A Distributed Algorithm for Directed Minimum-Weight Spanning Tree

Orr Fischer
Computer Science Department, Tel-Aviv University, Israel
orrfischer@mail.tau.ac.il

Rotem Oshman
Computer Science Department, Tel-Aviv University, Israel
roshman@mail.tau.ac.il

Abstract

In the *directed minimum spanning tree* problem (DMST, also called *minimum weight arborescence*), the network is given a root node r, and needs to construct a minimum-weight directed spanning tree, rooted at r and oriented outwards. In this paper we present the first sub-quadratic DMST algorithms in the distributed CONGEST network model, where the messages exchanged between the network nodes are bounded in size. We consider three versions: a model where the communication links are bidirectional but can have different weights in the two directions; a model where communication is unidirectional; and the Congested Clique model, where all nodes can communicate directly with each other.

Our algorithm is based on a variant of Lovász' DMST algorithm for the PRAM model, and uses a distributed single-source shortest-path (SSSP) algorithm for directed graphs as a black box. In the bidirectional CONGEST model, our algorithm has roughly the same running time as the SSSP algorithm; using the state-of-the-art SSSP algorithm, we obtain a running time of $\widetilde{O}(\min(\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D))$ rounds for the bidirectional communication case.

For the unidirectional communication model we give an $\widetilde{O}(n)$ algorithm, and show that it is nearly optimal. And finally, for the Congested Clique, our algorithm again matches the best known SSSP algorithm: it runs in $\widetilde{O}(n^{1/3})$ rounds.

On the negative side, we adapt an observation of Chechik in the sequential setting to show that in all three models, the DMST problem is at least as hard as the (s,t)-shortest path problem. Thus, in terms of round complexity, distributed DMST lies between single-source shortest path and (s,t)-shortest path.

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Distributed Computing, Directed Minimum Spanning Tree, Minimum Arborescence, CONGEST

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.16

Related Version A full version of the paper is available at `https://www.cs.tau.ac.il/~roshman/papers/DISC19_FO.pdf`.

Funding *Orr Fischer*: Research supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11) and the Laura Schwarz-Kipp Institute of Computer Networks.
Rotem Oshman: Research supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

1 Introduction

Finding a lightweight spanning subgraph of a network is among the most fundamental problems in distributed computing. The classical example is the minimum-weight spanning tree (MST) problem, which has received extensive attention: its round complexity in the CONGEST model was tightly characterized in a series of papers (e.g [14, 26, 15, 8, 9, 22, 17, 23, 10]). Generalizations, such as minimum-weight k-vertex-connected and k-edge connected subgraph, have also been studied (e.g [10, 5, 31]). To date, almost all distributed algorithms for MST and related problems have been for *undirected* graphs, with *symmetric* edge weights. However, in many settings, the cost associated with an edge is not necessarily symmetric: for example, in a wireless network, the energy required to send a message to a specific node can depend on contention and noise in that node's vicinity, and in peer-to-peer cellular phone mesh networks, the price of communicating across a given link could be dictated by market forces. If we have a single node that needs to repeatedly broadcast to the entire network, or to collect information from the entire network, can we quickly find a cheap spanning tree – oriented downwards (or upwards) – allowing it to do so?

The *directed minimum-weight spanning tree* (DMST) problem asks exactly this question: we have a weighted graph G, where edge weights are not necessarily symmetric, and a fixed root node r. Our goal is to construct a minimum-weight directed spanning tree, rooted at r and oriented downwards (or upwards).

Although the DMST problem has been extensively studied in the sequential setting [32, 7, 2, 6, 28, 29, 13], to date there has not been a distributed solution for DMST that runs quickly and does not use a lot of communication. In fact, prior to our work, no non-trivial (i.e., sub-quadratic) algorithm for the CONGEST model was known. In this paper we give distributed DMST algorithms for three variants of the CONGEST model: (a) undirected communication networks with asymmetric edge weights; (b) directed communication networks; and (c) the Congested Clique model, where the communication network is the complete graph (a clique).

Undirected MST is known to require $\tilde{\Theta}(\sqrt{n}+D)$ rounds [30, 10]. Clearly, we cannot hope for DMST to require less, as MST is a special case of DMST. Furthermore, in some scenarios (e.g., sequential dynamic graph algorithms), DMST is believed to be significantly harder than MST. Surprisingly, we show that when the underlying communication network is undirected and has a small diameter, DMST is no harder than MST. In fact, we show that in undirected networks, DMST essentially "reduces" to directed single-source shortest path (SSSP), so that up to a logarithmic factor, its round complexity is bounded from above by the running time of the best SSSP algorithm that can handle asymmetric weights (currently [12]). On the other hand, we show that DMST is no easier than (s,t)-shortest path – this is already known in the sequential setting (see Section 6), and we show that it also holds in all three variants of the CONGEST model. Therefore, DMST's round complexity is sandwiched between SSSP and (s,t)-shortest path.

Background. The best sequential algorithm for DMST is Gabow et al.'s implementation of Edmonds' algorithm [7, 13]. It performs a series of *contractions*, where every vertex $v \neq r$ deducts the weight of its minimum-weight incoming edge from all its incoming edges, and then each zero-weight directed cycle is contracted into a single vertex. Eventually, we are left with a zero weight tree; the weight of the DMST is then given by the sum of all the weights deducted during the algorithm's run (See Section 4 for details). Actually *finding* the DMST is not immediate, and requires recursively undoing the contractions and carefully adding edges to the DMST at each step. (Counter-intuitively, the lightest incoming edge of any given node does not necessarily belong to a DMST, and in fact, even the lightest edge in the entire graph might not belong to it.)

The drawback of Edmonds' algorithm in a parallel setting is that it may require $n-1$ contractions to contract the entire graph, and the contractions are not easy to parallelize. In [24], Lovász gave a PRAM algorithm that "speeds up" this process, and contracts the entire graph in $O(\log n)$ parallel steps. In CONGEST, Lovász' algorithm cannot be implemented efficiently as-is, for several reasons – including the fact that it uses all-pairs shortest path (APSP) as a subroutine (APSP requires linear time in CONGEST [1]), and that certain steps of the algorithm would lead to too much congestion if we try to implement them in CONGEST. We modify Lovász' algorithm to obtain a variant that lends itself to an efficient distributed implementation, and then give implementations for the three variants of CONGEST. The implementations overcome several challenges that are not encountered in undirected MST, such as the fact that in each step we need to run SSSP inside many disjoint subgraphs, but each component can have a diameter that is much larger than the diameter of the network as a whole. If we called SSSP directly inside each component, our running time would depend on the largest diameter encountered during the run, which could be linear in the worst case. We show how to overcome this difficulty in Section 5.

