Broadcast and Minimum Spanning Tree with $o(m)$ Messages in the Asynchronous CONGEST Model

Ali Mashreghi[1]
Department of Computer Science, University of Victoria, BC, Canada
ali.mashreghi87@gmail.com

Valerie King[2]
Department of Computer Science, University of Victoria, BC, Canada
val@uvic.ca

Abstract

We provide the first asynchronous distributed algorithms to compute broadcast and minimum spanning tree with $o(m)$ bits of communication, in a sufficiently dense graph with n nodes and m edges. For decades, it was believed that $\Omega(m)$ bits of communication are required for any algorithm that constructs a broadcast tree. In 2015, King, Kutten and Thorup showed that in the KT1 model where nodes have initial knowledge of their neighbors' identities it is possible to construct MST in $\tilde{O}(n)$ messages in the synchronous CONGEST model. In the CONGEST model messages are of size $O(\log n)$. However, no algorithm with $o(m)$ messages were known for the asynchronous case. Here, we provide an algorithm that uses $O(n^{3/2} \log^{3/2} n)$ messages to find MST in the asynchronous CONGEST model. Our algorithm is randomized Monte Carlo and outputs MST with high probability. We will provide an algorithm for computing a spanning tree with $O(n^{3/2} \log^{3/2} n)$ messages. Given a spanning tree, we can compute MST with $\tilde{O}(n)$ messages.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms, Mathematics of computing → Graph algorithms

Keywords and phrases Distributed Computing, Minimum Spanning Tree, Broadcast Tree

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.37

Related Version A full version of the paper is available at [22], `https://arxiv.org/abs/1806.04328`.

[1] Funded with an NSERC grant.
[2] Funded with an NSERC grant.

1 Introduction

We consider a distributed network as an undirected graph with n nodes and m edges, and the problem of finding a spanning tree and a minimum spanning tree (MST) with efficient communication. That is, we require that every node in the graph learns exactly the subset of its incident edges which are in the spanning tree or MST, resp. A spanning tree enables a message to be broadcast from one node to all other nodes with only $n-1$ edge traversals. In a sensor or ad hoc network where the weight of a link between nodes reflects the amount of energy required to transmit a message along the link [19], the minimum spanning tree (MST) provides an energy efficient means of broadcasting. The problem of finding a spanning tree in a network has been studied for more than three decades, since it is the building block of many other fundamental problems such as *counting*, *leader election*, and *deadlock resolution* [3].

A spanning tree can be constructed by a simple breadth-first search from a single node using m bits of communication. The tightness of this communication bound was a "folk theorem", according to Awerbuch, Goldreich, Peleg and Vainish [4]. Their 1990 paper defined the KT1 model where nodes have unique IDs and know only their neighbors. It showed, for a limited class of algorithms, a lower bound of $\Omega(m)$ messages in a synchronous KT1 network. In 2015, Kutten et al. [19] proved a lower bound for general randomized algorithms with $O(\log n)$ bit messages, in the KT0 model, where nodes do not know their neighbors. In 2015, King, Kutten, and Thorup gave the first distributed algorithm ("KKT") with $o(m)$ communication to build a broadcast tree and MST in the KT1 model. They devised Monte Carlo algorithms in the synchronous KT1 model with $\tilde{O}(n)$ communication [18]. This paper and a followup paper [21] left open the problem of whether a $o(m)$ bit communication algorithm in the asynchronous model was possible, for either the spanning tree or MST problem, when nodes know their neighbors' IDs.

In an asynchronous network, there is no global clock. All processors may wake up at the start and send messages, but further actions by a node are event-driven, i.e., in response to messages received. The pioneer work of Gallager, Humblet, and Spira [14] ("GHS") presented an asynchronous protocol for finding the MST in the CONGEST model, where messages are of size $O(\log n)$. GHS requires $O(m + n\log n)$ messages and $O(n \log n)$ time if all nodes are awakened simultaneously. Afterwards, researchers worked on improving the time complexity of MST algorithms in the CONGEST model but the message complexity remained $\Omega(m)$. In this paper, we provide the first algorithm in the KT1 model which uses $o(m)$ bits of communication for finding a spanning tree in an asynchronous network, specifically we show the following:

▶ **Theorem 1.** *Given any network of n nodes where all nodes awake at the start, a spanning tree and a minimum spanning tree can be built with $O(n^{3/2}\log^{3/2} n)$ messages in the asynchronous KT1 CONGEST model, with high probability.*

1.1 Techniques

Many distributed algorithms to find an MST use the Boruvka method: Starting from the set of isolated nodes, a forest of edge disjoint rooted trees which are subtrees of the MST are maintained. The algorithms runs in phases: In a phase, in parallel, each tree A finds a minimum weight *outgoing edge*, that is, one with exactly one endpoint in A and its other endpoint in some other tree B. Then the outgoing edge is inserted to create the "merged" tree containing the nodes of A and B. In what seems an inherently synchronous process, every tree (or a constant fraction of the trees) participates in some merge, the number of trees is reduced by a constant factor per phase, and $O(\log n)$ phases suffice to form a single tree. [14, 3, 18, 21].

The KKT paper introduced procedures *FindAny* and *FindMin* which can find any or the minimum outgoing edge leaving the tree, respectively. These require $O(|T|)$ messages and $\tilde{O}(|T|)$, resp., where $|T|$ is the number of nodes in the tree T or a total of $\tilde{O}(n)$ per phase. As this is done synchronously in KKT, only $O(\log n)$ phases are needed, for a total number of only $O(n \log n)$ messages to build a spanning tree.

While *FindAny* and *FindMin* are asynchronous procedures, the Boruvka approach of [18] does not seem to work in an asynchronous model with $o(m)$ messages, as it does not seem possible to prevent only one tree from growing, one node at a time, while the other nodes are delayed, for a cost of $O(n^2)$ messages. The asynchronous GHS also uses $O(\log n)$ phases to merge trees in parallel, but it is able to synchronize the growth of the trees by assigning a *rank* to each tree. A tree which finds a minimum outgoing edge waits to merge until the tree it is merging with is of equal or higher rank. The GHS algorithm subtly avoids traversing the whole tree until a minimum weight outgoing edge to an appropriately ranked tree is found. This method seems to require communication over all edges in the worst case.

