *can be transformed into an equivalent minimal instance of* DSF-IC *in* $\mathcal{O}(D + k)$ *rounds.*

3. LOWER BOUNDS

As our first result, we show that applying Lemma 2.3 to instances of DSF-CR comes at no penalty in asymptotic running time (a lower bound of $\Omega(D)$ is trivial).

LEMMA 3.1. *Any distributed algorithm for* DSF-CR *with finite approximation ratio has time complexity* $\Omega(t/\log n)$*. This is true even in graphs with diameter at most* 4 *and no more than* 2 *input components.*

The main result of this section is the following theorem.

THEOREM 3.2. *Any algorithm for the Steiner Forest problem with non-trivial approximation ratio has worst-case expected time complexity in* $\tilde{\Omega}(\min\{s, \sqrt{n}\} + k + D)$.

The proof of Theorem 3.2 in fact consists of proving the following two separate lower bounds.

LEMMA 3.3. *Any distributed algorithm for* DSF-IC *with finite approximation ratio has time complexity* $\Omega(k/\log n)$*. This is true even for unweighted graphs of diameter 3.*

LEMMA 3.4. *For* $s \in \mathcal{O}(\sqrt{n})$*, any distributed algorithm for* DSF-IC *or* DSF-CR *with finite approximation ratio requires* $\Omega(s/\log n)$ *rounds. This holds even for instances with* $t = 2$*,* $k = 1$*, and* $D \in \mathcal{O}(\log n)$.

We remark that the proofs of Lemmas 3.1, 3.3, and 3.4 are by reductions from Set Disjointness (see, e.g., [17]). In Lemmas 3.1 and 3.3, it is trivial to increase the other parameters, i.e., D, s, t, or n, so we may apply Lemmas 2.3 and 2.4 to obtain a minimal instance of DSF-IC without affecting the asymptotic time complexity.

4. DETERMINISTIC ALGORITHM

We start the description of our algorithm with a quick survey of known results that we shall use.

The "red" and "blue" rules. Tarjan [25] was the first to formulate the dual rules for MST construction in the following way. Define a *cut* to be a partition of the node set.

- An edge which is the lightest edge across *some* cut of the graph is called *blue*. An edge known to be blue is a part of the MST.
- An edge which is the heaviest edge in *some* cycle of the graph is called *red*. An edge proved to be red is not a part of the MST.

Kruskal's algorithm. Kruskal's algorithm constructs an MST by scanning the edge set in order of ascending weight and selecting each edge that does not close a cycle. The correctness of this procedure follows from the red rule.

A distributed version of this idea is presented in [13]. The edges are convergecast over a BFS tree, where each tree node sends them to its parent in ascending order and discards heavy cycle-closing edges. They show that nice pipelining is achieved: the root learns the first l MST edges in $l + \mathcal{O}(D)$ rounds; by broadcasting them over the BFS tree, these edges become common knowledge at the same asymptotic time complexity (instead of the trivial $\mathcal{O}(|E|)$ complexity).

Prim's/GHS algorithm. Prim's algorithm constructs an MST by, starting from an arbitrary node, adding in each step the lightest edge leaving the so far connected node set to the tree. The correctness of this algorithm follows from the blue rule.

Prim's algorithm can be parallelized as follows: starting with no selected edges, in each step, each connected component (initially just isolated nodes) selects its least-weight outgoing edge, until there is a single connected component, which is the MST. Using the selected edges for communication within the components, one obtains the Gallager-Humblet-Spira algorithm [12]. Since in each step the minimum size of a connected component at least doubles, the running time is $\mathcal{O}(\mu \log n)$, where $\mu < n$ is the hop diameter of the MST. In [12], this bound is, in fact, derived in the asynchronous setting using few messages. For the synchronous case, one can reduce the time complexity to $\mathcal{O}(\mu)$ by allowing only components of strong diameter at most 2^i to select edges. Note, however, that possibly $\mu = n - 1$, irrespective of other parameters like D or s.

GKP algorithm. Garay, Kutten, and Peleg [13, 18] show how to combine the distributed variants of Kruskal's and Prim's algorithm to obtain an algorithm of running time $\mathcal{O}(\sqrt{n \log^* n} + D)$, which is optimal up to a factor of $\mathcal{O}(\sqrt{\log n \log^* n})$, even for approximate solutions [9, 23]. The key idea is to first apply the GHS approach until components contain at least $\kappa := \sqrt{n/\log^* n}$ nodes, and then

connect the remaining at most $n/\kappa = \sqrt{n \log^* n}$ components by the pipelined variant of Kruskal's algorithm, taking $\mathcal{O}(\sqrt{n \log^* n} + D)$ rounds.

A central challenge in realizing this approach is that, for the second stage of the algorithm, two nodes must be able to determine whether they belong to the same component. In [18], this is solved efficiently by breaking down the components from the first stage into $\mathcal{O}(n/\kappa)$ smaller pieces of strong diameter $\mathcal{O}(\kappa)$ using a recursive procedure determining $\mathcal{O}(\kappa)$-hop dominating sets of size $\mathcal{O}(n/\kappa)$.

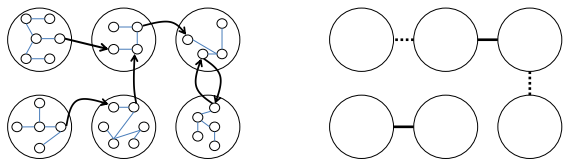

Figure 1: *Example for component growing. Left: each component is a big circle, arrows represent proposed edges. Right: Thick lines represent matching edges, dashed lines represent edges from isolated components.*

As a secondary contribution, in this work we propose a simple solution to this problem, avoiding to grow components too much in the first place. Instead of adding all proposed edges (i.e., least-weight outgoing edges of small components) in each step, we build a graph whose nodes are components and whose edges are the proposed edges; we then emulate a maximal matching algorithm on this graph, and add only the matching edges and those proposed by components which were not matched (see Figure 1). This ensures that small components are still guaranteed to be merged, but the longest "merging paths" in the component graph have at most 3 hops. Since each component proposes at most one edge, a maximal matching can be computed efficiently by simulating the Cole-Vishkin algorithm [7] on the component graph. We note that this variant has the same asymptotic time complexity as the original GKP algorithm.

