Distributed Broadcast and Minimum Spanning Tree Algorithms with Low Communication Complexity

by

Ali Mashreghi
B.Sc., Ferdowsi University of Mashhad, 2012
M.Sc., Sharif University of Technology, 2014

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Distributed Broadcast and Minimum Spanning Tree Algorithms with Low Communication Complexity

by

Ali Mashreghi
B.Sc., Ferdowsi University of Mashhad, 2012
M.Sc., Sharif University of Technology, 2014

Supervisory Committee

Dr. Valerie King, Supervisor
(Department of Computer Science)

Dr. Bruce M. Kapron, Departmental Member
(Department of Computer Science)

Dr. Lin Cai, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Valerie King, Supervisor
(Department of Computer Science)

Dr. Bruce M. Kapron, Departmental Member
(Department of Computer Science)

Dr. Lin Cai, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

In distributed computing, a set of processors that have their own input collaborate to compute a function. Processors can communicate by exchanging messages of limited size over links available on a predetermined communication network. In this thesis, we consider the problems of broadcast and minimum spanning tree construction in a distributed setting. These problems are of fundamental importance. Efficient solutions for these problems can lead to improvements in algorithms for a number of other distributed problems such as leader election.

Since 1990, due to the "folk theorem" mentioned in Awerbuch et al. JACM, it was believed that to construct a minimum spanning tree (or even broadcast tree) in a network with n processors and m communication links, $\Omega(m)$ messages are needed. However, in 2015, King, Kutten, and Thorup [KKT15] showed that if the nodes initially know the identity of their neighbors, the communication can be brought down to $O(n \log n)$ which is $o(m)$ for sufficiently dense graphs. Our research has been focused on obtaining algorithms for constructing minimum spanning and broadcast trees that use only $o(m)$ messages. At the same time, we have tried to improve the time complexity of our algorithms.

We provide time improvements to the algorithms of King et al. in the synchronous network. Also, we provide the first asynchronous minimum spanning tree algorithm that achieves $o(m)$ message complexity. This research will help to highlight the limitations imposed by asynchrony. It also shows that when nodes initially know the identities of their neighbors, we can design algorithms that break the barrier of $\Omega(m)$ messages proved in models where nodes do not have this knowledge.

Contents

List of Tables

List of Algorithms

ACKNOWLEDGEMENTS

First, I want to thank my supervisor Dr. Valerie King who always supported me and gave me an advice when I felt lost. She always invested a lot of time and energy to make sure that my studies are going in the right direction. This thesis would not have been completed if it was not for her constant care and attention.

I want to thank Dr. Bruce Kapron for the courses he taught me and the support he provided for me during my PhD studies.

I am grateful to Dr. Kapron and Dr. Lin Cai who did me a huge favor by acting as members of both my candidacy exam committee and my supervisory committee. They gave me a lot of useful recommendations which helped to improve the quality of this thesis.

I want to thank Dr. Philipp Woelfel for agreeing to become my external examiner.

I want to thank Dr. Michael Elkin whose talk in Montreal during STOC 2017 gave me very key ideas which led to a significant breakthrough in my thesis.

I want to thank Wendy Beggs, Kath Milinazzo, and Dr. Rich Little who were always patient and helped me out with regulations, services, accommodations, labs, lectures and so much more.

I want to thank my friends in Victoria, specially Avishan and Alireza whose presence was always heartwarming.

In the end, I want to dedicate this thesis to my family, my mother Zahra, my father Mohammadsadegh, and my brothers Milad and Hamid. They supported me in all stages of life. They sacrificed their own comfort to make sure that I can stay focused on my studies. They always gave me all they had but never asked for anything. I owe, not only this thesis, but my happiness, well-being, and peace of mind to them and their unconditional love.

Chapter 1

Introduction

1.1 What is Distributed Computing?

In distributed computing, a set of processors that each have their own input collaborate to compute a function of these inputs. We consider these processors to be nodes which can communicate via bi-directional links on a communication network such as the internet. We do not assume the existence of shared memory.

Each node knows the algorithm. An algorithm is also referred to as a *protocol*. In each step, a node sends messages and performs local computations according to the protocol. Throughout this thesis, n and m denote the number of nodes and the number of communication links in the network, respectively.