Asynchrony precludes approaches that can be used in the synchronous model. For example, in the synchronous model, if nodes of low degree send messages to all their neighbors, in one round all nodes learn which of their neighbors do not have low degree, and therefore they can construct the subgraph of higher degree nodes. In the asynchronous model, a node, not hearing from its neighbor, does not know when to conclude that its neighbor is of higher degree.

The technique for building a spanning tree in our paper is very different from the technique in [18] or [14]. We grow one tree T rooted at one preselected *leader* in phases. (If there is no preselected leader, then this may be done from a small number of randomly self-selected nodes.) Initially, each node selects itself with probability $1/\sqrt{n \log n}$ as a *star node.* (We use $\log n$ to denote $\log_2 n$.) This technique is inspired from [10], and provides a useful property that every node whose degree is at least $\sqrt{n} \log^{3/2} n$ is adjacent to a star node with high probability. Initially, star nodes (and low-degree nodes) send out messages to all of their neighbors. Each high-degree node which joins T waits until it hears from a star node and then invites it to join T. In addition, when low-degree and star nodes join T, they invite their neighbors to link to T via their incident edges. Therefore, with high probability, the following invariant for T is maintained as T grows:

Invariant: T includes all neighbors of any star or low-degree node in T, as well. Each high-degree node in T is adjacent to a star node in T.

The challenge is for high-degree nodes in T to find neighbors outside T. If in each phase, an outgoing edge from a high-degree node in T to a high-degree node x (not in T) is found and x is invited to join T, then x's adjacent star node (which must lie outside T by the Invariant) is also found and invited to join. As the number of star nodes is $O(\sqrt{n}/\log^{1/2} n)$, this number also bounds the number of such phases. The difficulty is that there is no obvious way to find an outgoing edge to a high degree node because, as mentioned above, in an asynchronous network, a high degree node has no apparent way to determine if its neighbor has high degree without receiving a message from its neighbor.

Instead, we relax our requirement for a phase. With each phase either **(A)** A high-degree node (and star node) is added to T or **(B)** T is expanded so that the number of outgoing edges to low-degree nodes is reduced by a constant factor. As there are no more than $O(\sqrt{n}/\log^{1/2} n)$ phases of type **A** and no more than $O(\log n)$ phases of type **B** between each type **A** phase, there are a total of $O(\sqrt{n} \log^{1/2} n)$ phases before all nodes are in T. The key idea for implementing a phase of type **B** is that the tree T waits until its nodes have heard enough messages passed by low-degree nodes over outgoing edges before initiating an expansion. The efficient implementation of a phase, which uses only $O(n \log n)$ messages, requires a number of tools which are described in the preliminaries section.

Once a spanning tree is built, we use it as a communication network while finding the MST. This enables us to "synchronize" a modified GHS which uses $FindMin$ for finding minimum outgoing edges, using a total of $\tilde{O}(n)$ messages.

Note: If we do not assume the existence of a pre-selected leader, or the graph is not connected, then a variant of the algorithm described in the arxiv version [22] is needed.

1.2 Related work

The Awerbuch, Goldreich, Peleg and Vainish [4] lower bound on the number of messages holds only for (randomized) algorithms where messages may contain a constant number of IDs, and IDs are processed by comparison only and for general deterministic algorithms, where ID's are drawn from a very large size universe.

Time to build an MST in the CONGEST model has been explored in several papers. Algorithms include, in the asynchronous KT0 model, [14, 3, 13, 26], and in the synchronous KT0 model, [20, 15, 7, 17]. Recently, in the synchronous KT0 model, Pandurangan gave a [23] $\tilde{O}(D + \sqrt{n})$ time and $\tilde{O}(m)$ message randomized algorithm, which Elkin improved by logarithmic factors with a deterministic algorithm [11]. The time complexity to compute spanning tree in the algorithm of [18] is $O(n \log n)$ which was improved to $O(n)$ in [21].

Lower bounds on time for approximating the minimum spanning tree has been proved in the synchronous KT0 model In [8, 25] . Kutten et al. [19] show an $\Omega(m)$ lower bound on message complexity for randomized general algorithms in the KT0 model.

$FindAny$ and $FindMin$ which appear in the KKT algorithms build on ideas for sequential dynamic connectivity in [16]. A sequential dynamic *ApproxCut* also appeared in that paper [16]. Solutions to the sequential linear sketching problem for connectivity [1] share similar techniques but require a more complex step to verify when a candidate edge name is an actual edge in the graph, as the edges names are no longer accessible once the sketch is made (See Subsection 2.3).

The *threshold detection* problem was introduced by Emek and Korman [12]. It assumes that there is a rooted spanning tree T where events arrive online at T's nodes. Given some threshold k, a termination signal is broadcast by the root if and only if the number of events exceeds k. We use a naive solution of a simple version of the problem here.

A synchronizer, introduced by Awerbuch [2] and studied in [6, 5, 24, 9], is a general technique for simulating a synchronous algorithm on an asynchronous network using communications along a spanning tree. To do this, the spanning tree must be built first. Using a general synchronizer imposes an overhead of messages that affect *every single step* of the synchronous algorithm that one wants to simulate, and would require more communication than our special purpose method of using our spanning tree to synchronize the modified GHS.

1.3 Organization

Section 2 describes the model. Section 3 gives the spanning tree algorithm for the case of a connected network and a single known leader. Finally, Section 4 provides the MST algorithm. (Due to lack of space the algorithm for computing a minimum spanning *forest* in disconnected graphs or minimum spanning tree for dealing with the case of no pre-selected leader is available on the arxiv version [22]. This variant of the algorithm has the same message complexity.)

2 Preliminaries

2.1 Model

Let $c \geq 1$ be any constant. The communications network is the undirected graph $G = (V, E)$ over which a spanning tree or MST will be found. Edge weights are integers in $[1, n^c]$. IDs are assigned uniquely by the adversary from $[1, n^c]$. All nodes have knowledge of c and n which is an upper bound on $|V|$ (number of nodes in the network) within a *constant* factor. All nodes know their own ID along with the ID of their neighbors (KT1 model) and the weights of their incident edges. Nodes have no other information about the network. e.g., they do not know $|E|$ or the maximum degree of the nodes in the network. Nodes can only send direct messages to the nodes that are adjacent to them in the network. If the edge weights are not unique they can be made unique by appending the ID of the endpoints to its weight, so that the MST is unique. Nodes can only send direct messages to the nodes that are adjacent to them in the network. Our algorithm is described in the CONGEST model in which each message has size $O(\log n)$. Its time is trivially bounded by the total number of messages. The KT1 CONGEST model has been referred to as the "standard model" [4].