SF Approximation via the Terminal Graph. The *terminal graph* G_T is the complete graph on the node set T with edge weights of $\mathrm{wd}(v, w)$, i.e., the terminal-terminal distances in G. Mapping an optimal solution to the Steiner Forest problem on G_T to G by mapping an edge in G_T to a shortest path in G yields a 2-approximation in G [19, 24].

The Bellman-Ford Algorithm (BF). "Finding" a terminal graph edge amounts to determining the distance between its terminals in G. Having a canonical interpretation as distributed algorithm, the Bellman-Ford (henceforth, "BF") algorithm is an obvious choice for this task. The single-source version of BF runs for at most s rounds and, by maintaining a pointer to the node whose message gave rise to the most recent distance update, constructs a distributed representation of a shortest-path-tree rooted at the source.

Moat Growing Algorithm. For the Steiner Forest problem, the best known polynomial-time approximation ratio is, essentially, 2. It is achieved by the centralized *moat growing* algorithm proposed by Agrawal, Ravi, and Klein [1]. The algorithm proceeds as follows.

1. Grow weighted balls around each terminal up to the radius where a pair of balls centered at v and w touch.[2]

[2]Recall that we assume w.l.o.g. that different paths have different weights.

2. Contract each ball to a node. This merges v and w into a single terminal u.

3. Add the edges of a shortest path from v to w to the solution (if v or w are contracted balls, connect the pair of original terminals $v' \in v$ and $w' \in w$ closest to each other).

4. Merge the input components of v and w, if they are different (they are now connected anyway).

5. If the input component of u contains a single terminal (u), delete that component (i.e., its terminals become regular nodes).

6. If terminals remain, go to Step 1.

Note that in the special case of a Steiner Tree, the only deletion terminates the algorithm. The algorithm thus simulates Kruskal's algorithm on the terminal graph and maps the computed MST back to G as a 2-approximation. In general, such an analogy does not hold in a straightforward way, as the combination of deletion and contraction operation changes distances in the terminal graph; refraining from contractions and simply deleting connected components whose terminals are unions of a subset of the input components yields different results.

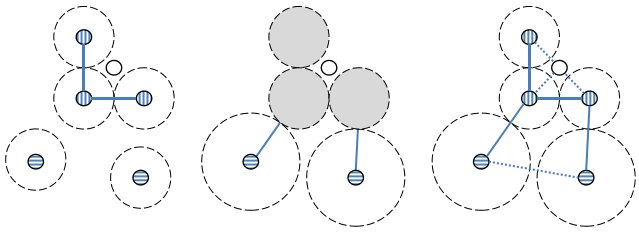

Figure 2: *Example for moat growing. circles with horizontal lines are terminals of Input Component (IC) 1, and vertical lines indicate IC 2. The white circle is a non-terminal node. Left: after growing moats, moats of IC 1 meet, path corresponding to the think lines are chosen, and the new moat is inactivated. Middle: the moats of terminals from IC 2 grow and meet with the inactive moat, adding the new thick lines. Right: The solution of the algorithm (thick lines) and the optimal solution (dotted lines).*

The proof that the algorithm yields a 2-approximation for the general case, i.e., arbitrary number of input components k, is an elegant primal-dual argument based on the observation that, in each step, the weight of the intersection of an optimal solution with the balls is at least the number of (current) balls times their radius (see Fig. 2): the center of each ball must be connected to *some* other terminal outside the ball. On the other hand, terminals are connected by shortest paths, whose intersection with the two terminal's balls they connect weighs at most twice the balls' radius. The ingenuity of the argument lies in the fact that it allows for using contracted balls whose centers have been deleted to be used as shortcuts—the respective path weights inside the contracted balls have been accounted for, as *any* forest connecting the ball centers by shortest paths intersecting only two balls each requires less than twice the budget of an optimal solution.

4.1 Key Ideas and Main Results

Voronoi Decompositions and the Terminal Graph. The key observation underlying our algorithm is that certain Voronoi (or closely related) decompositions of G yield

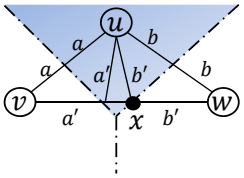

Figure 3: *v, u, w are terminals and dashed lines are borders between Voronoi regions. If a shortest path between v and w crosses the region of u (shaded), then* $\mathrm{wd}(v,w) = \mathrm{wd}(v,x) + \mathrm{wd}(x,w) > 2b' \geq 2b$. *Similarly,* $\mathrm{wd}(v,w) > 2a$.

crucial information on the terminal graph. A Voronoi decomposition of G w.r.t. the terminals T is a partition of the graph into regions so that $u \in \mathrm{Vor}_v$ (for some $u \in V$ and $v \in T$) iff $v = \mathrm{argmin}_{w \in T}\{\mathrm{wd}(w,u)\}$. Such a decomposition and the corresponding distances $\mathrm{wd}(u,v)$ for $u \in \mathrm{Vor}_v$ can be computed in $\mathcal{O}(s)$ rounds using the Bellman-Ford algorithm. To see this, connect all terminals to a virtual source node by edges of weight 0 and piggy-back on all messages (and store) the identifier of the terminal through which the indicated path to the virtual node passes.

The utility of the resulting distributed data structure is illustrated by the special case of a Steiner Tree. Recall that an MST of the terminal graph induces a 2-approximate solution. let $\{v,w\}$ be an MST edge in the terminal graph, and consider any shortest path p from v to w in G. The salient point is that p must be contained in $\mathrm{Vor}_v \cup \mathrm{Vor}_w$. Otherwise, p would pass through Vor_u for some $u \in T$, implying that $\mathrm{wd}(v,u) < \mathrm{wd}(v,w)$ and $\mathrm{wd}(w,u) < \mathrm{wd}(v,w)$ (see Fig. 3). In other words, the terminal graph edge $\{v,w\}$ is the heaviest edge of the cycle (v,w,u,v) in the terminal graph, contradicting the red rule.