Our results. We give one "meta-algorithm" for DMST, and then implement it in the three models we consider (undirected, directed, and Congested Clique). For the bidirectional CONGEST model and the Congested Clique, we show that given an efficient algorithm for single-source shortest paths (SSSP), we can find a DMST in roughly the same running time. Specifically, let $T(n, D)$ be the time required in CONGEST to compute SSSP in undirected graphs of size n and diamter D, with non-negative, asymmetric integer weights, and let $\mathcal{A}_{\text{SSSP}}$ be an SSSP algorithm with running time $T(n, D)$. We prove:

▶ **Theorem 1** (Informal). *There is a DMST algorithm for undirected* CONGEST *with asymmetric weights that runs in $\widetilde{O}(T(n, D))$ rounds. Moreover, the DMST algorithm is deterministic if $\mathcal{A}_{\text{SSSP}}$ is deterministic.*

We take extra care to ensure that our "reduction" from DMST to SSSP be *deterministic*, so that if in the future an efficient deterministic SSSP algorithm is discovered, we can use it to get a deterministic DMST algorithm.

Plugging in the randomized Las-Vegas SSSP algorithm of [12], we obtain the following algorithm for the undirected CONGEST model with asymmetric edge weights:

▶ **Theorem 2.** *In the undirected* CONGEST *model with asymmetric weights, there is a randomized DMST algorithm that always succeeds, and requires $\widetilde{O}(\min(\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D))$ rounds in expectation.*

For small diameter networks, $D = O(\text{polylog}(n))$, our algorithm is optimal up to polylogarithmic factors, and nearly matches the lower bound for *undirected* MST [30]. For larger diameter, we can also write the running time as $\widetilde{O}(n^{2/3} + D)$, a slightly weaker bound than the one stated in Theorem 2. Since our algorithm calls the SSSP algorithm as a black box, any improvement in SSSP will yield an improved DMST algorithm as well.

A similar result holds for the Congested Clique. At present, the best SSSP algorithm for that model runs in $\tilde{O}(n^{1/3})$ rounds [3], and so we obtain an $\tilde{O}(n^{1/3})$-round DMST algorithm for the Congested Clique. For the directed communication model, we give a deterministic algorithm with running time $\widetilde{O}(n)$, and we show that this is tight (up to a logarithmic factor). The algorithm and the lower bound assume that the weight of each edge (u, v) is known only to its destination v, and that G is strongly connected. These results are described in full version of the paper [11].

As Theorem 2 shows, in the undirected CONGEST model, the DMST problem is no harder than single-source shortest path. Is the converse true? For the sequential setting, this is conjectured to hold, and Chechick showed [4] that DMST is at least as hard as the (s, t)-shortest path problem. We give a reduction that allows the proof from [4] to work in the distributed setting, showing that DMST is no easier than (s, t)-shortest path in all three distributed models we consider.

For lack of space, we only give a high-level overview of the algorithm for the undirected case, and defer many technical details – as well as pseudo-code and proofs of correctness – to the full version of the paper [11]. The other two models (directed networks and the Congested Clique) are also relegated to the the full version of the paper. Finally, we focus here on computing the *weight* of the DMST, and defer the details of how to find the DMST edges (which is requires some details) to the last section of the paper.

We note that our algorithm naturally extends to approximating DMST using an approximate SSSP algorithm for directed graphs with non-zero weights. Using this extension, a c-approximation directed SSSP algorithm yields a $c^{\log n}$-approximation of the DMST (meaning we require an $(1+\frac{1}{\Omega(\log n)})$-approximate SSSP algorithm in order to get a constant or sub-constant DMST approximation using this method). The best known $(1+\epsilon)$-SSSP approximation algorithm for directed graphs in CONGEST has round complexity of $\tilde{O}((\sqrt{n}D^{1/4}+D)/\epsilon)$ [12], which yields an $(1+\frac{1}{\text{polylog}\, n})$-approximation of the DMST in $\tilde{O}(\sqrt{n}D^{1/4}+D)$ rounds. We defer the details to the full version of the paper.

2 Related Work

Distributed MST is one of the most fundamental problems in CONGEST, with a wide range of works, a very short subset of which include [14, 26, 15, 8, 9, 22, 17, 23, 10]. In particular, Ghaffari et al.[15] gave a simple MST algorithm using a framework called *low-congestion shortcuts*. This framework also serves as the basis for our DMST algorithm, as it allows to handle connected components that grow too large for their nodes to communicate with each other directly. Our algorithm also uses procedures from [19, 10] to deterministically decompose a directed tree into few components with relatively small diameter. Several lower bounds were shown by [27, 8, 30], proving that in the CONGEST model, finding the MST's weight takes $\widetilde{\Omega}(\sqrt{n}+D)$ rounds ,even for any approximation factor of up to $poly(n)$.

Minimum Directed MST (or Minimum weight Arborescence) had been extensively studied in the sequential model. The first algorithms for DMST in the sequential setting were independently found by [32, 7, 2]. A faster implementation was given by Tarjan [6], which included ideas from [28, 29]. The most efficient known implementation of Edmonds' algorithm in the sequential setting is due to [13], with running time $O(m+n\log n)$.

A parallel NC algorithm for DMST was given by Lovász [24]. Humblet [21] showed a distributed $O(n^2)$ round algorithm for DMST with message complexity $O(n^2)$. To our knowledge, ours is the first DMST algorithm for CONGEST that has better than the trivial round complexity of $O(n^2)$.

Our algorithm uses a directed single source shortest path algorithm as a black-box. Recently, two such algorithms were developed [16, 12]. The best known running time for for both directed and undirected graphs is $\widetilde{O}(\min(\sqrt{nD}, \sqrt{n}D^{1/4}+n^{3/5}+D))$ due to Nanongkai et al. [12]. In [12] an $(1+o(1))$-approximation in time $\widetilde{O}(\sqrt{n}D^{1/4}+D)$ for the directed case was shown. In the undirected case, [20] gave a deterministic $(1+o(1))$-approximation in time $\widetilde{O}(n^{1/2+o(1)}+D^{1+o(1)})$. In the Congested Clique, Censor-Hillel et al.[3] gave a $\widetilde{O}(n^{1/3})$ APSP algorithm for directed graphs based on algebraic methods.