1.2 Models of Distributed Computing

We consider the message passing model in which passing messages is the only means of communication. An important assumption in this model is that the cost of passing messages dominates other costs. For example, a node's internal computations can be done much faster than passing messages. However, we do not to abuse this assumption. For example, we do not assume that the nodes can run exponential time algorithms faster than the time required for any message to be transmitted. The following is a list of criteria based on which a distributed computing model is determined:

- **Communication Graph:** A communication graph determines which nodes can talk (send messages) to each other. For example, in the case of a complete

communication graph, all nodes can talk to each other directly. In this thesis, we always assume that the input graph corresponds to the communication graph. The communication graph is considered undirected which implies that messages can be sent in either direction.

- **Synchrony:** Communication between the nodes can be synchronous or asynchronous.

  Synchronous: Messages are sent in *rounds* also called *time steps*. At the beginning of each round, all nodes that want to send a message will send it, and all of these messages will be delivered by the end of this round. In other words, all nodes are synchronized according to a global clock and transmitting a message takes one time unit.

  Asynchronous: In this model, communication is event-driven. Nodes send messages (or wait) based on *events*. An event is the receipt of a message. In fact, nodes are not synchronized. Although they can measure time for themselves, clocks of different nodes work at different unknown rates which does not allow the nodes to be synchronized. Moreover, when messages are sent they will be delivered eventually but they could be delayed arbitrarily. The algorithm does not know an upper bound on these delays. We will elaborate on this in Section 1.3.

- **Failure:** Although various models exist based on the failure of nodes and links, we do not assume any failure in this thesis.

- **Order of Messages (in a link):** In the synchronous model, we assume that messages are received in the same order they were sent. However, in the asynchronous model, we do not make this assumption since arbitrary delays can change the order in which the messages are received.

- **Size of Messages (Congestion vs Locality)** [Pel00, Pri05] Two main models based on the message size are:

  CONGEST: This model puts a limit of $O(\log n)$ bits on the size of messages. This model accounts for the effect of *congestion* on the efficiency of the algorithms.

  LOCAL: This model allows for arbitrarily long messages since its purpose is to analyze the limitations that are imposed by having only local information.

Our algorithms assume the CONGEST model.

- **Knowledge of the Topology:** In a distributed system nodes do not have global knowledge about the network's topology. However, there are different levels of how much a node knows *initially* about the topology of the network. The least amount of knowledge is reflected in the KT0 model in which a node does not know the ID of its neighbors [AGVP90]. A node has only ports that connect it to its neighbors, but it does not know which port goes to which neighbor, unless it obtains this information by exchanging messages.

  Next is the KT1 model in which a node initially knows the ID of its neighbors. Our algorithms are presented in this model.

  Similarly, KTρ can be defined, where each node initially knows the topology up to a radius of ρ hops from itself [AGVP90].

1.3 Efficiency of the Algorithms

In this thesis, we consider the time and the message complexity as the measures for efficiency of an algorithm.

Efficiency in the CONGEST model: Message complexity is the number of messages sent during the execution of the algorithm. Time complexity of a synchronous algorithm is the number of rounds (time units) required for the algorithm to terminate. In the asynchronous model, however, analyzing the time complexity is a bit different. In an asynchronous network, message delays could be finitely long; however, we assume that the delays are normalized and the longest delay is one time unit. Then, the worst-case time complexity happens when the messages are received in the worst possible order, i.e., one that causes the algorithm to require the maximum number of time units.

For example, consider the following pseudocode for the flooding algorithm which is used for broadcasting (from [Pel00]):

Algorithm 1 Flooding Algorithm

1: **procedure** FLOOD(s)
2: The source node s sends the message to all of its neighbors.
3: Any node $t \neq s$ that receives the message for the first time, forwards it to all of its neighbors except the one that sent the message to t.
4: Any node $t \neq s$ that receives the message again, discards it.
5: **end procedure**

The time complexity of this algorithm is $\Theta(D)$ in both synchronous and asynchronous models, where D is the diameter of the network. For the synchronous model, it takes one time step for the message to reach all of the nodes at distance 1 from s, then another time step to reach all of the nodes at distance 2, and so on. Therefore, the time complexity in the synchronous model is $\Theta(D)$.