Message cost is the sum over all edges of the number of messages sent over each edge during the execution of the algorithm. If a message is sent it is eventually received, but the adversary controls the length of the delays and there is no guarantee that messages sent by the same node will be received in the order they are sent. There is no global clock. All nodes awake at the start of the protocol simultaneously. After awaking and possibly sending its initial messages, a processor acts only in response to receiving messages.

We say a network "finds" a subgraph if at the end of the distributed algorithm, every node knows exactly which of its incident edges in the network are part of the subgraph. The algorithm here is Monte Carlo, in that it succeeds with probability $1 - n^{-c''}$ for any constant c'' ("w.h.p.").

We initially assume there is a special node (called *leader*) at the start and the graph is connected. These assumptions are dropped in the algorithm we provide for disconnected graphs in the full version of the paper.

2.2 Definitions and Subroutines

T is initially a tree containing only the leader node. Thereafter, T is a tree rooted at the leader node. We use the term *outgoing edge* from T to mean an edge with exactly one endpoint in T. An outgoing edge is described as if it is directed; it is *from* a node in T and *to* a node not in T (the "external" endpoint).

The algorithm uses the following subroutines and definitions:

- $Broadcast(M)$: Procedure whereby the node v in T sends message M to its children and its children broadcast to their subtrees.
- $Expand$: A procedure for adding nodes to T and preserving the Invariant after doing so.
- $FindAny$: Returns to the leader an outgoing edge chosen uniformly at random with probability $1/16$, or else it returns $\emptyset$. The leader then broadcasts the result. $FindAny$ requires $O(n)$ messages. We specify $FindAny(E')$ when we mean that the outgoing edge must be an outgoing edge in a particular subset $E' \subseteq E$.
- $FindMin$: is similarly defined except the edge is the (unique) minimum cost outgoing edge. This is used only in the minimum spanning tree algorithm. $FindMin$ requires $O(n \log^2 n / \log \log n)$ messages.

- *ApproxCut*: A function which w.h.p. returns an estimate in $[k/32, k]$ where k is the number of outgoing edges from T and $k > c \log n$ for c a constant. It requires $O(n \log n)$ messages.

  FindAny and *FindMin* are described in [18] (The *FindAny* we use is called *FindAny-C* there.) *FindAny-C* was used to find *any* outgoing edge in the previous paper. It is not hard to see that the edge found is a random edge from the set of outgoing edges; we use that fact here. The relationships among *FindAny*, *FindMin* and *ApproxCut* below are described in the next subsection.
- $Found_L(v)$, $Found_O(v)$: Two lists of edges incident to node v, over which v will send invitations to join T the next time v participates in *Expand*. After this, the list is emptied. Edges are added to $Found_L(v)$ when v receives ⟨*Low-degree*⟩ message or the edge is found by the leader by sampling and its external endpoint is low-degree. Otherwise, an edge is added to $Found_O(v)$ when v receives a ⟨*Star*⟩ message over an edge or if the edge is found by the leader by sampling and its external endpoint is high-degree. Note that star nodes that are low-degree send both ⟨*Low-degree*⟩ and ⟨*Star*⟩. This may cause an edge to be in both lists which is handled properly in the algorithm.
- *T-neighbor*(v): A list of neighbors of v in T. This list, except perhaps during the execution of *Expand*, includes all low-degree neighbors of v in T. This list is used to exclude from $Found_L(v)$ any non-outgoing edges.
- *ThresholdDetection*(k): A procedure which is initiated by the leader of T. The nodes in T experience no more than $k < n^2$ events w.h.p. The leader is informed w.h.p. when the number of events experienced by the nodes in T reaches the threshold $k/4$. Here, an event is the receipt of ⟨*Low-degree*⟩ over an outgoing edge. Following the completion of *Expand*, all edges (u, v) in $Found_L(u)$ are events if $v \notin$ *T-neighbor*(u). $O(|T| \log n)$ messages suffice.

2.3 Implementation of *FindAny*, *FindMin* and *ApproxCut*

We briefly review *FindAny* in [18] and explain its connection with *ApproxCut*. The key insight is that an outgoing edge is incident to exactly one endpoint in T while other edges are incident to zero or two endpoints. If there were exactly one outgoing edge, the parity of the sum of all degrees in T would be 1, and the parity of bit-wise XOR of the binary representation of the names of all incident edges would be the name of the one outgoing edge.

To deal with possibility of more than one outgoing edge, the leader creates an efficient means of sampling edges at different rates: Let $l = \lceil 2 \log n \rceil$. The leader selects and broadcasts one pairwise independent hash function $h : [edge\_names] \rightarrow [1, 2^l]$, where *edge_name* of an edge is a unique binary string computable by both its endpoints, e.g., $\{x, y\} = x \cdot y$ for $x < y$. Each node y forms the vector $\overrightarrow{h(y)}$ whose i^{th} bit is the parity of its incident edges that hash to $[0, 2^i]$, $i = 0, \ldots, l$. Starting with the leaves, a node in T computes the bitwise XOR of the vectors from its children and itself and then passes this up the tree, until the leader has computed $\overrightarrow{b} = XOR_{y \in T} \overrightarrow{h(y)}$. The key insight implies that for each index i, $\overrightarrow{b_i}$ equals the parity of just the outgoing edges mapped to $[0, 2^i]$. Let *min* be the smallest index i s.t. $\overrightarrow{b_i} = 1$. With constant probability, exactly one edge of the outgoing edges has been mapped to $[1, 2^{min}]$. The leader broadcasts *min*. Nodes send back up the XOR of the *edge_names* of incident edges which are mapped by h to this range. If exactly one outgoing edge has been indeed mapped to that range, the leader will find it by again determining the XOR of the *edge_names* sent up. One more broadcast from the leader can be used to verify that this edge exists and is incident to exactly one node in T.

Since each edge has the same probability of failing in $[0, 2^{min}]$, this procedure gives a randomly selected edge. Note also that the leader can instruct the nodes to exclude certain edges from the XOR, say incident edges of weight greater than some w. In this way the leader can binary search for the minimal weight outgoing edge to carry out $FindMin$. Similarly, the leader can select random edges without replacement.