3 Preliminaries

Let $G=(V,E,w_G)$ be a weighted directed graph, with a special *root* vertex $r\in V$. We assume throughout that r has a directed path to every node in G. We construct a spanning tree rooted at r and oriented downwards. (To obtain a tree oriented upwards towards

r, we can simply reverse all edge directions.) For convenience, if $(u,v) \notin E$, then we set $w_G(u,v) = \infty$. Also, given a vertex set $A \subseteq V$, we denote by $G(A)$ the subgraph induced on G by A. We sometimes abuse notation by writing $u \in H$ when u is a vertex of a subgraph H.

We assume w.l.o.g. that edge weights are integers in the range $[0, ..., \text{poly}(n)]$. This is not essential; negative weights and larger weights are easily handled, although if weights require more than $O(\log n)$ bits to represent, the SSSP algorithm will use more rounds.

For two nodes u, v, let $\text{dist}_G(u,v)$ be the weight of the shortest path from u to v according to the weight function w_G. Given a subgraph H of G and two nodes $u, v \in H$, we let $\text{dist}_H(u,v)$ denote the weight of the shortest path *using only vertices of* H from u to v. For a subgraph H, let $\text{In}(H) = \{(u,v) \in E | v \in H \wedge u \notin H\}$ be the set of edges entering H.

4 Overview of the Algorithm

In this section we give a high-level overview of our DMST algorithm, which is based on Edmonds' and Lovász' algorithms.

As it runs, the algorithm performs *contractions*, where a set of vertices is merged into one *super-vertex*. Here we describe the "meta-algorithm" that runs on the graph of super-vertices, and later we will show how this meta-algorithm is implemented on the actual network (where, of course, we cannot merge nodes).

The active edges. Throughout its run, the algorithm maintains a set of zero-weight directed edges, denoted H, with the property that every (super-)vertex except r has in-degree 1 in H.

To initialize H, each node v chooses a minimum-weight incoming edge (u,v), deducts its weight from all incoming edges, and adds (u,v) to H. (If there is more than one incoming edge with the minimum weight, then we choose arbitrarily.)

The weakly-connected component of H that contains the root is called the *root component*. The remaining weakly-connected components of H are called *active components*, and denoted $H_1, \ldots, H_k$. Since the in-degree in H is 1, each active component is a directed cycle, with trees rooted at some of the cycle's vertices and oriented outwards (see Fig. 2). We abuse notation by thinking of each H_i as both a set of edges and as a graph (the weakly-connected component). We let $C(H_i)$ denote the directed cycle that "lies at the heart" of the active component H_i.

The following property is helpful when trying to determine which vertices belong to a given active component: if we know some vertex v that lies on the cycle $C(H_i)$, then the vertices of H_i are exactly those vertices reachable from v along the directed edges of H.

Edmonds' contractions. Edmonds' algorithm makes a series of steps, where in each step,

(1) Each vertex v deducts the weight of its minimum-weight incoming edge from all its incoming edges, and remembers the weight it subtracted. (We must connect v to the DMST by *some* incoming edge, so we will pay at least the weight of its lightest incoming edge.)

(2) Each vertex adds one zero-weight incoming edge to H.

(3) Any newly-created zero-weight directed cycles in H are contracted into a single vertex. (This does not change the weight of the DMST.)

Eventually, we are left with only the root component, on which the H edges induce a directed spanning tree of weight zero. The weight of the DMST is then given by the total weight subtracted by all the nodes during the run. Then, we must "undo" the constructions and compute the edges of the DMST; we defer this part to the end of the paper, and focus for now on computing the weight of the DMST.

Each step of Edmonds' algorithm contracts at least two vertices, but unfortunately, it does not necessarily merge each active component with another active component: an active component might spend many steps contracting nested cycles of inner vertices, one after the other. While each step reduces the number of vertices by at least 1, we might require as many as n steps to contract the entire graph.

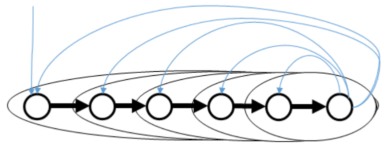

Figure 1 An active component in which Edmonds' algorithm contracts only two vertices at a time. In the worst case $\Omega(n)$ contractions occur before the active component is merged with another component.

Lovász' algorithm can be viewed, somewhat inaccurately, as a way to "jump ahead": roughly speaking, instead of spending a lot of time contracting nested cycles inside an active component, Lovász finds the first edge *coming in from outside the component* that would be added to H, and then performs in one fell swoop all the nested contractions leading up to that point. (This is not accurate, as we explain below; one step of Lovász cannot always be decomposed into steps of Edmonds.) After $O(\log n)$ "mega-steps", we are left with only the root component, and then we are done. See the full version of the paper [11] for a more detailed description of Lovász' algorithm.

The "mega-steps" of Lovász are difficult to implement in CONGEST: in each step, the algorithm computes all-pairs shortest-paths (APSP, which can be solved efficiently in PRAM), and it finds paths that may cut across many active components, leading to congestion. We give a less eager mechanism for speeding up Edmonds' algorithm, which is quite similar to Lovász but can be performed in parallel on all the active components in CONGEST; essentially, we show that the steps of Lovász' algorithm can be *confined inside the active components* without cutting across them, while preserving correctness and the fast running time.

Our modified meta-algorithm. Our meta-algorithm is obtained from Edmonds by asking: "what contractions would Edmonds' algorithm make inside an active component H_i before it adds to H an *incoming* edge of H_i, thereby merging it with another component?" We would like to jump ahead to that point.

Recall that Edmonds selects at each step the minimum-weight incoming edge of a node, adds it to H, and (eventually) contracts the resulting zero-weight cycle. It turns out that as it slowly consumes nodes inside H_i and eventually some node outside H_i, Edmonds implicitly finds the lightest path from a node outside H_i that *immediately enters* H_i and stays inside H_i until it arrives at some node of the cycle $C(H_i)$ (see Fig. 2); i.e., a path of the form $u, v_1, \ldots, v_k$ such that (a) $u \notin H_i$, (b) we have $v_j \in H_i \setminus C(H_i)$ for each $j = 1, \ldots, k$, and finally, (c) $v_k \in C(H_i)$.

Let β be the weight of this path. Edmonds does not progress *only* along the path; it contracts nodes inside H_i that can reach the cycle $C(H_i)$ with paths of increasing weight, until the weight reaches β. Our meta-algorithm finds the edge (u, v_1) and the weight β, and contracts all nodes $x \in H_i$ that have $\text{dist}_{H_i}(x, C(H_i)) \leq \beta$ (including v_1). (Lovász also computes β, but it does it differently: it uses all-pairs shortest-paths to find the shortest distance from any node outside H_i to the cycle $C(H_i)$, without insisting that the path have the form we described above. As a result, Lovász may find paths that start at a node u, wander outside H_i for a while (along zero-weight edges), enter H_i and leave it, and eventually enter H_i "for good" and go to the cycle $C(H_i)$. Lovász then makes a more aggressive contraction, merging all the nodes visited by such a path of weight at most β.)