In the asynchronous model, however, a single message may be forwarded very quickly along a path of length $O(n)$ before any other message is received. But even if that happens, according to the definition, one time step has not yet passed, because the longest message delay accounts for a time unit.

So, when s sends out the message in parallel to all of its neighbors, it takes one asynchronous time step for all nodes at distance 1 to have the message. Once they have it, they do the same and it takes another time step for the message to reach all nodes at distance 2 from s. Therefore, the time complexity is $\Theta(D)$. In both the synchronous and the asynchronous models the message complexity is $\Theta(m)$ since every node sends a message to all of its neighbors.

1.4 Organization

In Chapter 2, we describe the problems and our assumptions. In Chapter 3, we review the algorithm of [KKT15] and present our algorithms to improve its time complexity. Chapter 4 discusses the problems in the asynchronous model of communication. Finally, in Chapter 5, we conclude the discussion and present the open problems.

Chapter 2

Distributed Broadcast and Minimum Spanning Tree

We start by describing our assumptions then we define the problems and motivate them.

2.1 Assumptions

Let the distributed network be an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. We denote $|V|$ and $|E|$ by n and m, respectively. We assume that each node has a unique ID in $\{1, \ldots, n^k\}$, where k is a positive and integer constant. The *edge number* of an edge $\{u, v\}$ is the concatenation of the unique IDs of its endpoints, where the smallest ID is put first. Note that $\{u, v\}$ is an unordered-pair and is used when the order of the endpoints does not matter, otherwise (u, v) or (v, u) (ordered pairs) are used.

A *weighted* network is one that has a weight assignment $w : E \to \{1, \ldots, n^c\}$ for some positive constant c. The weighted graph is referred to as (G, w), and $w(u, v)$ is initially only known to u and v. The weight of a subgraph in (G, w) is the sum of the weights of the edges in that subgraph. For example, the weight of a tree is sum of the weights of the edges in that tree.

We do not assume that the *edge weights* are unique. However, we assume that the algorithms presented in this thesis make the weights unique by concatenating the weight of an edge to the front of its edge number. This ensures that in the case of constructing an MST the solution is unique.

Our algorithms are described in the CONGEST model in which each message has a size of $O(\log n)$. We consider the KT1 model in which nodes initially know the ID of their neighbors. Also, we assume that all the log's are in base 2 unless the base is explicitly mentioned.

2.2 Problems

1. **Spanning Tree or Forest (ST):** The objective is to find a subtree of G that spans the whole network. If G is connected, a spanning tree should have exactly n nodes and $n-1$ edges. A spanning tree protocol should be able to solve the spanning forest problem as well. Therefore, if the network is disconnected, the algorithm should find a spanning tree in each of the connected components. This is a fundamental problem as it provides a way for nodes to broadcast the information with only $O(n)$ messages. A spanning tree is also known as a *broadcast tree*.

2. **Minimum Spanning Tree or Forest (MST):** If the input graph is weighted, the objective is to find the spanning tree with the minimum weight among all spanning trees. If the input graph is disconnected, the protocol should find the minimum spanning forest. As mentioned earlier, we assume that the minimum spanning tree (or forest) is unique.

3. **Breadth First Search Tree:** In this problem, the goal is to find a tree in which the distance from a specific node (known as the source) to all other nodes is equal to the distance of the corresponding shortest path in the original graph. Note that our measure of distance is simply the number of edges on the path. Although finding an exact BFS tree is very useful, it is not always possible to do efficiently. Therefore, we also consider the approximate BFS tree problem, as well. In that case, the distance between the source and the other nodes should be an approximation of the actual shortest path. At times, we refer to an approximate BFS by a *low diameter*[1] *tree*. Note that we will not present algorithms for the BFS problem directly, but knowing its definition will be useful.

[1]Diameter of a graph is the the length of the longest shortest path in the graph.

2.3 Motivation

The problems of the ST and the MST construction are fundamental problems which have been considered by many researchers over the last three decades. However, in most of the research conducted on these problems, the time complexity has been the main focus and the message complexity is $\Omega(m)$. In this thesis, we provide algorithms in the synchronous and the asynchronous models of communication whose message complexity is sublinear in m, i.e. $o(m)$. The results of this thesis provide useful general techniques that can be used to reduce the message complexity of distributed algorithms for similar problems. Moreover, our results will highlight the differences between the synchronous and the asynchronous models of communication, and take one more step towards the understanding the limitations of the asynchronous model.