This basic insight already gives us the tools to derive an efficient distributed algorithm for 2-approximating a Steiner Tree based on the GKP algorithm:

1. Compute the Voronoi decomposition of G w.r.t. T.
2. Simulate GHS on the terminal graph up to components of at least $\sqrt{n}$ nodes and strong diameter at most $\mathcal{O}(\sqrt{n} + s)$ *in* G (each step does not only merge components, but also add a path of up to s hops). Note that nodes on the selected paths learn that respective edges in G are selected, as communication is routed via them.
3. Simulate Kruskal's algorithm on the terminal graph, starting from the already selected edge set.
4. For each terminal edge $\{v,w\}$ selected in the previous stage, notify the nodes on the corresponding path that the path's edges have been selected. This is done by sending tokens from the nodes of the path edge crossing the boundary between Vor_v and Vor_w to v and w, respectively, using the pointers inducing the shortest-path-trees constructed in the first step.

Note that in the last step, it suffices that each node in the shortest-path-tree of, e.g., node $v \in T$, sends only the first received token to its parent—we do not care how many paths use a specific edge in G. Hence, the overall running time becomes $\tilde{\mathcal{O}}(s + \sqrt{n})$.

One can refine the approach slightly by capping component's growth in the GHS phase of the algorithm at the following strong diameter value.

NOTATION 4.1. *We use* $\sigma \stackrel{\text{def}}{=} \sqrt{\min\{st, n\}}$.

The point is that the constructed solution will consist of the union of $t-1$ paths of at most s hops. Capping component's growth at σ improves the running time to $\tilde{\mathcal{O}}(s + \sigma)$. Our algorithm generalizes this idea.

COROLLARY 4.2. *For $k = 1$, a solution to* DSF-IC *that is optimal up to factor 2 can be computed deterministically in $\tilde{\mathcal{O}}(s + \sigma)$ rounds.*

Note that if $s > t$, the first term dominates the complexity. Hence this bound is always better than plain application of Kruskal's selection scheme, which would yield running time $\mathcal{O}(s + t)$.

Moat Growing without Contraction. In light of the above and the aforementioned fact that for $k = 1$ the moat growing algorithm also constructs an MST of the terminal graph, it is natural to seek to extend the existing distributed techniques in order to implement the moat growing algorithm. The main obstacle to an efficient implementation is the contraction of moats. Simulating operations on a graph in which we recursively contract long paths directly will be very slow: a single round of the Bellmann-Ford algorithm would require to route communication through a contracted region whose strong diameter satisfies only trivial upper bounds; on the other hand, trying to route communication through a BFS tree will result in high congestion.

Instead, we take a different point of view, avoiding contractions (this can be seen as a simulation argument). A considerable part of our technical contribution consists of rephrasing the centralized moat growing algorithm, adapting the proof of its approximation ratio to this formulation, and finally deriving and proving equivalent the distributed version of the algorithm. In a nutshell, we replace the contraction operation by electing a leader of each *moat*, a set of terminals that are connected by the currently selected edges. A moat can be *active*, corresponding to a non-deleted merged terminal in the original algorithm, or *inactive*. The latter state is attained when the moat is a union of whole input components and therefore would be deleted; we simply "halt" the growth of its surrounding "ball". Such a "ball" now is the *union* of weighted balls around the terminals, and when any two balls of terminals from different moats touch, the respective moats are *merged* and a shortest path connecting the centers of the touching balls is added.

Note that this differs from the original algorithm in that we may merge an active moat M_1 with inactive moat M_2 into an active moat M_{12}. In contrast, the original algorithm would have deleted the terminal representing the contracted region of M_2. The crucial observations are that with this modification,

(i) M_2 interferes only *twice* with "distances", when it is formed (the balls of its terminals stop growing) and when it gets merged (the balls start growing again); and

(ii) if later M_{12} gets merged with a moat M_3 so that the respective path connects a (formerly inactive) terminal of M_2 with one of M_3, we constructed a path that the original moat growing algorithm would have selected as well: the original algorithm would have grown the ball of M_1 "through" the contracted region associated with M_2 (represented by the corresponding merge operation), and once the balls of M_1 and M_3 touched, the selected shortest path would pass via the contracted region of M_2 that became part of $M_1 \cup M_2$ in our approach.

Observation (i) shows why we can implement the algorithm efficiently with this modification: for each input component, at most once a moat becomes inactive (stopping to grow its balls) and at most one inactive moat merges with an active moat (causing an "instantaneous" extension of the active moat); in both cases we recompute a suitable Voronoi decomposition of the graph, requiring in total $\mathcal{O}(sk)$ rounds throughout the entire algorithm.

Observation (ii) explains why forming connections to inactive moats does not render the algorithm useless. While the resulting forest may very well not be a 2-approximate solution, (ii) says that it *contains* a subforest that solves the problem and has cost at most twice the optimum. Therefore, to "fix" this issue, it suffices to postprocess the computed forest, leaving only the edges required to satisfy the requested connectivity.

The moat growing mechanism can be viewed as Kruskal's algorithm on the terminal graph, *but with the terminal graph changing up to* $2k$ *times* during the execution of the algorithm. On each such event, we recompute a suitably adapted Voronoi decomposition and recommence executing the pipelined version of Kruskal's edge selection scheme until the next such change is encountered. When this procedure is done, the root of the BFS tree knows all selected terminal graph edges. Hence the root can compute the minimal subset satisfying all constraints (learning each terminal's input label takes $\mathcal{O}(t+D)$ rounds over the BFS tree), and broadcast the result to all nodes. Finally, the underlying edges in G can be marked in $\mathcal{O}(s)$ rounds as before. The result is a 2-approximation of SF with running time $\tilde{\mathcal{O}}(sk+t)$, as at most $t-1$ terminal graph edges are to be selected.

THEOREM 4.3. DSF-IC *can be solved deterministically with approximation factor* 2 *in* $\mathcal{O}(sk+t)$ *rounds.*

Combining Moat Growing and Prim's Approach. Interpreting the moat growing algorithm as a generalization of Kruskal's algorithm raises the question of whether we can also here combine it with the GHS approach to achieve a time complexity that is sublinear in the number of terminals. This is possible, but it is more involved than the special cases of MST and Steiner Tree. Intuitively, this is because edge weights in the terminal graph are dynamic. More concretely, while scanning edges in ascending order of weight (as in Kruskal's and the moat growing algorithm), we also need to recompute distance information on every merge in which a participating or the resulting moat is inactive. In contrast, naive application of Prim's selection scheme may merge two moats under the assumption that they remain active up to the ball radii corresponding to the respective path weight, but this assumption may turn out to be wrong due to merges for smaller radii the algorithm has not determined yet (see Fig. 4).