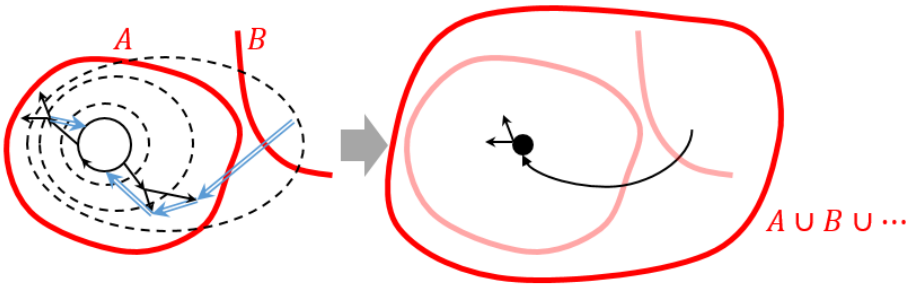

Figure 2 Our algorithm performs in one step three steps of Edmonds' algorithm: it contracts the zero-weight directed cycle of component A and makes two more contractions, finally adding an incoming edge of A to H. Component A then merges with B, and possibly with other components. Edges initially in H are shown as solid lines, edges that are added to H during Edmonds are shown as double lines.

We now give a more formal description. In each step of our meta-algorithm, we find in parallel for each active component H_i an incoming edge $(u, v) \in \text{In}(H_i)$, with $v \in H_i$ and $u \notin H_i$, that minimizes the internal distance to the cycle, $\beta(u, v) := w(u, v) + \text{dist}_{H_i}(v, C(H_i))$. (Recall that $\text{dist}_{H_i}(v, C(H_i))$ is the distance *inside* H_i from node v to any node of the cycle $C(H_i)$; only paths using edges of H_i can be used.)

For an active component H_i, let $\beta_i = \min_{(u,v)\in\text{In}(H_i)} \beta(u, v)$ be the "minimum entering distance" associated with H_i. This is the weight that would be subtracted by Edmonds' algorithm in all the contractions internal to H_i, plus the first step that connects H_i to another active component.

Our algorithm finds an edge (u, v) that has $\beta(u, v) = \beta_i$ (that is, an edge that minimizes $\beta(u, v)$). Then, we contract the zero-weight cycle $C(H_i)$, *together with all nodes inside* H_i that have distance at most $\beta(u, v)$ to the cycle $C(H_i)$. Formally, the set of nodes we contract into one super-vertex is given by

$$U_i := \{v \in H_i \mid \text{dist}_{H_i}(v, C(H_i)) \leq \beta_i\} . \tag{1}$$

We represent a super-vertex as the set of all original graph vertices that were merged into it; merging super-vertices means replacing them by their union.

For the new super-vertex S, we update the weights in the contracted graph (in which S is a vertex):

$$w'(x, y) = \begin{cases} \min_{z\in S} \beta(x, z) - \beta_i & \text{if } y = S, \\ \min_{z\in S} w(z, y) & \text{if } x = S, \\ w(x, y) & \text{otherwise.} \end{cases}$$

After the contraction, the incoming edge (u, S), which replaces (u, v) (the edge that had $\beta(u, v) = \beta_i$) and now has weight zero, is added to H. This causes H_i to merge with the active component to which u belongs (see Fig. 2).

Because every active component merges with another active component in each iteration, the number of components is reduced by at least half, and therefore after $O(\log n)$ iterations, the edge set H has only one weakly-connected component – the root component. At this point, the weight of the DMST is computed by summing all the β_i's that were subtracted during the entire run, and the algorithm terminates.

In the full version of the paper [11], we prove that unlike Lovász' algorithm, each step of our meta-algorithm can be decomposed into a series of Edmonds' contractions, and we give an example showing the difference between Lovász's algorithm and our variant.

5 Implementation in CONGEST

In this section we explain, on a high level, how we translate our meta-algorithm to the CONGEST model. We start by introducing the main ingredients that go into the implementation.

The meta-graph and the physical graph. We refer to the "real" nodes of the communication network as *physical vertices* or *physical nodes*. Super-vertices are simply sets of physical vertices, but each super-vertex has a unique identifier, which is the ID of some physical vertex in it. We often conflate a super-vertex with its ID. Let $\mathcal{S}$ be the set of all super-vertex IDs (as we said, these are simply IDs from V, but for clarity we use different notation).

During the run of our algorithm, each physical vertex v keeps track of $sId(v)$, the ID of the super-vertex that contains it in the meta-graph. Node v also knows which of its physical edges correspond to meta-edges in H: that is, for each physical edge $\{v, u\}$, node v knows whether or not $(sId(v), sId(u)) \in H$.

Given a physical network graph $G = (V, E, w)$ and a mapping $sId : V \to \mathcal{S}$ of physical nodes onto super-vertices, the meta-graph that corresponds to G and sId is a *multi-graph*, where two super-vertices $S_1, S_2 \in S$ are connected by all the edges that connect physical vertices $(u, v) \in E$ such that $u \in S_1, v \in S_2$. (Although our modified algorithm above is stated for graphs rather than multi-graphs, it is easy to see that its correctness translates immediately to multi-graphs as well.) For each super-vertex $S \in \mathcal{S}$, there is a single incoming meta-edge (T, S) in H (recall that all nodes have in-degree exactly 1 in H). The meta-edge (T, S) may correspond to many physical edges; the algorithm chooses one such edge, $(u, v) \in E$ such that $u \in T$ and $v \in S$, and defines $\text{entry}(S) = v$ to be "the physical entry-point of S".

Soft contractions. Since we cannot contract vertices of the communication network, we replace contractions with *soft contractions*, which have the same effect but change only the weight function and the super-vertex mapping.