2.4 Common Algorithms

2.4.1 Borůvka's Algorithm

Borůvka's algorithm (see [NMN01] for translation) is a commonly used MST and ST algorithm invented in 1926. Borůvka's algorithm runs in $O(\log n)$ phases. The idea is to maintain a set of fragments, i.e., subtrees, of the MST and merge them gradually until the final MST is constructed. We assume that each fragment has a specific node called *leader*. The fragment (subtree) is rooted at its leader. The ID of the leader is the fragment ID and all nodes in the same fragment have the same fragment ID.

Initially, each node is a fragment. In a phase, each fragment finds the minimum weight edge leaving the fragment. Such an edge is also called the *minimum outgoing edge* or the *lightest outgoing edge*. Then, fragments are merged using the lightest edges found in this phase. The merges are handled as follows. When fragment A finds the minimum outgoing edge to fragment B, it requests to merge with B. If B accepts the merge, the two fragments A and B along with the minimum outgoing edge are merged into one fragment rooted at B's leader. (To avoid unnecessary complication, the specific rules for accepting and executing a merge will be discussed in the next section.)

The implementation of the Borůvka's algorithm in the synchronous model allows a constant fraction of fragments to merge in each phase. Therefore, $O(\log n)$ phases suffice to reach the final fragment which is the MST. Each phase requires $O(n)$ rounds

since the height of any fragment is at most n. Therefore, the time complexity is $O(n \log n)$.

To find the minimum outgoing edge a *convergecast* algorithm can be used. A convergecast is the opposite of a broadcast since data is collected from the leaves towards the leader of the fragment. The algorithm works as follows:

Convergecast Algorithm

In each fragment, first all leaf nodes find their minimum outgoing edge by testing their incident edges in the ascending order of their weights to see whether an edge is outgoing or not. An edge is outgoing if the endpoints have different fragment IDs. Then, each leaf node sends up its minimum outgoing incident edge to its parent as a possible candidate. Then, each node after receiving the candidates from all of its children, compares them with its own minimum outgoing edge, and sends up the one with minimum weight. Eventually, the leader will find the minimum outgoing edge leaving the fragment.

In each phase, the convergecast algorithm over all fragments requires $O(n)$ time and message complexity, which is $O(n \log n)$ over all phases. Also, since the nodes, in the worst case, test all of their incident edges, it requires $O(m)$ messages. Therefore, Borůvka's algorithm has a time complexity of $O(n \log n)$ and a message complexity of $O(m + n \log n)$.

2.4.2 GHS Algorithm

The next common algorithm is by Gallager Humblet Spira (GHS) [GHS83]. Borůka's algorithm will not perform well in an asynchronous network. The reason is that there is no guarantee that fragments grow together and in fact there will be no phases. In other words, it is possible that one fragment of size $\Theta(n)$ grows faster than the others and repeatedly performs tests, broadcasts, and convergecasts to find minimum outgoing edges. Although by definition this will not affect the time complexity, it can increase the message complexity to $\Theta(n^2)$. This happens when a fragment of size $\Theta(n)$ performs $\Theta(n)$ merges.

Now, we briefly explain the main ideas of the GHS algorithm. Our description here is slightly different from the original paper since we want to use GHS alongside our own algorithms.

In GHS, each fragment is assigned a level which is used to ensure that fragments are growing together. Note that all nodes in the same fragment have the same level.

Initially, each fragment (node) has level 0. Fragments use a modified version of the convergecast algorithm mentioned in the previous section. When a node in fragment A sends a test message over an outgoing edge to a node in fragment B, the message is handled as follows. Let l_A and l_B be the levels of A and B, respectively. If $l_A \leq l_B$, the node in B responds by its fragment ID so the node in A can figure out if this is an outgoing edge. Otherwise ($l_A > l_B$), the response is delayed until $l_B \geq l_A$ is satisfied. This condition prevents the larger fragments to repeatedly spend messages on intra-fragment communications and allows the other fragments to grow as well.