Observe that if the number of outgoing edges is close to 2^j, we'd expect min to be $l - j$ with constant probability. Here we introduce distributed asynchronous *ApproxCut* which uses the sampling technique from $FindMin$ but repeats it $O(\log n)$ times with $O(\log n)$ randomly chosen hash functions. Let min_sum be the minimum i for which the sum of $\overrightarrow{b_i}$s exceeds $c \log n$ for some constant c. We show 2^{min_sum} approximates the number of outgoing edges within a constant factor from the actual number. *ApproxCut* pseudocode is given in Algorithm 5.

We show:

▶ **Lemma 2.** *With probability* $1 - 1/n^c$, ApproxCut *returns an estimate in* $[k/32, k]$ *where* k *is the number of outgoing edges and* $k > c' \log n$, c' *a constant depending on* c. *It uses* $O(n \log n)$ *messages.*

The proof is given in Section 3.2.

3 Asynchronous ST construction with $o(m)$ messages

In this section we explain how to construct a spanning tree when there is a preselected leader and the graph is connected.

Initially, each node selects itself with probability $1/\sqrt{n \log n}$ as a *star node*. Low-degree and star nodes initially send out $\langle$*Low-degree*$\rangle$ and $\langle$*Star*$\rangle$ messages to all of their neighbors, respectively. (We will be using the $\langle M \rangle$ notation to show a message with content M.) A low-degree node which is a star node sends both types of messages. At any point during the algorithm, if a node v receives a $\langle$*Low-degree*$\rangle$ or $\langle$*Star*$\rangle$ message through some edge e, it adds e to $Found_L(v)$ or $Found_O(v)$ resp.

The algorithm FindST-Leader runs in phases. Each phase has three parts: **1) Expansion** of T over found edges since the previous phase and restoration of the Invariant; **2) Search** for an outgoing edge to a high-degree node; **3) Wait** until messages to nodes in T have been received over a *constant fraction of the outgoing edges* whose external endpoint is low-degree.

1) Expansion. Each phase is started with *Expand*. *Expand* adds to T any nodes which are external endpoints of outgoing edges placed on a *Found* list of any node in T since the last time that node executed *Expand*. In addition, it restores the Invariant for T.

Implementation. Expand is initiated by the leader and broadcast down the tree. When a node v receives $\langle$*Expand*$\rangle$ message for the first time (it is not in T), it joins T and makes the sender its parent. If it is a high-degree node and is not a star, it has to wait until it receives a $\langle$*Star*$\rangle$ message over some edge e, and then adds e to $Found_O(v)$. It then forwards $\langle$*Expand*$\rangle$ over the edges in $Found_L(v)$ or $Found_O(v)$ and empties these lists. Otherwise, if it is a low-degree node or a star node, it forwards $\langle$*Expand*$\rangle$ to *all* of its neighbors.

On the other hand, if v is already in T, it forwards $\langle$*Expand*$\rangle$ message to its children in T and along any edges in $Found_L(v)$ or $Found_O(v)$, i.e. outgoing edges which were "found" since the previous phase, and empties these lists. All $\langle$*Expand*$\rangle$ requests received by v are answered, and their sender is added to *T-neighbor*(v). The procedure ends in a bottom-up way and ensures that each node has heard from all the nodes it sent ⟨*Expand*⟩ requests to before it contacts its parent.

Let T^i denote T after the execution of *Expand* in phase i. Initially T^0 consists of the leader node and as its Found lists contain all its neighbors, after the first execution of *Expand*, if the leader is high-degree, T_1 satisfies the invariant. An easy inductive argument on T^i shows:

▶ **Observation 1.** *For all $i > 0$, upon completion of Expand, all the nodes reachable by edges in the Found lists of any node in T^{i-1} are in T^i, and for all $v \in T$, T-neighbor(v) contains all the low-degree neighbors of v in T.*

Expand is called in line 6 of the main algorithm 1. The pseudocode is given in *Expand* Algorithm 1.

2) Search for an outgoing edge to a high degree node. A sampling of the outgoing edges without replacement is done using $FindAny$ multiple times. The sampling either (1) finds an outgoing edge to a high degree node, or (2) finds all outgoing edges, or (3) determines w.h.p. that at least half the outgoing edges are to low-degree nodes and there are at least $2c \log n$ such edges. If the first two cases occur, the phase ends.

Implementation. Endpoints of sampled edges in T communicate over the outgoing edge to determine if the external endpoint is high-degree. If at least one is, that edge is added to the $Found_O$ list of its endpoint in T and the phase ends. If there are fewer than $2 \log n$ outgoing edges, all these edges are added to $Found_O$ and the phase ends. If there are no outgoing edges, the algorithm ends. If all $2 \log n$ edges go to low-degree nodes, then the phase continues with Step 3) below. This is implemented in the **while** loop of FindST-Leader.

Throughout this section we will be using the following fact from Chernoff bounds:
Assume $X_1, X_2, \ldots, X_T$ are independent Bernoulli trials where each trial's outcome is 1 with probability $0 < p < 1$. Chernoff bounds imply that given constants c, $c_1 > 1$ and $c_2 < 1$ there is a constant c'' such that if there are $T \geq c'' \log n$ independent trials, then $Pr(X > c_1 \cdot E[X]) < 1/n^c$ and $Pr(X < c_2 \cdot E[X]) < 1/n^c$, where X is sum of the $X_1, \ldots, X_T$.

We show:

▶ **Lemma 3.** *After Search, at least one of the following must be true with probability $1 - 1/n^{c'}$, where c' is a constant depending on c: 1) there are fewer than $2c \log n$ outgoing edges and the leader learns them all; 2) an outgoing edge is to a high-degree node is found, or 3) there are at least $2c \log n$ outgoing edges and at least half the outgoing edges are to low-degree nodes.*

Proof. Each *FindAny* has a probability of 1/16 of returning an outgoing edge and if it returns an edge, it is always outgoing. After $48c \log n$ repetitions without replacement, the expected number of edges returned is $3c \log n$. As these trials are independent, Chernoff bounds imply that at least 2/3 of trials will be successful with probability at least $1 - 1/n^c$, i.e., $2c \log n$ edges are returned if there are that many, and if there are fewer, all will be returned.