▶ **Definition 3** (Soft contraction). *Fix a physical graph $G = (V, E, w_G)$, a mapping $sId : V \to \mathcal{S}$ of physical vertices to their super-vertices, a set $H \subseteq \mathcal{S}^2$ of zero-weight directed meta-edges, an active component H_i, and a set $A \subseteq \mathcal{S}$ of super-vertices to contract. Define $G_{\sim A} = (V, E, w_{G_{\sim A}})$ to be the physical graph with the same vertices and edges as G, but with the following weight function:*

$$w_{G_{\sim A}}(u, v) = \begin{cases} w_G(u, v) + \text{dist}_{G(A)}(v, C(H_i)) - \beta_i & \textit{if } u \in V \setminus A \textit{ and } v \in A, \\ 0 & \textit{if } u, v \in A \textit{ and } (u, v) \in E, \\ w_G(u, v) & \textit{otherwise.} \end{cases}$$

The soft contraction *operation updates the weights as above, and replaces the mapping sId with sId′, where $sId'(v) = sId(v)$ if $v \notin A$, and $sId'(v) = A.id$. The id of A is the id of some "leader" vertex $v \in A$.*

Intuitively, a soft contraction is the same as the meta-step we defined in Section 4, but instead of merging vertices, it simply zeroes out the weight of the edges between them. We prove that if we take $A = U_i$ (as defined in (1) above), then the soft contraction operation is equivalent to the meta-step we defined in Section 4, and decreases the weight of the DMST by β_i.

Small and large components. As usual in MST algorithms, after we perform some meta-steps of the algorithm, some super-vertices may become so large that we cannot afford for their physical nodes to communicate with each other directly. We resolve this in the usual way (see, e.g., [15, 10]): super-vertices are classified into "small" super-vertices, which comprise at most $\sqrt{n}$ physical nodes, and "large" super-vertices, comprising more than $\sqrt{n}$ nodes. The small super-vertices are small enough that we can compute on them directly (in parallel). As for large super-vertices, there are at most $\sqrt{n}$ of them, and the entire network helps them carry out their computation. For example, if we have large super-vertices $S_1, \ldots, S_k$, and we want each physical vertex of S_i to learn some value x_i, then we will propagate all values $x_1, \ldots, x_k$ throughout the entire network, and each physical vertex $v \in S_i$ will pick out the value x_i it needs to learn.

We remark that unlike undirected MST, in our case there is a distinction between a *super-vertex* and an *active component.* (In distributed implementations of Boruvka's undirected MST algorithm, there are super-vertices, but there is no notion of "active component".) In addition to computing on all the super-vertices in parallel, our algorithm also carries out steps on the *active components* in parallel, but an active component consists of many super-vertices, some small and some large. This presents some complications compared to the undirected case.

Centers. A key part of our algorithm is concerned with finding some super-vertex that lies on the cycle $C(H_i)$ of an active component. This cycle may consist of any number of super-vertices, themselves comprising many physical-vertices. To find a super-vertex on the cycle, we "chop up" the cycle into more manageable parts: we select a *center set*, a set of $\tilde{O}(\sqrt{n})$ super-vertices (always including the root super-vertex), with the property that for any H-path $S_1, \ldots, S_k$ of super-vertices, if the total number of physical vertices in $S_1, \ldots, S_k$ is at least $\sqrt{n}$, then at least one super-vertex S_i is a center.

Centers are often used in shortest-path computations (e.g., [12, 16, 25] in the CONGEST model, and many other examples in dynamic algorithms and distance oracles), but here we use them in a non-standard way: we construct a *center graph* representing the reachability relation between centers, and use this graph to find the cycle $C(H_i)$ and determine which super-vertices are reachable from it.

Running SSSP on many disjoint components in parallel. During our algorithm we encounter the following scenario: we have a collection of vertex-disjoint connected subgraphs $G_1, \ldots, G_k \subseteq G$, with one marked node $v_i \in G_i$ in each component, and also some external node $r \notin \bigcup_i G_i$. The diameter of the entire graph G is D, but the diameter of each G_i can be arbitrarily large. We wish to compute, *in parallel* for all i, the distances $\mathrm{dist}_{V_i}(v_i, u)$ (that is, the distance from v_i to each node inside its component V_i, using only nodes of V_i).[1]

[1] The keen-eyed reader might notice that the directions here are reversed – in Section 4 we wanted distances *to* a node of $C(H_i)$, and now we ask for distances *from* some node to all others in the component. We handle this by reversing all edge directions.

Moreover, we want to "pay" only in terms of the diameter D of the entire network, not the diameters of the individual components. This rules out running a separate SSSP instance inside each component using only its internal edges.

Our solution is to simulate *one* execution of SSSP on a "virtual network" G', defined as follows. The vertices of G' are the vertices of G, but we also add, for each $v \in V(G)$, a "shadow vertex" v'. The edges of G' are

(a) all edges that are internal to some subgraph G_i, with their original weights; (b) the "shadow copies" (u', v') of all edges in E, with weight zero; (c) for each marked node v_i, we add a zero-weight edge (v_i', v_i) from v_i's shadow to v_i, and also an edge (v_i, v_i') in the opposite direction, with "infinite" (or sufficiently large) weight.

The network G' can be simulated efficiently by the nodes of G, by having each node simulate itself and its shadow. Note that the edges we added allow for such a simulation; for example, two shadow nodes only need to communicate in G' if their corresponding "real nodes" can communicate in G. Also, $\mathrm{diam}(G') \leq 2\,\mathrm{diam}(G) + 1$.

Now, to simultaneously compute all the distances $\mathrm{dist}_{V_i}(v_i, u)$, we simulate a call to SSSP from node r', the shadow of r, in G'. A lightest path from r' to a node $u \in V_i$ traverses the shadow network from r' to v_i' at zero cost, then moves to v_i with no cost, and then traverses from v_i to u inside the "real" copy of G_i. Thus, the distance from r' to $u \in V_i$ in G' is exactly $\mathrm{dist}_{V_i}(v_i, u)$. See full paper for details.

5.1 The Algorithm

We now give a more detailed description of our algorithm (while still omitting many technical details). The algorithm runs in $O(\log n)$ iterations. At the beginning of each iteration, each node $v \in V$ knows an identifier $sId(v)$ for its super-vertex (initially, $s(v) = v$), it knows which of its edges correspond to meta-edges in H, and it knows whether or not it is part of the root component.

Nodes do not necessarily know which active component they belong to at any given moment; the first part of each iteration of our algorithm is concerned with finding the current active components, after some of them were merged at the end of the previous iteration. Nevertheless, it is convenient to think of the algorithm as "operating in parallel" on all the active components.

Each iteration proceeds as follows, in parallel for each active component H_i:

(1) We find some super-vertex $c(H_i) \in C(H_i)$ that lies on the cycle of H_i, and disseminate the ID of $c(H_i)$ to all physical nodes in H_i. In particular, we must determine which super-vertices belong to H_i. This is described in Section 5.2.