The distance information available from the Voronoi decomposition can be invalidated due to an inactive moat either forming, or being merged. We handle these cases differently. First, we defer moats inactivation to ball radii that are integral powers of $1+\varepsilon$, for a given constant $\varepsilon > 0$. Since the maximal ball radius required to merge all moats is polynomial in n, it follows that there at most $i_{\max} \stackrel{\text{def}}{=} \lfloor \log_{1+\varepsilon} \mathrm{WD} \rfloor + 1 \in \mathcal{O}(\log n/\varepsilon)$ different radii at which we need to recompute the Voronoi decomposition due to inactivation of moats: for each power $i \in \{1,\dots,i_{\max}\}$, we determine all merges that correspond to moat radius in $\big[(1+\varepsilon)^{i-1}/2, (1+\varepsilon)^i/2\big)$. It is not hard to show that the approximation ratio remains bounded by $2(1+\varepsilon)$.

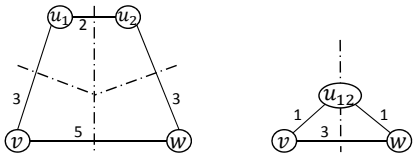

Figure 4: *Example for moat order-changing contraction. Left: numbers indicate terminal-terminal distance and dashed lines represent region borders. Right: after growing the radius to* 1, u_1 *and* u_2 *merge into a single inactive moat which is a shortcut between* v *and* w.

The second issue is when an inactive moat is merged with an active one, resulting in resumed moat growing. This is solved within a *growth phase* (i.e., a range of radii as above) as follows. Since all balls around active terminals grow at the same rate, we determine the minimal increase in radius to see when an active terminal's moat touches a an inactive moat v. We then repeat distance calculations from the corresponding radius on, where all terminals in the moat of v are now active, find the next inactive moat that is merged, and so on, until the maximal radius of balls reaches the next power of $1+\varepsilon$. This allows us to collect all the information on terminal graph edges that we need for the current growth phase.

Using this process, we can iteratively compute *all* the decompositions needed to provide correct distance information between moats throughout the growth phase. Based on this knowledge, we first grow moats locally and concurrently as in the GHS algorithm, until components either contain σ nodes, or have no further outgoing edges that may be selected during the current growth phase (this happens in phase i if all outgoing edges have weight at least $(1+\varepsilon)^i/2$). Subsequently, we apply the pipelined variant of Kruskal's algorithm to the edges connecting different components that have weight at most $(1+\varepsilon)^i/2$. Note that, as before, it is essential that all terminals in a component refer to it by the same identifier in order to avoid closing cycles by choosing edges that connect terminals of the same component. Due to the iterative approach switching back and forth between Prim and Kruskal type edge selection, there can be no guarantee that components are small. However, there are at most $\min\{st,n\}/\sigma = \sigma$ components of strong diameter larger than σ; for the purpose of selecting an identifier (e.g., the largest node identifier in the component), we simply pipeline their communication over a single BFS tree, taking $\mathcal{O}(\sigma+D)$ rounds in each of the $\mathcal{O}(\log n)$ growth phases.

COROLLARY 4.4. *For any instance of* DSF-IC*, a solving forest* F *with the property that its minimal subforest solving the instance is optimal up to factor* $2+\varepsilon$ *can be computed in* $\tilde{\mathcal{O}}(sk+\sigma)$ *rounds.*

Fast Pruning of Unneeded Edges. By Corollary 4.4, we now face the following setting. We have a forest of edges that connect any two terminals in the same component by a unique path, but some edges are superfluous. We are guaranteed that the minimal subforest that is a solution is a $(2+\varepsilon)$-approximation. Discarding the unnecessary edges is trivial centrally, but may be costly distributively: e.g., col-

lecting the information at a single node may result in $\Omega(t)$ time; also, the tree depths could be as large as $\Omega(n)$.

Our approach is to decompose the problem into a few subproblems; we first solve the high-level problem and then, in parallel, all subproblems. Specifically, we first decompose the forest into clusters of strong diameter $\tilde{\mathcal{O}}(\sigma)$ that contain at least σ nodes each (see Fig. 5). This can be done, say, by running the first stage of the GKP algorithm up to size σ, in $\tilde{\mathcal{O}}(\sigma)$ rounds. Note that there are at most σ clusters. We then focus on the *cluster forest* graph, denoted FC, in which each cluster is a node, and two nodes are connected iff the corresponding clusters are connected by an edge (at most $\sigma - 1$ edges). FC is broadcast to all nodes at the cost of $\mathcal{O}(\sigma + D)$ rounds.

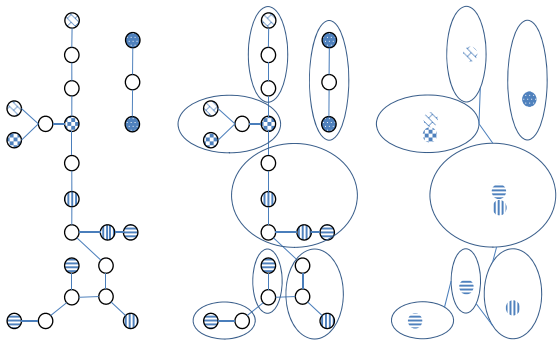

Figure 5: *First steps of pruning. left: the initial forest. The pattern in the circles indicate the input component (empty circles are non-terminals). Middle: clustering. Right: The cluster forest FC before the label merging.*

Next, we collect the label information of FC over the global BFS tree. A naïve implementation may cost $\Omega(k\sigma)$ time, because each of the clusters may be labeled by $\Omega(k)$ labels. Instead, we use the following rule for disseminating the necessary information to all nodes. Nodes send up the BFS tree messages of the form (C, λ), meaning "Cluster C contains a terminal with label λ." Terminals have the base information. When a node v receives a message (C, λ), it applies the following changes to its local copy of FC:

(a) Label C by λ.

(b) If there is another cluster C' labeled λ, label by λ all clusters and edges along the unique path connecting C to C' in FC.

(c) If there is now some edge of FC labeled both λ and λ', then these labels are merged in the sense that any cluster or edge labeled either λ or λ' is relabeled by both λ and λ'.

(d) If the local copy of FC has changed, enqueue the (C, λ) message for forwarding it to the parent.