The edges are picked uniformly at random by independent repetitions of $FindAny$. If more than half the outgoing edges are to high-degree nodes, the probability that all edges returned are to low-degree nodes is $1/2^{2c \log n} < 1/n^{2c}$. ◀

3) Wait to hear from outgoing edges to low-degree external nodes. This step forces the leader to wait until T has been contacted over a constant fraction of the outgoing edges to (external) low-degree nodes. Note that we do not know how to give a good estimate on the number of low-degree nodes which are neighbors of T. Instead we count outgoing edges.

Implementation. This step occurs only if the $2c\log n$ randomly sampled outgoing edges all go to low-degree nodes and therefore the number of outgoing edges to low-degree nodes is at least this number. In this case, the leader waits until T has been contacted through a constant fraction of these edges.

If this step occurs, then w.h.p., at least half the outgoing edges go to low-degree nodes. Let k be the number of outgoing edges; $k \geq 2c\log n$. The leader calls *ApproxCut* to return an estimate $q \in [k/32, k]$ w.h.p. It follows that w.h.p. the number of outgoing edges to low-degree nodes is $k/2$. Let $r = q/2$. Then $r \in [k/64, k/2]$.

The nodes $v \in T$ will eventually receive at least $k/2$ messages over outgoing edges of the form $\langle \textit{Low-degree} \rangle$. Note that these messages must have been received by v after v executed *Expand* and added to $Found_L(v)$, for otherwise, these would not be outgoing edges.

The leader initiates a *ThresholdDetection* procedure whereby there is an event for a node v for each outgoing edge v has received a $\langle \textit{Low-degree} \rangle$ message over since the last time v executed *Expand*. As the *ThresholdDetection* procedure is initiated after the leader finishes *Expand*, the T-*neighbor*(v) includes any low-degree neighbor of v that is in T. Using T-*neighbor*(v), v can determine which edges in $Found_L(v)$ are outgoing.

Each event experienced by a node causes it to flip a coin with probability $\min\{c\log n/r, 1\}$. If the coin is heads, then a trigger message labelled with the phase number is sent up to the leader. The leader is triggered if it receives at least $(c/2)\log n$ trigger messages for that phase. When the leader is triggered, it begins a new phase. Since there are $k/2$ triggering events, the expected number of trigger messages eventually generated is $(c\log n/r)(k/2) \geq c\log n$. Chernoff bounds imply that at least $(c/2)\log n$ trigger messages will be generated w.h.p. Alternatively, w.h.p., the number of trigger messages received by the leader will not exceed $(c/2)\log n$ until at least $k/8$ events have occurred, as this would imply twice the expected number. We can conclude that w.h.p. the leader will trigger the next phase after 1/4 of the outgoing edges to low-degree nodes have been found.

▶ **Lemma 4.** *When the leader receives $(c/2)\log n$ messages with the current phase number, w.h.p, at least 1/4 of the outgoing edges to low-degree nodes have been added to $Found_L$ lists.*

3.1 Proof of the main theorem

Here we prove Theorem 1 as it applies to computing the spanning tree of a connected network with a pre-selected leader.

▶ **Lemma 5.** *W.h.p., after each phase except perhaps the first, either **(A)** A high-degree node (and star node) is added to T or **(B)** T is expanded so that the number of outgoing edges to low-degree nodes is reduced by a 1/4 factor (or the algorithm terminates with a spanning tree).*

Proof. By Lemma 3 there are three possible results from the **Search** phase. If a sampled outgoing edge to a high-degree node is found, this edge will be added to the $Found_O$ list of its endpoint in T. If the **Search** phase ends in fewer than $2c\log n$ edges found and none of them are to high degree nodes, then w.h.p. these are all the outgoing edges to low-degree nodes, these edges will all be added to some $Found_L$. If there are no outgoing edges, the algorithm terminates and a spanning tree has been found. If the third possible result occurs, then there are at least $2\log n$ outgoing edges, half of which go to low-degree nodes. By Lemma 4, the leader will trigger the next phase and it will do so after at least 1/4 of the outgoing edges to low-degree nodes have been added to $Found_L$ lists.

By Observation 1, all the endpoints of the edges on the $Found$ lists will be added to T in the next phase, and there is at least one such edge or there are no outgoing edges and the spanning tree has been found. When *Expand* is called in the next phase, T acquires a new high degree node in two possible ways, either because an outgoing edge on a $Found$ list is to a high-degree node or because the recursive *Expand* on outgoing edges to low-degree edges eventually leads to an external high-degree node. In either case, by the Invariant, T will acquire a new star node as well as a high-degree node. Also by the Invariant, all outgoing edges must come from high-degree nodes. Therefore, if no high-degree nodes are added to T by Expand, then no new outgoing edges are added to T. On the other hand, 1/4 of the outgoing edges to low-degree nodes have become non-outgoing edges as their endpoints have been added to T. So we can conclude that the number of outgoing edges to low-degree nodes have been decreased by 1/4 factor. ◀

It is not hard to see:

▶ **Lemma 6.** *The number of phases is bounded by* $O(\sqrt{n}\log^{1/2} n)$.

Proof. By Lemma 5, every phase except perhaps the first, is of type A or type B. Chernoff bounds imply that w.h.p., the number of star nodes does not exceed its expected number $(\sqrt{n}/\log^{1/2} n)$ by more than a constant factor, hence there are no more than $O(\sqrt{n}/\log^{1/2} n)$ phases of type A. Before and after each such phase, the number of outgoing edges to low-degree nodes is reduced by at least a fraction of 1/4; hence, there are no more than $\log_{4/3} n^2 = O(\log n)$ phases of type B between phases of type A. ◀

Finally, we count the number of messages needed to compute the spanning tree.

▶ **Lemma 7.** *The overall number of messages is* $O(n^{3/2}\log^{3/2} n)$.

Proof. The initialization requires $O(\sqrt{n}\log^{3/2} n)$ messages from $O(n)$ low-degree nodes and $O(n)$ messages from each of $O(\sqrt{n}/\log^{1/2} n)$ stars. In each phase, *Expand* requires a number of messages which is linear in the size of T or $O(n)$, except that newly added low-degree and star nodes send to their neighbors when they are added to T, but this adds just a constant factor to the initialization cost. $FindAny$ is repeated $O(\log n)$ times for a total cost of $O(n\log n)$ messages. $ApproxCut$ requires the same number. The Threshold Detector requires only $O(\log n)$ messages to be passed up T or $O(n\log n)$ messages overall. Therefore, by Lemma 6 the number of messages over all phases is $O(n\log^{3/2} n)$. ◀

Theorem 1 for spanning trees in connected networks with a pre-selected leader follows from Lemmas 7 and 6.