When the minimum outgoing edge is found, a merge request is sent over it. If $l_A < l_B$, B will absorb A. In fact, A's leader will broadcast B's fragment ID and level to all nodes in its subtree and A becomes a part of B. Otherwise, if $l_A = l_B$, and A and B have the same minimum outgoing edge, B absorbs A but all nodes update their level to $l_B + 1$.

In GHS, a fragment only increases its level when it merges with a fragment of equal or higher level, so its size at least doubles with each level increase. Therefore, the maximum level is $\log n$. It can be proved (see [GHS83]) using induction that after $i \cdot \Theta(n)$ time units, all fragments have a level of at least i; therefore, the time complexity is $O(n \log n)$.

Each time a fragment participates in a merge (and increases its level), it has only performed one convergecast. Therefore, over all fragments at a certain level, the number of messages required for the broadcasts and the convergecasts is $O(n)$. So, the message complexity is $O(m + n \log n)$. (Note that m is still present since the nodes are still testing their incident edges.)

2.5 Related Work

Asynchronous Networks: The first breakthrough for computing the MST was by Gallager, Humblet, and Spira [GHS83], who designed an asynchronous algorithm which achieved $O(n \log n)$ time and $O(m + n \log n)$ message complexity. The algorithm was in the CONGEST model. Then, later works gradually improved the time complexity of asynchronous MST computation to linear in n [CT85, Gaf85, Awe87, SB95, FM95, FM04]. The message complexity of the aforementioned papers is $O(m + n \log n)$. Since our focus here is on dense graphs with at least $\Omega(n^{3/2})$

edges, we may omit the $n \log n$ term. Notice that all of these algorithms could take $\Theta(n)$ time for some input. In other words, an MST algorithm with $O(n)$ time and $O(m + n \log n)$ is *existentially* optimal, i.e., if the input graph has a diameter of $\Theta(n)$ no algorithm with better time complexity can be presented. However, in practice, most networks have a diameter much smaller than n. This led the researchers to find algorithms that achieve sublinear time complexity for such graphs. However, the first algorithms with this property were in the synchronous model.

Synchronous Networks: Garay et al. [GKP98] were the first to give a sublinear time $O(D + n^{0.614})$-round MST algorithm, where D is the hop diameter of the network. Kutten and Peleg [KP95] gave an algorithm with $O(D + \sqrt{n} \log^* n)$ time complexity, and Elkin [Elk04] provided an algorithm with time $\tilde{O}(\mu(G, w) + \sqrt{n})$. In his algorithm, $\mu(G, w)$ is the *MST-radius* of the network that for certain graphs can be much smaller than D. In KT0 model, lower bounds of $\tilde{\Omega}(D + \sqrt{n})$ on the time complexity and lower bounds of $\Omega(m)$ on the message complexity have been proven [Elk06, SHK+12, PR00, KPP+15, AGV87]. There are algorithms that match both lower bounds simultaneously up to a polylogarithmic factor [PRS16, Elk17b].

A spanning tree can be constructed by a simple breadth-first search from a single node using m messages. The tightness of this communication bound was a folk theorem, according to Awerbuch, Goldreich, Peleg and Vainish [AGVP90]. For a limited class of algorithms, they showed a lower bound of $\Omega(m)$ messages in the synchronous KT1 network.

However, in 2015, King et al. [KKT15], provided an algorithm in the KT1 model with $\tilde{O}(n)$ time and message complexity, which was the first algorithm that obtained a $o(m)$ message complexity. Their algorithm is randomized and Monte Carlo, i.e., it outputs the solution with high probability. Note that with high probability (w.h.p.) means with a probability of $1 - \frac{1}{n^c}$, where constant c is a parameter of the algorithm.

Ghaffari and Kuhn [GK18] provided an algorithm with $\tilde{O}(D + \sqrt{n})$ round complexity and $\tilde{O}(\min\{m, n^{3/2}\})$ message complexity, in the synchronous network. In other words, they achieved sublinear time in n and sublinear communication in m, simultaneously.

These synchronous algorithms can also be simulated in an asynchronous network using a synchronizer. A synchronizer is basically an algorithm that converts any synchronous algorithm to an asynchronous one, and they have been well studied over the past decades. However, since most sychronizers are designed to be general-purpose, they are usually inferior to asynchronous algorithms that are designed for a specific

problem (see [Awe85, AP90, AKM$^+$93, AK93, KPS97, KPS98, BK07, AKM$^+$07]). In particular, either the superlinear time for initializing the synchronizer or their significant message overhead makes them unusable for our purposes when we are designing our asynchronous ST and MST algorithms.