(2) We compute shortest paths from all super-vertices of H_i to $C(H_i)$: this is done by a single call to SSSP, as described above, using $c(H_i)$ as the marked node in component H_i. We use reverse edge weights, so that instead of computing shortest paths *from* $c(H_i)$ we compute shortest paths *to* $c(H_i)$. Note that since $C(H_i)$ is a cycle of zero-weight edges, the distance to $c(H_i)$ is also the distance to all nodes of $C(H_i)$.

(3) We find an incoming edge $e_i = (u, v) \in \mathrm{In}(H_i)$ that minimizes the "entering distance", $\beta(u, v) = w(u, v) + \mathrm{dist}_{H_i}(v, C(H_i))$, and disseminate e_i and $\beta_i = \beta(e_i)$ to all nodes of H_i. This is done using the small component/large component methodology, but some care is needed (as done in [15, 18]. see procedure `LearnMin` in the full version for details [11]).

(4) Finally, having computed β_i and $e_i = (u, v) \in \mathrm{In}(H_i)$, we soft-contract H_i with threshold β_i, "virtually merging" all super-vertices with distance at most β_i to $C(H_i)$ into one super-vertex. The ID of the new super-vertex is set to $c(H_i)$. We add edge e_i to H, which has the implicit effect of merging H_i with another active component.

After $O(\log n)$ iterations, no active components remain, and we have only the root component. We now compute a spanning tree of the network graph, and use it to sum the values of β_i subtracted throughout the algorithm. The root of the DMST returns this value as the weight of the DMST.

5.2 Finding A Cycle Super-Vertex and Identifying the Active Component

In this section we show how to find, for each active component H_i, some super-vertex $c(H_i)$ on the cycle $C(H_i)$. When we begin this part of the algorithm, the physical nodes know which of their edges are in H, but they do not know which active component (i.e., which weakly-connected components of H) they belong to. Part of our goal is to identify the boundaries of the active components, in preparation for finding a minimum-distance incoming edge of each active component; this is accomplished by disseminating $c(H_i)$ to all nodes that can be reached from the cycle $C(H_i)$ along paths of H-edges. Thus, $c(H_i)$ serves as an *active component ID*, which all physical nodes of H_i agree on.

As we said above, in order to identify long cycles and paths, we cut them into shorter pieces by choosing a set of *centers*. Formally, we need the following property:

▶ **Definition 4.** *A set of super-vertices $\mathcal{T} \subseteq \mathcal{S}$, which includes the root super-vertex, is said to be a* good center set *if $|\mathcal{T}| \leq 4\sqrt{n}$, and for any H-path $S_1, \ldots, S_k$, if $|\bigcup_{i=1}^{k} S_i| \geq \sqrt{n}$ (that is, if $S_1, \ldots, S_k$ together contain at least $\sqrt{n}$ physical vertices), then $\mathcal{T}$ includes some super-vertex S_i.*

A good center set can be constructed deterministically in a very similar manner to either the star-decomposition of [19], or using the fragment joining of [10]. Details regarding this can be found in the full version of the paper [11].

In the sequel we assume that we have such a set, *Centers*.

Recall that in H, every super-vertex has in-degree exactly 1. For a super-vertex S (not necessarily a center), we define $\text{pred}(S)$ to be the first center we reach by starting from S and traversing backwards along reverse H edges. (Note that a super-vertex can be its own predecessor, if it is a center and is part of a directed cycle in H that includes no other centers.)

The *center graph* is the graph induced by pred:

▶ **Definition 5** (The center graph, H^*). *The* center graph *induced by H and Centers, denoted H^*, is given by $H^* = (\mathit{Centers}, \{(\text{pred}(c), c) \mid c \in \mathit{Centers}\})$.*

For an active component H_i, let H_i^* be the subgraph of H^* induced by the centers $\mathit{Centers} \cap H_i$ selected from H_i. Note the following properties: (1) Like H, the center graph H^* also has in-degree 1, except for the root (always selected as a center), which has no incoming edges; (2) If H_i includes a center, then H_i^* is a weakly-connected component of H^*; (3) Whenever $C(H_i)$ includes at least one center, H_i^* contains a non-empty cycle $C(H_i^*)$ (possibly one center with a self-loop), whose vertices are the centers from $C(H_i)$.

Finding $c(H_i)$. After setting up the centers and the center graph, to find some super-vertex from the cycle $C(H_i)$, we divide into three cases, depending on whether the active component H_i and its cycle $C(H_i)$ include more than $\sqrt{n}$ physical nodes or not.

I. H_i is a "small component", including at most $\sqrt{n}$ physical nodes: then in particular, the physical size of the cycle $C(H_i)$ does not exceed $\sqrt{n}$. We can find $C(H_i)$ by having each super-vertex start a forward-BFS along the edges of H for $O(\sqrt{n})$ rounds, and

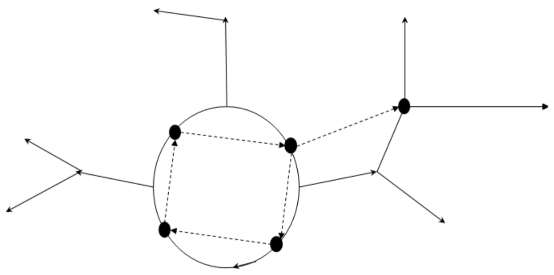

Figure 3 The center graph, overlayed on the super-vertex graph. Bold vertices represent centers, and dashed arrows represent the edges of the center graph.

always propagating the ID of the smallest super-vertex heard so far; after $O(\sqrt{n})$ rounds, some super-vertex receives back its own ID, and this super-vertex then becomes $c(H_i)$. We inform all nodes of H_i by propagating the ID of $c(H_i)$ for $O(\sqrt{n})$ rounds. This is handled by procedure `FindSmallCycles` (see details in full version [11]).

II. $C(H_i)$ is "small" (at most $\sqrt{n}$ physical nodes), but H_i is "large" (more than $\sqrt{n}$ nodes): in this case, procedure `FindSmallCycles` still selects some super-vertex $c(H_i) \in C(H_i)$ just as above. However, we cannot afford to disseminate the ID of $c(H_i)$ throughout H_i by broadcasting it, because H_i is too large. Instead, we add $c(H_i)$ to the center set *Centers*, and handle its dissemination below.

III. $C(H_i)$ is "large": then $C(H_i)$ includes at least one center, and we can identify $C(H_i)$ by examining the center graph H^* and looking for the corresponding cycle there.

In cases (II) and (III), after `FindSmallCycles` is called, $C(H_i)$ includes at least one center: either it was there before, or if the cycle was too small, we added some center in `FindSmallCycles`. Therefore, the component H_i^* that corresponds to H_i in the center graph contains a cycle $C(H_i^*)$.