3.2 Proof of $ApproxCut$ Lemma

Proof. Let W be the set of the outgoing edges. For a fixed z and i, we have:

$$Pr(h_{z,i}(T) = 1) = Pr(\textit{an odd number of edges in } W \textit{ hash to } [2^i]) \geq$$

$$Pr(\exists!\ e \in W \textit{hashed to } [2^i]).$$

This probability is at least $1/16$ for $i = l - \lceil \log |W| \rceil - 2$ (Lemma 5 of [18]). Therefore, since $X_j = \sum_{z=1}^{c \log n} h_{z,j}$ (from pseudocode), $E[X_j] = \sum E[h_{z,j}] \geq c \log n/16$, where $j = l - \lceil \log |W| \rceil - 2$. Note that $j = l - \lceil \log |W| \rceil - 2$ means that $\frac{2^l}{2^{j+3}} < |W| < \frac{2^l}{2^{j+1}}$. Consider $j - 4$. Since the probability of an edge being hashed to $[2^{j-4}]$ is $\frac{2^{j-4}}{2^l}$, we have

$$Pr(h_{z,j-4}(T) = 1) \leq Pr(\exists e \in W hashed\ to\ [2^{j-4}]) = |W| \frac{2^{j-4}}{2^l} \leq \frac{1}{2^5} \leq \frac{1}{32}.$$

Thus, $E[X_{j-4}] \leq c \log n/32$. Since an edge that is hashed to $[2^{j-k}]$ (for $k > 4$) is already hashed to $[2^{j-4}]$, we have:

$$Pr(h_{z,j-4}(T) = 1 \vee \ldots \vee h_{z,0}(T) = 1) \leq Pr(\exists e \in W hashed\ to\ [2^{j-4}] or \ldots or [2^0])) =$$

$$Pr(\exists e \in W hashed\ to\ [2^{j-4}]) = \frac{1}{32}.$$

Let y_z be 1 if $h_{z,j-4}(T) = 1 \vee \ldots \vee h_{z,0}(T) = 1$, and 0 otherwise. Also, let $Y = \sum_{z=1}^{c \log n} y_z$. We have$E[Y] \leq c \log n/32$. Also, for any positive integer a,

$$Pr(X_{j-4} > a \vee \ldots \vee X_0 > a) \leq Pr(Y > a).$$

From Chernoff bounds:

$$Pr(X_j < (3/4)c \log n/16) = Pr(X_j < (3/4)E[X_j]) < 1/n^{c'}$$

and,

$$Pr(X_{j-4} > (3/2)c \log n/16 \vee \ldots \vee X_0 > (3/2)c \log n/16) \leq Pr(Y > (3/2)c \log n/16) =$$

$$Pr(Y > (3/2)c \log n/32) < Pr(Y > (3/2)E[Y]) < 1/n^{c'}.$$

Therefore, by finding the smallest i (called min in pseudocode) for which $X_i > (3/2)c \log n/16$, w.h.p. min is in $[j-3, j]$. As a result, $2|W| \leq 2^{l-min} \leq 64|W|$. Therefore,
$|W|/32 \leq 2^{l-min}/64 \leq |W|$.

Furthermore, broadcasting each of the $O(\log n)$ hash functions and computing the corresponding vector takes $O(n)$ messages; so, the lemma follows. ◀

3.3 Pseudocode

Algorithm 1 Initialization of the spanning tree algorithm.

```
1: procedure INITIALIZATION
2:     Every node selects itself to be a star node with probability of $1/\sqrt{n \log n}$.
3:     Nodes that have degree $< \sqrt{n} \log^{3/2} n$ are low-degree nodes. Otherwise, they are
       high-degree nodes. (Note that they may also be star nodes at the same time.)
4:     Star nodes send ⟨Star⟩ messages to all of their neighbors.
5:     Low-degree nodes send ⟨Low-degree⟩ messages to all of their neighbors (even if they
       are star nodes too).
6: end procedure
```

Algorithm 2 Asynchronous protocol for the leader to find a spanning tree.

```
1: procedure FindST-Leader
2:     Leader initially adds all of its incident edges to its $Found_L$ list. // By exception
       leader does not need to differentiate between $Found_L$ and $Found_O$
3:     $i \leftarrow 0$
4:     repeat (Phase $i$)
5:         $i \leftarrow i + 1$.
6:         Leader calls $Expand()$. // Expansion
           // Search and Sampling:
7:         $counter \leftarrow 0, A \leftarrow \emptyset$.
8:         while $counter < 48c \log n$ do
9:             $FindAny(E \setminus A)$.
10:            if $FindAny$ is successful and finds an edge $(u, v)$ ($u \in T$ and $v \notin T$) then
11:                $u$ sends a message to $v$ to query $v$'s degree, and sends it to the leader.
12:                $u$ adds $(u, v)$ to either $Found_L(u)$ or $Found_O(u)$ based on $v$'s degree.
13:            end if
14:            $counter \leftarrow counter + 1$.
15:        end while
16:        if $|A| = 0$ then
17:            terminate the algorithm as there are no outgoing edges.
18:        else if $|A| < 2 \log n$ (few edges) or $\exists (u, v) \in A$ s.t. $v$ is high-degree then
19:            Leader starts a new phase to restore the Invariant.
20:        else (at least half of the outgoing edges are to low-degree nodes) // Wait:
21:            $r \leftarrow ApproxCut()/2$.
22:            Leader calls $ThresholdDetection(r)$.
23:            Leader waits to trigger and then starts a new phase.
24:        end if
25:    until
26: end procedure
```