Chapter 3

Synchronous Algorithms

In this chapter, we will prove the following theorems:

Theorem 1. *The MST can be constructed w.h.p. in* $O(diam(MST)\frac{\log^2 n}{\log\log n})$ *time and* $O(n\frac{\log^2 n}{\log\log n}\log diam(MST))$ *messages, where* $diam(MST)$ *is the diameter of the MST.*

Theorem 2. *The MST can be constructed w.h.p. in* $O(n/\epsilon)$ *time and using* $O((1/\epsilon)n^{1+\epsilon}\log\log n)$ *messages where* $\log\log n/\log n \leq \epsilon < 1$.

Theorem 3. *A spanning tree can be constructed w.h.p. in* $O(n)$ *time and using* $O(n\log n\log\log n)$ *messages.*

The results of this chapter have been published in the International Conference on Distributed Computing and Networking (ICDCN) 2017 [MK17].

3.1 Synchronous ST and MST algorithms

Spanning tree construction was long believed to require $\Omega(n)$ messages [AGVP90]. In 2015, King, Kutten and Thorup presented a Monte Carlo algorithm which broke this communication bound. In particular, they showed that the minimum spanning tree or forest (MST) can be constructed using time and messages $O(n\log^2 n/\log\log n)$, and a spanning tree or forest (ST) can be constructed using time and messages $O(n\log n)$. In the next section, we review the algorithms of [KKT15] since our algorithms directly rely on the routines designed there. All of the algorithms in this chapter are presented in the KT1 model in which nodes have initial knowledge of the IDs of their neighbors. As mentioned before, in the KT0 model (a.k.a. plain network) a lower bound of $\Omega(m)$

exists on the message complexity. However, all of the algorithms in this section break this barrier by utilizing the initial knowledge of neighbors and randomness.

3.1.1 Review of *KKT* and *ST-KKT*

Throughout this chapter, KKT and *ST-KKT* refer to the algorithms in [KKT15] for constructing MST and ST, respectively. The elegance of these algorithms is in using a more message-efficient technique in order to find the outgoing edges. We need the following definitions:

Definitions: For an edge $\{u, v\}$, its edge number is the concatenation of the unique IDs of the edges endpoints, smallest first. For any fragment F, $maxEdgeNum(F)$ and $maxWt(F)$ denote the maximum edge number and edge weight, respectively, of any edge in the tree of F. Here, $[j, k]$ denotes the set of integers $\{j, j+1, \ldots, k\}$, and $[r]$ denotes the set of integers $\{1, 2, \ldots, r\}$..

All of the following algorithms and lemmas are taken from [KKT15]. For complementary details and proofs please see the original paper.

Description of TestOut

TestOut(x) with constant probability of error, returns true if there is an outgoing edge leaving F_x (fragment with leader x), and false otherwise. Note that here 1 corresponds to true and 0 corresponds to false.

$TestOut$ uses an *odd hash function* to test if there is an outgoing edge. An ϵ-odd hash function is a randomly chosen function which is defined as $h : [1, m] \to \{0, 1\}$ such that for any non-empty set $S \subseteq [1, m]$, with probability $\geq \epsilon$, hashes an odd number of members from S to 1. The construction of the function is as follows. We pick an odd multiplier a and threshold t, uniformly at random from $[1, 2^w]$, where $w = O(\log n)$ is the machine's word size in bits. Then, we define $h : [1, 2^w] \to \{0, 1\}$ as $h(x) = 1$ if $(ax \mod 2^w) \leq t$, and 0 otherwise. It can be proved that this function is 1/8-odd [Tho18].

Let $h : [1, maxEdgeNumber] \to \{0, 1\}$ be the odd hash function. Let $E(v)$ denote the edge numbers of v's incident edges. Let $(F, V \setminus F)$ be the set of edges (i.e., the cut) with exactly one endpoint in fragment F. To test the existence of any outgoing edge with constant probability, i.e., *TestOut*(x), each node v in F locally computes $\sum_{e \in E(v)} h(e) \mod 2$. If $E(v) = \emptyset$, the result is 0. These values are then aggregated via a convergecast:

$$\sum_{v \in F} \sum_{e \in E(v)} h(e) \mod 2 = \sum_{e \in (F, V \setminus F)} h(e) \mod 2$$

The reason for the last equality is that a non-outgoing edge will appear twice in the sum and will not contribute to the parity.