After calling `FindSmallCycles`, every super-vertex S learns the identity of $\text{pred}(S)$. Because every H-path of physical size at least $\sqrt{n}$ includes a center, for each super-vertex S (not necessarily a center), the physical distance from the entry vertex of $\text{pred}(S)$ to some physical vertex in S is at most $\sqrt{n}$. Thus, the super-vertex $\text{pred}(S)$ can "tell S" that it is its predecessor by doing a forward BFS for $\sqrt{n}$ rounds (we omit the details here).

The center graph has $O(\sqrt{n})$ edges: its in-degree is 1, and even after adding some centers in step II, we still have $O(\sqrt{n})$ centers, because a center is only added for active components of physical size $> \sqrt{n}$. Thus, we can afford to disseminate all edges of H^* throughout the network, in $O(\sqrt{n} + D)$ rounds.

Finally, each physical node v locally examines the graph H^*, and constructs the weakly connected components of H^*. It associates itself with the correct component H_i^* by choosing the component of H^* that contains the center $\text{pred}(sId(v))$, that is, the predecessor of its own super-vertex. If H_i^* includes the root, then v sets the root's ID as its active component ID. Otherwise, node v finds $C(H_i^*)$, selects the center with the smallest id $c \in C(H_i^*)$, and sets $cId(v) = c$.

6 DMST vs. (s, t)-Shortest Path

We have shown that the DMST problem is no harder than single-source shortest path. In this section we adapt a reduction of Chechick [4] from the sequential setting to CONGEST, showing that distributed DMST is at least as hard as (s, t)-shortest path, where we are given two vertices s, t and must find the shortest directed path from s to t. The reduction holds for all three models we consider in this paper (assuming we work with strongly-connected graphs): it simply modifies the graph on which we want to solve (s, t)-SP, so that any DMST on the graph will reveal the shortest path from s to t. We take care that the modified graph can be *simulated* by the original graph without much additional communication.

Given a graph $G = (V, E)$, we define a graph G' as follows (see Fig. 4): G' contains all vertices and edges of G, and in addition, for each vertex $v \in V$, we add a "shadow vertex" v', with a zero-weight edge (v', v). For each original edge (u, v) we add a "shadow edge" (u', v'), again with weight zero. Finally, we add the zero-weight edge (t, t') (where t is the target node).

Observe that all the edges we added to G' are either shadow edges or edges incoming into vertices of G, except for the edge (t, t'), which is outgoing from t. Therefore, in G', we did not create any path from s to t that was not already in G.

▶ **Lemma 6.** *The weight of the DMST of G' rooted at s is the weight of the (s, t)-shortest path in G.*

Proof. Let W' be the weight of the DMST of G' rooted at s, and let d be the weight of the shortest path from s to t in G. It is easy to see that $W' \geq d$, because the DMST must contain *some* path from s to t, and in G' we did not create any path from s to t that was not already in G.

To show that $W' \leq d$, consider the following DMST: take a shortest path π from s to t in G, and add all its edges to the DMST. In addition, take edge (t, t'), and some arbitrary directed spanning tree of the shadow vertices, comprising only shadow edges and oriented outwards from the root t'. (Such a spanning tree exists, because we can take a directed spanning tree of G rooted at t and "copy it" onto the shadow edges.) Finally, for each $v \in V$ that is not on π, add the edge (v', v). The resulting tree is spanning and oriented outwards from s, and its weight is exactly d, because other than the edges of π, it uses only zero-weight edges. ◀

▶ **Theorem 7.** *The asymptotic round complexity of DMST in* CONGEST *is at least that of (s, t)-shortest path.*

Proof. Given a DMST algorithm $\mathcal{A}$ and a graph G, we can solve (s, t)-shortest path on G by constructing G' and simulating the execution of $\mathcal{A}$ on G'. Each vertex v of G simulates itself and its shadow vertex v'. To simulate one round of $\mathcal{A}$ on G', each vertex sends to its neighbors the messages that it would send under $\mathcal{A}$ on its own edges, and also the messages its shadow vertex would send on *its* edges under $\mathcal{A}$. This increases the communication by only a constant factor. ◀

7 Finding the Directed Minimum-Weight Spanning Tree

In this section we describe how to find the edges of the DMST, after contracting the entire graph into one component. This is an adaptation of the unpacking procedore from Lovasz' DMST algorithm [24], implemented in CONGEST. Again, some technical details are omitted here.

Recall that when we performed contractions, we looked for an edge that minimizes the *entering distance* into H_i,

$$\beta(u, v) := w(u, v) + \text{dist}_{H_i}(v, C(H_i)),$$

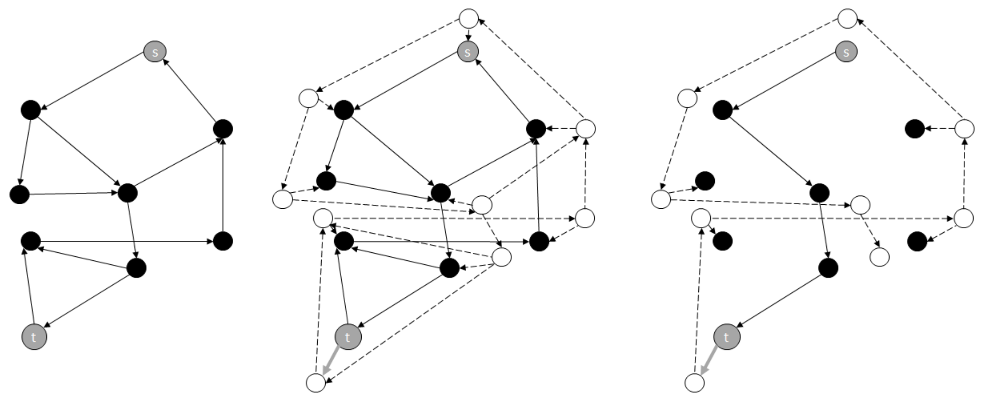

Figure 4 Local reduction of (s, t)-shortest path to DMST. The shadow nodes are shown in white. The rightmost figure shows the DMST.

and we denoted this minimum distance by

$$\beta_i = \min_{(u,v)\in \mathrm{In}(H_i)} \beta(u, v).$$

For an active component H_i, let $G_{\triangleright i}$ denote the contracted graph in which, starting from G, we contracted the cycle $C(H_i)$, together with all vertices inside H_i that have distance up to β_i from $C(H_i)$ (or rather, from the active component ID, $c(H_i)$), into one super-vertex. We now describe how to "undo" the contraction, so that we can unpack $G_{\triangleright i}$ back into G and add the correct edges to the DMST.