Algorithm 3 Given r at phase i, this procedure detects when nodes in T receive at least $r/4$ $\langle Low - degree \rangle$ messages over outgoing edges. c is a constant.

```
1: procedure ThresholdDetection
2:     Leader calls Broadcast($\langle Send\text{-}trigger, r, i \rangle$).
3:     When a node $u \in T$ receives $\langle Send\text{-}trigger, r, i \rangle$, it first participates in the broadcast.
       Then, for every event, i.e. every edge $(u, v) \in Found(u)_L$ such that $v \notin T\text{-}neighbor(u)$,
       $u$ sends to its parent a $\langle Trigger, i \rangle$ message with probability of $c \log n / r$.
4:     A node that receives $\langle Trigger, i \rangle$ from a child keeps sending up the message until it
       reaches the leader. If a node receives an $\langle Expand \rangle$ before it sends up a $\langle Trigger, i \rangle$, it
       discards the $\langle Trigger, i \rangle$ messages as an Expand has already been triggered.
5:     Once the leader receives at least $c \log n / 2$ $\langle Trigger, i \rangle$ messages, the procedure terminates
       and the control is returned to the calling procedure.
6: end procedure
```

Algorithm 4 Leader initiates Expand by sending ⟨*Expand*⟩ to all of its children. If this is the first Expand, leader sends to all of its neighbors. Here, x is any non-leader node.

```
 1: procedure Expand
 2:     When node $x$ receives an ⟨Expand⟩ message over an edge $(x, y)$:
 3:     $x$ adds $y$ to $T\text{-}neighbor(x)$.
 4:     if $x$ is not in $T$ then
 5:         The first node that $x$ receives ⟨Expand⟩ from becomes $x$'s parent. //$x$ joins $T$
 6:         if $x$ is a high-degree node and $x$ is not a star node then
 7:             It waits to receive a ⟨Star⟩ over some edge $e$, then adds $e$ to $Found_O(x)$.
 8:             It forwards ⟨Expand⟩ over edges in $Found_L(x)$ and $Found_O(x)$ (only once in
                case an edge is in both lists), then removes those edges from the Found lists.
 9:         else ($x$ is a low-degree or star node)
10:             It forwards the ⟨Expand⟩ message to all of its neighbors.
11:         end if
12:     else ($x$ is already in $T$)
13:         If the sender is not its parent, it sends back ⟨Done-by-reject⟩. Else, it forwards
            ⟨Expand⟩ to its children in $T$, over the edges in $Found_L(x)$ and $Found_O(x)$,
            then removes those edges from the Found lists.
14:     end if
    // Note that if $x$ added more edges to its Found list after forward of
    ⟨Expand⟩, the new edges will be dealt with in the next Expand.
15:     When a node receives ⟨Done⟩ messages (either ⟨Done-by-accept⟩ or ⟨Done-by-reject⟩)
        from all of the nodes it has sent to, it considers all nodes that have sent
        ⟨Done-by-accept⟩ as its children. Then, it sends up ⟨Done-by-accept⟩ to its parent.
16:     The algorithm terminates when the leader receives ⟨Done⟩ from all of its children.
17: end procedure
```

Algorithm 5 Approximates the number of outgoing edges within a constant factor. c is a constant.

```
1: procedure ApproxCut($T$)
2:     Leader broadcasts $c \log n$ random 2-wise independent hash functions defined from
    $[1, n^{2c}] \to [2^l]$.
3:     For node $y$, and hash function $h_z$ vector $\overrightarrow{h_z}(y)$ is computed where $h_{z,i}(y)$ is the parity
       of incident edges that hash to $[2^i]$, $i = 0, \ldots, l$.
4:     For hash function $h_z$, $\overrightarrow{h_z}(T) = \oplus_{y \in T} \overrightarrow{h_z}(y)$ is computed in the leader.
5:     For each $i = 0, \ldots, l$, $X_i = \sum_{z=1}^{c \log n} h_{z,i}(T)$.
6:     Let $min$ be the smallest $i$ s.t. $X_i \geq (3/4)c \log n/16$.
7:     Return $2^{l-min}/64$.
8: end procedure
```

4 Finding MST with $o(m)$ asynchronous communication

The MST algorithm implements a version of the GHS algorithm which grows a forest of disjoint subtrees (“fragments”) of the MST in parallel. We reduce the message complexity of GHS by using *FindMin* to find minimum weight outgoing edges *without* having to send messages across every edge. But, by doing this, we require the use of a spanning tree to help synchronize the growth of the fragments.

Note that GHS nodes send messages along their incident edges for two main purposes: (1) to see whether the edge is outgoing, and (2) to make sure that fragments with higher rank are slowed down and do not impose a lot of time and message complexity. Therefore, if we use $FindMin$ instead of having nodes to send messages to their neighbors, we cannot make sure that higher ranked fragments are slowed down. Our protocol works in phases where in each phase only fragments with smallest ranks continue to grow while other fragments wait. A spanning tree is used to control the fragments based on their rank. (See [14] for the original GHS.)

Implementation of FindMST. Initially, each node forms a fragment containing only that node which is also the leader of the fragment and fragments all have rank zero. A *fragment identity* is the node ID of the fragment's leader; all nodes in a fragment know its identity and its current rank. Let the pre-computed spanning tree T be rooted at a node r, All fragment leaders wait for instructions that are broadcast by r over T.

The algorithm runs in phases. At the start of each phase, r broadcasts the message $\langle \mathit{Rank\text{-}request} \rangle$ to learn the current minimum rank among all fragments after this broadcast. Leaves of T send up their fragment rank. Once an internal node in T receives the rank from all of its children (in T) the node sends up the minimum fragment rank it has received including its own. This kind of computation is also referred to as a *convergecast*.

Then, r broadcasts the message $\langle \mathit{Proceed}, \mathit{minRank} \rangle$ where $\mathit{minRank}$ is the current minimum rank among all fragments. Any fragment leader that has rank equal to $\mathit{minRank}$, *proceeds* to finding minimum weight outgoing edges by calling FindMin on its own fragment tree. These fragments then send a $\langle \mathit{Connect} \rangle$ message over their minimum weight outgoing edges. When a node v in fragment F (at rank R) sends a $\langle \mathit{Connect} \rangle$ message over an edge e to a node v' in fragment F' (at rank R'), since R is the current minimum rank, two cases may happen: (Ranks and identities are updated here.)