Step c is justified by the observation that in the solution we seek, the input component set is divided into equivalence classes defined by the connected components of the solution: if two input components λ and λ' are connected in the solution, they are in the same equivalence class, and it is these classes that we need.

The important advantage of this approach is that for any given node, there can be at most $\mathcal{O}(\sigma + k)$ messages that pass the filter of Step d. To see this, observe that whenever the local copy of FC changes, it holds that either (i) a new label $\lambda \in \Lambda$ is added that was not present before, or (ii) an edge receives its first label, or (iii) at least two (current) equivalence classes of labels are merged. Clearly, (i) and (ii) can apply at most k times each, while (ii) may happen no more than $\sigma-1$ times. It follows that every node sends $\mathcal{O}(\sigma+k)$ messages, and thus, after $\mathcal{O}(\sigma + k + D)$ rounds, the root knows the full (new) labeling of FC; this information then is disseminated to all nodes in the same asymptotic running time, by the root broadcasting the non-filtered labels over the BFS tree.

This information allows the nodes to identify the inter-cluster edges that are required in the output. Inside each cluster, we can solve the problem directly: Each node needs to know what labels are at the other end of each of its incident edges (a terminal has its original label, and the labels of inter-cluster edges are known by the previous stage). Selecting intra-cluster edges takes $\mathcal{O}(\sigma + k)$ rounds, because the cluster diameter is $\mathcal{O}(\sigma)$ and there are at most k labels to propagate. Thus, the pruning procedure yields the main result of this section.

THEOREM 4.5. *For any constant $\varepsilon > 0$, a deterministic distributed algorithm can compute a solution for problem* DSF-IC *that is optimal up to factor $(2 + \varepsilon)$ in $\tilde{\mathcal{O}}(sk + \sqrt{\min\{st, n\}})$ rounds.*

Finally, we preprocess the input to make the instance minimal using Lemma 2.4 and note that, because weights are integer, there can be at most $2\,\mathrm{WD}$ different moat radii at which moats are merged and thus recomputation of the Voronoi decomposition may be required.

COROLLARY 4.6. *For any constant $\varepsilon > 0$, a deterministic distributed algorithm can compute a $(2+\varepsilon)$-optimal solution for problem* DSF-IC *in $\tilde{\mathcal{O}}(s \min\{k_0, WD\} + \sqrt{\min\{st, n\}} + k + D)$ rounds, where k_0 is the number of input components with at least two terminals.*

5. RANDOMIZED ALGORITHM

Again, we start by describing the known results and techniques our randomized algorithm makes use of.

Probabilistic Tree Embedding. A *metric embedding* of a metric space (X, d_X) is an injective mapping $\iota : X \to Y$, where (Y, d_Y) is a metric space and for all $x, x' \in X$ it holds that $d_X(x, x') \leq d_Y(\iota(x), \iota(y))$. A *tree embedding* is a metric embedding for which d_Y is the distance metric of a weighted tree on Y. A *probabilistic tree embedding of expected stretch* ρ is a distribution of metric tree embeddings satisfying that $\mathbb{E}[d_Y(\iota(x), \iota(y))] \leq \rho \cdot d_X(x, x')$ for all $x, x' \in X$.

Probabilistic tree embeddings have proven to be an excellent tool for deriving probabilistic approximation algorithms to hard problems, granted that the cost function is the sum of edge weights of the solution. Instead of solving the problem on (X, d_X), one solves the problem on the instance on (Y, d_Y) sampled from the distribution, translating the instance by means of ι. Since (Y, d_Y) is a tree, the problem is much simpler to solve, yet in expectation the cost of an optimal solution increases by at most ρ. This strategy lies at the heart of our randomized algorithm.

FRT Embedding. The FRT algorithm [11] produces a probabilistic tree embedding of G (i.e., $(X, d_x) = (V, \mathrm{wd})$) with expected stretch $\mathcal{O}(\log n)$. It works as follows.

1. Select uniformly at random: a permutation $r : V \to [n]$, and a real $\beta \in [1, 2)$.

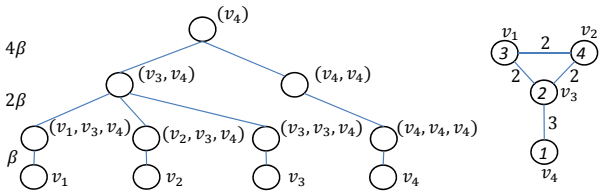

Figure 6: *FRT embedding example, with $n = 4$ and $L = 2$. Right: original graph, permutation values inside circles. Left: tree, edge weights indicated at the left hand side.*

2. Set $L := \lceil \log W \rceil \in \mathcal{O}(\log n)$. For each $v \in V$ and $i \in \{0, \ldots, L\}$, define $v_i := \operatorname{argmin}_{v \in B_G(v, \beta 2^i)}\{r(v)\}$.
3. Set $Y := V \cup \{(v_i, \ldots, v_L) \mid v \in V \wedge i \in \{0, \ldots, L\}\}$ and $\iota(v) := v$ for all $v \in V$.
4. For each $v \in V$, there is an edge $\{v, (v_0, \ldots, v_L)\}$ of weight β and, for each $v \in V$ and $i \in \{1, \ldots, L\}$, there are edges $\{(v_{i-1}, \ldots, v_L), (v_i, \ldots, v_L)\}$ of weight $\beta 2^i$ (see Fig. 6). The metric d_Y is the distance metric of the weighted tree on Y defined by these edges and weights.

SF Approximation Based on FRT Embedding. We can solve DSF-IC as follows. (1) Map an instance of DSF-IC to the virtual tree (Y, d_Y), (2) solve the instance optimally on (Y, d_Y) by selecting, for each label λ, the minimal subtree containing all $v \in T$ with $\lambda(v) = \lambda$ (rooted at their least common ancestor), and (3) map the solution back to G by selecting, for each solution edge between child c and parent $p = (v_i, \ldots, v_L)$, an arbitrary terminal t in c's subtree, and outputting the edges of a shortest path from t to v_i in G.

Now, the solution is optimal in (Y, d_Y), and hence its image in G has weight $\mathcal{O}(\log n)$ times the optimum in G.

Khan et al. [16] give a distributed implementation of this scheme running in time $\tilde{\mathcal{O}}(sk)$ w.h.p., based on *LE lists*.