Unpacking a super-vertex. Consider a graph R with a set H of active edges, and let T' be a DMST of the contracted graph $R' = R_{\triangleright i}$. Let w, w' be the weight functions of R, R' respectively. Let $s = \bigcup U_i(\beta_i)$ be the new super-vertex in R' formed by merging together all vertices with distance at most β_i from $C(H_i)$ in R.

We define an *unpacking operation* that constructs from T' a new tree, denoted $T'_{i\triangleleft}$, for the original graph R, as follows:

- The new tree agrees with T' on all edges that are not adjacent to s: for any edge $e = (s_1, s_2)$ where $s_1, s_2 \neq s$ we have $(s_1, s_2) \in T'_{i\triangleleft}$ if $(s_1, s_2) \in T'$.
- Let (u, s) be the edge in T' that is incoming into s (there must be such an edge, since T' is spanning). Let $v^* \in s$ be the vertex that minimizes $\beta(u, v)$ among all incoming edges into s. We add to T the edge (u, v^*) and the lightest-weight path π from v to $C(H_i)$ in $R(H_i)$.
- If T' contains an outgoing edge (s, x) from s, each such edge is again replaced by an edge (y, x), where $y \in s$, that has $w(y, x) = \min_{y' \in s} w(y', x)$.
- Finally, we add to T the edges $H_i \setminus \mathrm{dest}(\pi)$, where $\mathrm{dest}(\pi)$ is the set of edges in R whose destinations are nodes on π.

▶ **Lemma 8** (Variant of [24], Claim 2). *If T' is a DMST of $R' = R_{\triangleright i}$, then $T'_{i\triangleleft}$ is a DMST of R.*

Unpacking the DMST. Now we describe how we "unpack" the entire DMST, starting from the final state of the algorithm where the graph has been contracted until only the root active component remains. The algorithm we describe here is run by the physical vertices of G, and it runs in $\log(n)$ iterations, indexed downwards, $\log n, \dots, 1$, where the j-th iteration unpacks the super-vertices created at step j. Let $\left\{H_1^t, \dots, H_{k_i}^t\right\}$ be the set of active components in iteration i.

We may assume w.l.o.g. that in an SSSP algorithm for directed graph with non-negative integer weights, each node also outputs a parent in a SSSP tree (e.g [12]). We require the nodes to store these edges; specifically, each node needs to remember, for each contraction, its edges in the reverse shortest-paths tree from $c(H^j)$ that was computed during the contraction.

Each super-vertex created during the current iteration is unpacked in parallel, and the edges taken into the DMST are the edges described above. When choosing which of their edges to add, the only computation nodes cannot perform locally is to find the shorest-path edges from v^* into $c(H(v^*))$. We handle this using centers, just as we did in Section 5.2: if the path is short, we can find it by doing a short BFS; and if the path is long, it will contain at least one vertex, and we can use the center graph to have the vertices of the path learn that they are on the path and add edges accordingly.

References

1 Aaron Bernstein and Danupon Nanongkai. Distributed Exact Weighted All-Pairs Shortest Paths in Near-Linear Time. In *Proceedings of the 51th Annual ACM SIGACT Symposium on Theory of Computing*, STOC '19, 2019.

2 F. Bock. An algorithm to construct a minimum directed spanning tree in a directed network. In *Developments in Operations Research*, pages 29–44, 1971.

3 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 143–152, 2015.

4 S. Chechik. Private Communication.

5 Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed Edge Connectivity in Sublinear Time. In *Proceedings of the 51th Annual ACM SIGACT Symposium on Theory of Computing*, STOC '19, 2019.

6 Tarjan R. E. Finding optimum branchings. *Networks*, 7(1):25–35, 1965.

7 Jack Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B(4):233–240, 1967.

8 M. Elkin. An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.

9 Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.*, 72(8):1282–1308, 2006.

10 Michael Elkin. A Simple Deterministic Distributed MST Algorithm, with Near-Optimal Time and Message Complexities. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 157–163, 2017.

11 Orr Fischer and Rotem Oshman. A Distributed Algorithm for Directed Minimum-Weight Spanning Tree. `https://www.cs.tau.ac.il/~roshman/papers/DISC19_FO.pdf`.

12 Sebastian Forster and Danupon Nanongkai. A Faster Distributed Single-Source Shortest Paths Algorithm. In *59th IEEE Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 686–697, 2018.

13 H N Gabow, Z Galil, T Spencer, and R E Tarjan. Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. *Combinatorica*, 6(2):109–122, January 1986.

14 J. A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 659–668, 1993.

15 Mohsen Ghaffari and Bernhard Haeupler. Distributed Algorithms for Planar Networks II: Low-Congestion Shortcuts, MST, and Min-Cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 202–219, 2016.

16 Mohsen Ghaffari and Jason Li. Improved Distributed Algorithms for Exact Shortest Paths. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 431–444, 2018.

17 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 19–28, 2016.

18 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-Congestion Shortcuts without Embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC) 2016*, pages 451–460, 2016.

19 Bernhard Haeupler and Jason Li. Faster Distributed Shortest Path Approximations via Shortcuts. In *32nd International Symposium on Distributed Computing, (DISC) 2018*, pages 33:1–33:14, 2018.

20 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A Deterministic Almost-tight Distributed Algorithm for Approximating Single-source Shortest Paths. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 489–498, 2016.

21 P. Humblet. A Distributed Algorithm for Minimum Weight Directed Spanning Trees. *IEEE Transactions on Communications*, 31(6):756–762, 1983.

22 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2620–2632, 2018.

23 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-Weight Spanning Tree Construction in $O(\log\log n)$ Communication Rounds. *SIAM J. Comput.*, 35(1):120–131, 2005.

24 L. Lovasz. Computing ears and branchings in parallel. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)(FOCS)*, volume 00, pages 464–467, 1985.

25 Danupon Nanongkai. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 565–573, 2014.

26 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A Time- and Message-optimal Distributed Algorithm for Minimum Spanning Trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 743–756, 2017.

27 D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed MST construction. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 253–261, 1999.

28 F. Maffioli P.M. Camerini, L. Fratta. A note on finding optimum branchings. *Networks*, 7:309—-312, 1979.

29 F. Maffioli P.M. Camerini, L. Fratta. The k best spanning arborescences of a network. *Networks*, 10(2):91—-109, 1980.

30 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

31 Ramakrishna Thurimella. Sub-linear Distributed Algorithms for Sparse Certificates and Biconnected Components. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, 1995.

32 T.H. Liu. Y.J. Chu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14(2):1396–1400, 1965.