1. $\boldsymbol{R < R'}$: In this case, v' answers immediately to v by sending back an $\langle \mathit{Accept} \rangle$ message, indicating that F can merge with F'. Then, v initiates the merge by changing its fragment identity to the identity of F', making v' its parent, and broadcasting F''s identity over fragment F so that all nodes in F update their fragment identity as well. Also, the new fragment (containing F and F') has rank R'.
2. $\boldsymbol{R = R'}$: v' responds $\langle \mathit{Accept} \rangle$ immediately to v if the minimum outgoing edge of F' is e, as well. In this case, F merges with F' as mentioned in rule 1, and the new fragment will have F''s identity. Also, both fragments increase their rank to $R' + 1$.
   Otherwise, v' does not respond to the message until F''s rank increases. Once F' increased its rank, it responds via an $\langle \mathit{Accept} \rangle$ message, fragments merge, and the new fragment will update its rank to R'.

The key point here is that fragments at minimum rank are not kept waiting. Also, the intuition behind rule 2 is as follows. Imagine we have fragments $F_1, F_2, ..., F_k$ which all have the same rank and F_i's minimum outgoing edge goes to F_{i+1} for $i \leq k-1$. Now, it is either the case that F_k's minimum outgoing edge goes to a fragment with higher rank or it goes to F_k. In either case, rule 2 allows the fragments $F_{k-1}, F_{k-2}, \ldots$ to update their identities in a cascading manner right after F_k increased its rank.

When all fragments finish their merge at this phase they have increased their rank by at least one. Now, it is time for r to star a new phase. However, since communication is asynchronous we need a way to tell whether all fragments have finished. In order to do this, $\langle \mathit{Done} \rangle$ messages are convergecast in T. Nodes that were at minimum rank send up to their parent in T a $\langle \mathit{Done} \rangle$ message only after they increased their rank and received $\langle \mathit{Done} \rangle$ messages from all of their children in T.

Algorithm 6 MST construction with $\tilde{O}(n)$ messages. T is a spanning tree rooted at r.

```
1: procedure FindMST
2:     All nodes are initialized as fragments at rank 0.
       // Start of a phase
3:     $r$ calls Broadcast($\langle$Rank-request$\rangle$), and $minRank$ is computed via a convergecast.
4:     $r$ calls Broadcast($\langle Proceed, minRank\rangle$).
5:     Fragment leaders at rank $minRank$ that have received the $\langle Proceed, minRank\rangle$
       message, call FindMin. Then, these fragments merge by sending Connect messages
       over their minimum outgoing edges. If there is no outgoing edge the fragment leader
       terminates the algorithm.
6:     Upon receipt of $\langle Proceed, minRank\rangle$, a node $v$ does the following:
       If it is a leaf in $T$ at rank $minRank$, sends up $\langle Done\rangle$ after increasing its rank.
       If it is a leaf in $T$ with a rank higher than $minRank$, it immediately sends up $\langle Done\rangle$.
       If it is not a leaf in $T$, waits for $\langle Done\rangle$ from its children in $T$. Then, sends up the
       $\langle Done\rangle$ message after increasing its rank.
7:     $r$ waits to receive $\langle Done\rangle$ from all of its children, and starts a new phase at step 3.
8: end procedure
```

As proved in Lemma 8, this algorithm uses $\tilde{O}(n)$ messages.

▶ **Lemma 8.** *FindMST uses $O(n \log^3 n/ \log\log n)$ messages and finds the MST w.h.p.*

Proof. All fragments start at rank zero. Before a phase begins, two broadcasts and convergecasts are performed to only allow fragments at minimum rank to proceed. This requires $O(n)$ messages. In each phase, finding the minimum weight outgoing edges using FindMin takes $O(n \log^2 n/ \log\log n)$ over all fragments. Also, it takes $O(n)$ for the fragments to update their identity since they just have to send the identity of the higher ranked fragment over their own fragment. As a result, each phase takes $O(n \log^2 n/ \log\log n)$ messages.

A fragment at rank R must contain at least two fragments with rank $R-1$; therefore, a fragment with rank R must have at least 2^R nodes. So, the rank of a fragment never exceeds $\log n$. Also, each phase increases the minimum rank by at least one. Hence, there are at most $\log n$ phases. As a result, message complexity is $O(n \log^3 n/ \log\log n)$. ◀

From Lemma 8, Theorem 1 for minimum spanning trees follows.

5 Conclusion

We presented the first asynchronous algorithm for computing the MST in the CONGEST model with $\tilde{O}(n^{3/2})$ communication when nodes have initial knowledge of their neighbors' identities. This shows that the KT1 model is significantly more communication efficient than KT0 even in the asynchronous model. Open problems that are raised by these results are: (1) Does the asynchronous KT1 model require substantially more communication that the synchronous KT1 model? (2) Can we improve the time complexity of the algorithm while maintaining the message complexity?

References

1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.

2 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

3 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240. ACM, 1987.

4 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2):238–256, 1990.

5 Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Transactions on Dependable and Secure Computing*, 4(3), 2007.

6 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 514–522. IEEE, 1990.

7 Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 359–368. Society for Industrial and Applied Mathematics, 2004.

8 Michael 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. Synchronizers, spanners. In *Encyclopedia of Algorithms*, pages 1–99. Springer, 2008.

10 Michael Elkin. Distributed exact shortest paths in sublinear time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 757–770, New York, NY, USA, 2017. ACM. doi:10.1145/3055399.3055452.

11 Michael Elkin. A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. *arXiv preprint arXiv:1703.02411*, 2017.

12 Yuval Emek and Amos Korman. Efficient threshold detection in a distributed environment. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 183–191. ACM, 2010.

13 Michalis Faloutsos and Mart Molle. Optimal distributed algorithm for minimum spanning trees revisited. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 231–237. ACM, 1995.

14 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.

15 Juan A Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

16 Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. Society for Industrial and Applied Mathematics, 2013.

17 Maleq Khan and Gopal Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 355–369, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11864219_25.

18 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an mst in a distributed network with o (m) communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 71–80. ACM, 2015.

19 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of leader election. *Journal of the ACM (JACM)*, 62(1):7, 2015.

20 Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 238–251. ACM, 1995.

21 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 8. ACM, 2017.

22 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with $o(m)$ messages in the asynchronous congest model. *arXiv*, 2018. `arXiv:1806.04328`.

23 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*, pages 743–756. ACM, 2017.

24 David Peleg and Jeffrey D Ullman. An optimal synchronizer for the hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 77–85. ACM, 1987.

25 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 Journal on Computing*, 41(5):1235–1265, 2012.

26 Gurdip Singh and Arthur J Bernstein. A highly asynchronous minimum spanning tree protocol. *Distributed Computing*, 8(3):151–161, 1995.