LE Lists: definition and computation. Given a uniformly random permutation of the nodes, the *Least Elements (LE)* list L_v of node $v \in V$ is the list of pairs $(w, \mathrm{wd}(v, w))$ for all $w \in V$, ordered in increasing distance $\mathrm{wd}(v, w)$, after deleting all nodes whose rank is larger than the rank of some prior (closer) node. Given β and its LE list, v can easily find $v_0, \ldots, v_L$ and a neighbor w which is the next hop on a shortest path from v to v_i (for a given i) such that $(v_i, \mathrm{wd}(w, v_i)) \in L_w$. LE lists therefore provide a distributed data structure representing the virtual tree, and the aforementioned distributed $\mathcal{O}(\log n)$-approximation in $\tilde{\mathcal{O}}(sk)$ time is straightforward to find, given these lists.

We now sketch the algorithm computing LE lists from [16]. First, each node v selects a uniformly random number $r(v)$ of $c \log n$ bits (c is large enough to ensure that all $r(v)$ are distinct w.h.p.). Then, after initializing $L_v := \{(v, 0, r(v))\}$, the following is repeated until no L_v variable is modified:

1. Each node v sends L_v to all neighbors.
2. For each $(u, d, r(u))$ received from neighbor w, each node v sets $L_v \cup \{(u, d + W(v, w), r(u))\}$.
3. Each node scans L_v in increasing weight order (second entries), deleting tuples whose rank (third entry) is not larger than all previous ones.

This algorithm terminates after at most s iterations. One can show that w.h.p., the LE lists at each node, at any time, have $\mathcal{O}(\log n)$ entries, and therefore the algorithm terminates in $\tilde{\mathcal{O}}(s)$ rounds w.h.p.

Spanner Construction. A *ρ-spanner* of a graph is a graph obtained by deleting edges so that distance increase at most by factor ρ. Typically one is interested in sparse spanners, i.e., those that have few edges. A distributed randomized spanner construction that can be directly run in the CONGEST model was given by Baswana and Sen [2]. It computes a $(2\kappa - 1)$-spanner with $\mathcal{O}(n^{1+1/\kappa})$ expected edges in $\mathcal{O}(\kappa)$ rounds, for $\kappa \in \mathbb{N}$.

Skeleton Graph and Skeleton Spanner. We use a technique introduced in [19], the construction of a spanner of a *skeleton graph*. A skeleton graph is defined similarly to the terminal graph, with some important variations. Given a subset $\mathcal{S} \subseteq V$ and a parameter $h \in \mathbb{N}$, the skeleton graph has nodes $\mathcal{S}$ and edges $\{\{v, w\} \mid \exists p \in \mathcal{P}(v, w) : \ell(p) \leq h\}$, where an edge $\{v, w\}$ has weight $\min_{p \in \mathcal{P}(v,w): \ell(p) \leq h}\{W(p)\}$.

Now, let $\mathcal{S}$ be a random set, where each node is included independently with probability $1/\sqrt{n}$. Then any node set (including paths) with $\Omega(\sqrt{n} \log n)$ nodes contains a node from $\mathcal{S}$ w.h.p. Hence, taking such a random $\mathcal{S}$ and setting $h = \sqrt{n} \log n$, we obtain a skeleton graph in which the distances are, w.h.p., identical to the corresponding distances (between nodes of $\mathcal{S}$) in G.

The number of edges in the skeleton graph may be as large as $\Theta(|\mathcal{S}|^2) = \Theta(n)$. In [19], we show how to simulate the Baswana-Sen algorithm on the skeleton graph to obtain a $(2\kappa - 1)$-spanner of the skeleton graph with $\tilde{\mathcal{O}}(n^{1/2+1/(2\kappa)})$ edges w.h.p., within $\tilde{\mathcal{O}}(n^{1/2+1/(2\kappa)} + D)$ rounds. This edge set can be made known to all nodes in the same asymptotic time, and nodes on the corresponding paths in G will know the respective next hops on the paths.

SF Approximation Using Skeleton Spanner. For a given instance of DSF-IC, if we add the set of terminals T to $\mathcal{S}$ (in addition to the random sample), the above algorithm determines and makes known to all nodes a $(2\kappa - 1)$-spanner of the skeleton graph with $\tilde{\mathcal{O}}((n^{1/2} + t)^{1+1/\kappa})$ edges in time $\tilde{\mathcal{O}}((n^{1/2} + t)^{1+1/\kappa} + D)$ w.h.p. In other words, all nodes obtain knowledge of a metric embedding of the terminal graph of (worst-case) stretch $2\kappa - 1$.

After broadcasting the labels $\lambda(v)$, $v \in T$, over a BFS tree to all nodes in $\mathcal{O}(t + D)$ rounds, each node deterministically solves the instance of DSF-IC on the skeleton spanner locally by a centralized α-approximation algorithm.[3] The result is interpreted as an $\alpha(2\kappa - 1)$-approximate solution on the terminal graph. As discussed in Section 4, this in turn induces a $2\alpha(2\kappa - 1)$-approximate solution of the original instance on G. Mapping the locally computed solution back to G is trivial, as the nodes on the corresponding paths know their incident edges on such paths. Overall, for $\alpha \in \mathcal{O}(1)$, we obtain a randomized $\mathcal{O}(\kappa)$-approximation to DSF-IC in $\tilde{\mathcal{O}}((n^{1/2} + t)^{1+1/\kappa} + D)$ rounds w.h.p.

5.1 Key Ideas and Main Results

Pipelining of FRT Edge Selection. Our first observation is that it is possible to exploit the fact that LE lists have $\mathcal{O}(\log n)$ entries w.h.p. to improve on the construction from [16]. From a high-level perspective, our algorithm proceeds as follows. Initially, each terminal v has label $\{\lambda(v)\}$. Then, for $i = 0, \ldots, L$, all nodes execute:

1. Each active terminal v *moves* (i.e., sends and subsequently deletes) its label to v_i. These messages are routed in G according to the LE lists, and all traversed edges are

[3] Finding an exact solution is NP-hard.

added to the solution in G (this corresponds to selecting and mapping back to G the virtual tree edge between $(v_i, \dots, v_L)$ and $(v_{i-1}, \dots, v_L)$ or, if $i = 0$, v).

2. Delete any label which is received by a single node v_i (because $(v_i, \dots, v_L)$ is the least common ancestor of all terminals with that label).

3. For each label that v_i received and did not delete, it picks a terminal v that sent the label to v_i and moves the label back to v; here all labels sent by terminals in the same subtree of the virtual tree are sent to the same terminal. This is done by backtracking sending routes.

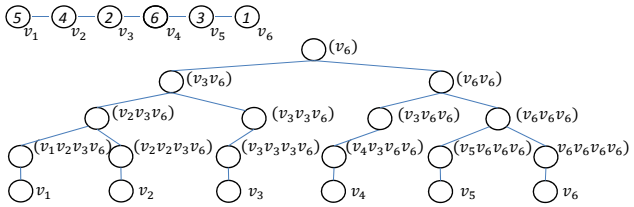

Figure 7: *Another FRT embedding example. Top Left: original graph, permutation values inside circles. In the tree, v_3 appears in both $(v_3v_3v_6)$ and $(v_3v_6v_6)$.*

Given as local inputs the LE lists and β (i.e., the distributed representation of the FRT embedding), this *almost* implements the strategy for deriving an expected $\mathcal{O}(\log n)$-approximation outlined earlier (see Fig. 7). The difference is that on the virtual tree, v_i may correspond to multiple nodes $(v_i, \dots, v_L)$, $(w_i = v_i, \dots, w_L)$, etc. However, v_i picks only a single terminal u so that $u_i = v_i$ to send a given received (non-deleted) label to. Hence, the algorithm will only select paths corresponding to a subset of the edges of the virtual tree solution. Nonetheless, we obtain a valid solution: Since all terminals that sent a label to v_i are already connected by the selected edge set, for the purpose of the Steiner Forest problem, there is no need to uphold the distinction that they are in different subtrees of the virtual tree.

This procedure enables pipelining, thus avoiding the multiplicative complexity of $\tilde{\mathcal{O}}(sk)$ of [16]. Since the messages are routed on least-weight paths, all messages destined at a node v_i are sent over a shortest-paths tree rooted at v_i. This tree has depth at most s by definition. Because it does not matter from which particular terminal v_i receives a given label λ, it suffices if each node forwards only the *first* message λ destined to v_i to its parent in the tree. Hence, if there was a single tree only and each node transmitted a single label in each round (guaranteeing message size $\mathcal{O}(\log n)$), all labels would be delivered to v_i within at most $s + k$ rounds.

In general, however, each node participates in multiple trees, and if the same neighbor serves as its parent in a few of them, congestion occurs. Fortunately (w.h.p.) the LE lists contain only $\mathcal{O}(\log n)$ entries, and therefore, each node participates in at most $\mathcal{O}(\log n)$ shortest-paths trees used for routing. It follows that by using time-multiplexing, we can guarantee that for each of the trees it participates in, each node sends at least one message every $\mathcal{O}(\log n)$ rounds. Since we can determine in $\mathcal{O}(k + D)$ rounds whether a label is held by a single node only, we conclude that each loop iteration can be completed within $\mathcal{O}((s + k) \log n + D) \subseteq \tilde{\mathcal{O}}(s + k)$ rounds w.h.p., and since $L \in \mathcal{O}(\log n)$, the entire procedure completes within $\tilde{\mathcal{O}}(s + k)$ rounds.

THEOREM 5.1. *An $\mathcal{O}(\log n)$-optimal solution to problem* DSF-IC *can be computed in $\tilde{\mathcal{O}}(s + k)$ rounds w.h.p.*

Partial FRT Construction plus Skeleton Spanner. If $s > \sqrt{n}$, the above bound does not match the lower bound of Theorem 3.2. The problem is that in the FRT construction information travels long routes. We can avoid this issue by performing the construction only partially and leveraging the spanner construction for the top levels. The key is that the $\sqrt{n}$ nodes of largest rank are a random subset of V. Therefore we do the following.

1. Let $\mathcal{S}$ be the set consisting of the $\sqrt{n}$ nodes of highest rank. Delete $\mathcal{S}$ from the FRT tree.

2. For each node v, let i_v be such that $(v_{i_v}, \dots, v_L)$ is the first deleted ancestor. Replace $(v_{i_v}, \dots, v_L)$ by $\tilde{v}_{i_v} := \operatorname{argmin}_{w \in \mathcal{S}}\{\mathrm{wd}(v, w)\}$, the node from $\mathcal{S}$ closest to v.

3. Run the FRT-based algorithm from above on the resulting (virtual) forest.

As on each path a sampled node is encountered within $\mathcal{O}(\sqrt{n} \log n)$ hops w.h.p., this routine can be executed within $\tilde{\mathcal{O}}(\sqrt{n} + k + D)$ rounds w.h.p.—the maximal depth of the shortest-path-trees used for routing becomes $\tilde{\mathcal{O}}(\sqrt{n})$ w.h.p., replacing the additive s in the complexity. We note the following crucial properties of the selected edge set:

- The total cost of all selected edges is bounded from above by the cost of the optimal solution on the virtual tree: abstractly speaking, we deleted all edges between nodes in $\mathcal{S}$, and replaced edges between nodes in $\mathcal{S}$ and $V \setminus \mathcal{S}$ by "lighter" ones (terminals connect to the closer $\tilde{v}_{i_v}$ instead of $(v_{i_v}, \dots, v_L)$). Thus we selected edges of weight $\mathcal{O}(\log n)$ times the optimum in expectation.
- Recall that the optimal solution on the virtual tree is the union, over all λ, of the minimal trees spanning all terminals v with $\lambda(v) = \lambda$. If such a tree is still present in the forest resulting from deleting $\mathcal{S}$, the selected edge set connects all terminals v with $\lambda(v) = \lambda$.
- Conversely, if terminal v is *not* connected to all terminals w with $\lambda(v) = \lambda(w)$, it must be connected to *some* node from $\mathcal{S}$ that received its label, by a path of $\mathcal{O}(L\sqrt{n} \log n) \subset \tilde{\mathcal{O}}(\sqrt{n})$ hops.

Hence, if we decide that nodes in $\mathcal{S}$ simply hold on to received labels instead of pushing them back to some terminal, when the above routine terminates, all that is left is to connect nodes from $\mathcal{S}$ that share a label.

To this end, we use the spanner-based algorithm of [19]. Since $|\mathcal{S}| = \sqrt{n}$, setting $\kappa = \log n$ results in time complexity $\tilde{\mathcal{O}}(n^{1/2} + D)$, which is our target. However, this may cause the approximation ratio to deteriorate to $\mathcal{O}(\log^2 n)$: Since $\mathcal{S}$ is not the original set of terminals, but only connected to it by edges of total (expected) weight $\mathcal{O}(\log n)$ times the optimum, the weight of an optimal solution on the derived instance on terminal set $\mathcal{S}$ may grow by this factor.

Component Contraction. We resolve this issue as follows. We associate each unsatisfied terminal with a node from $\mathcal{S}$ to which it is already connected. Since unsatisfied terminals are within $\tilde{\mathcal{O}}(\sqrt{n})$ hops from $\mathcal{S}$ using only selected edges, this can be done in $\tilde{\mathcal{O}}(\sqrt{n})$ rounds.

Next, we contract the constructed clusters, associating with each resulting cluster the labels of all terminals it contains. Leveraging pipelining techniques similar to those used earlier, in $\mathcal{O}(|\mathcal{S}| + k + D)$ rounds we can assign to each cluster a single label so that the connectivity requirements are identical to those given by the terminal labels. The key

observation here is that whenever two terminals in different clusters are to be connected, it suffices to connect *any* pair of nodes from the clusters, because all nodes within each cluster are already connected. Finally, we run the spanner-based algorithm from [19] on this instance; conveniently, the technique trivially extends to the modified setting with nodes that are, in fact, contracted sets of nodes in G.

Because we constructed the new instance by means of contraction, an optimal solution of the original instance induces a solution of at most the same weight for the new instance. Therefore, the computed solution has weight within factor $\mathcal{O}(\log n)$ of the optimum, but this weight contributes *additively* to the weight of the complete solution. And as the new instance has at most $|\mathcal{S}| = \sqrt{n}$ terminals, the algorithm completes within $\tilde{\mathcal{O}}(\sqrt{n} + D)$ rounds.

THEOREM 5.2. *There is an algorithm that solves* DSF-IC *in* $\tilde{\mathcal{O}}(\sqrt{n} + k + D)$ *rounds w.h.p. within factor* $\mathcal{O}(\log n)$ *of the optimum in expectation.*

By combining this algorithm and the one from the previous section of running time $\tilde{\mathcal{O}}(s+k)$, we obtain an $\mathcal{O}(\log n)$-approximation whose running time is optimal up to a polylogarithmic factor. Applying the algorithm $\mathcal{O}(\log n)$ times and choosing the solution of minimum weight ensures that the approximation guarantee holds w.h.p.

COROLLARY 5.3. *There is an algorithm that solves* DSF-IC *in* $\tilde{\mathcal{O}}(\min\{s, \sqrt{n}\} + k + D)$ *rounds within factor* $\mathcal{O}(\log n)$ *of the optimum w.h.p.*

Acknowledgements

We thank Fabian Kuhn for valuable discussions.

6. REFERENCES

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner tree problem on networks. *SIAM Journal of Computing*, 24:440–456, 1995.

[2] S. Baswana and S. Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures and Algorithms*, 30(4):532–563, 2007.

[3] M. Brazil, R. Graham, D. Thomas, and M. Zachariasen. On the history of the Euclidean Steiner tree problem. *Archive for History of Exact Sciences*, pages 1–28, 2013.

[4] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità. An improved LP-based Approximation for Steiner Tree. In *Proc. 42nd ACM Symp. on Theory of Computing*, pages 583–592, 2010.

[5] P. Chalermsook and J. Fakcharoenphol. Simple Distributed Algorithms for Approximating Minimum Steiner Trees. In *Proc. 11th Conf. on Computing and Combinatorics*, volume 3595 of *LNCS*, pages 380–389, 2005.

[6] M. Chlebík and J. Chlebíková. The Steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science*, 406(3):207–214, 2008.

[7] R. Cole and U. Vishkin. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 206–219, 1986.

[8] R. Courant and H. Robbins. *What is Mathematics? An Elementary Approach to Ideas and Methods*. London. Oxford University Press, 1941.

[9] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. In *Proc. 43th ACM Symp. on Theory of Computing*, pages 363–372, 2011.

[10] M. Elkin. An Unconditional Lower Bound on the Time-Approximation Tradeoff for the Minimum Spanning Tree Problem. *SIAM Journal of Computing*, 36(2):463–501, 2006.

[11] J. Fakcharoenphol, S. Rao, and K. Talwar. A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics. *J. Comput. System Sci.*, 69(3):485–497, 2004.

[12] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. on Comp. Syst.*, 5(1):66–77, Jan. 1983.

[13] J. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal of Computing*, 27:302–316, 1998.

[14] M. Hauptmann and M. Karpinski. A Compendium on Steiner Tree Problems. http://theory.cs.uni-bonn.de/info5/steinerkompendium/netcompendium.html. Retreived January 2014.

[15] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

[16] M. Khan, F. Kuhn, D. Malkhi, G. Pandurangan, and K. Talwar. Efficient Distributed Approximation Algorithms via Probabilistic Tree Embeddings. *Distributed Computing*, 25:189–205, 2012.

[17] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[18] S. Kutten and D. Peleg. Fast Distributed Construction of Small k-Dominating Sets and Applications. *J. Algorithms*, 28(1):40–66, 1998.

[19] C. Lenzen and B. Patt-Shamir. Fast Routing Table Construction Using Small Messages: Extended Abstract. In *Proc. 45th Ann. ACM Symp. on Theory of Computing*, pages 381–390, 2013.

[20] C. Lenzen and B. Patt-Shamir. Improved Distributed Steiner Forest Construction. *CoRR*, abs/1405.2011, 2014.

[21] Z. Lotker, B. Patt-Shamir, and D. Peleg. Distributed MST for constant diameter graphs. *Distributed Computing*, 18(6):453–460, 2006.

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

[23] D. Peleg and V. Rubinovich. Near-tight Lower Bound on the Time Complexity of Distributed MST Construction. *SIAM J. Computing*, 30:1427–1442, 2000.

[24] H. Takahashi and A. Matsuyama. An Approximate Solution for the Steiner Problem in Graphs. *Mathematica Japonica*, 6:573–577, 1980.

[25] R. E. Tarjan. *Data Structures and network Algorithms*, chapter 6. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.