To implement this, the leader of F broadcasts h. All nodes compute their local parity and the results are aggregated at the leader via a convergecast.

Similarly, we can implement *TestOut*(x, j, k) which checks if there is an edge leaving F_x whose weight is in the interval $[j, k]$. To do this, nodes should locally only include edges whose weights are in the interval $[j, k]$. So, the sum for local computation at node v changes to

$$\sum_{e \in E(v) \wedge weight(e) \in [j,k]} h(e) \mod 2.$$

The time and message complexity of *TestOut* is $O(|F_x|)$, where $|F_x|$ is the size of the fragment.

High Probability TestOut

HP-TestOut(x) w.h.p. returns true if there is an outgoing edge leaving F_x, and false otherwise. This algorithm has time and message complexity of $O(|F_x|)$.

Let $E^{\uparrow}(u) = \{(u, v) \in E\}$ and $E^{\downarrow}(u) = \{(v, u) \in E\}$. For fragment F, $E^{\uparrow}(F) = \cup_{u \in F} E^{\uparrow}(u)$ and $E^{\downarrow}(F) = \cup_{u \in F} E^{\downarrow}(u)$.

Authors of KKT observe that there is an edge $\{u, v\} \in F$ with only one endpoint in F if and only if $E^{\uparrow}(F) \neq E^{\downarrow}(F)$. So, to test the existence of an outgoing edge it suffices to test whether $E^{\uparrow}(F) \neq E^{\downarrow}(F)$ is true or not.

To test set equality they use a method for polynomial identity testing [BK95]. Let B be the number of edges incident to nodes in F. Let $\epsilon(n)$ be the probability of error. Let $p > \max\{maxEdgeNumber(F), B/\epsilon(n)\}$ be a prime, where $|p| \leq w$. Remember that $w = O(\log n)$ is the maximum message size. For edge set D, define a polynomial over $\mathbb{Z}_p$ by

$$\mathcal{P}(D)(z) = \prod_{e \in D} (z - \text{EdgeNumber(e)}) \mod p.$$

If $E^{\uparrow}(F) \neq E^{\downarrow}(F)$, then $Pr_{\alpha \in \mathbb{Z}_p} \left[\mathcal{P}(E^{\uparrow}(F))(\alpha) = \mathcal{P}(E^{\downarrow}(F))(\alpha)\right] < \epsilon(n)$. *HP-TestOut*$(x)$ assumes that x knows $\epsilon(n)$ and is implemented as follows:

1. x broadcasts $\alpha \in \mathbb{Z}_p$ and p.

2. Each node y locally computes $Local^{\uparrow}(y) = \mathcal{P}(E^{\uparrow}(y))(\alpha)$ and $Local^{\downarrow}(y)(\alpha) = \mathcal{P}(E^{\downarrow}(y))(\alpha)$. When y receives $\mathcal{P}({F_z}^{\uparrow}(y))(\alpha)$ and $\mathcal{P}({F_z}^{\downarrow}(y))(\alpha)$ (F_z is the subtree rooted at node z) from its children z, it sends to its parent:

$$\mathcal{P}(E^{\uparrow}(T_y))(\alpha) = Local^{\uparrow}(y) \times \prod_{z \text{ child of } y} \mathcal{P}(E^{\uparrow}(F_z))(\alpha)$$

$$\mathcal{P}(E^{\downarrow}(T_y))(\alpha) = Local^{\downarrow}(y) \times \prod_{z \text{ child of } y} \mathcal{P}(E^{\downarrow}(F_z))(\alpha)$$

3. x determines the existence of an outgoing edge by checking $\mathcal{P}(E^{\uparrow}(F))(\alpha) \neq \mathcal{P}(E^{\downarrow}(F))(\alpha)$.

Similarly, *HP-TestOut*(x, j, k) can be implemented with the same time and message complexity by only considering edges whose weights are in the interval $[j, k]$.

Finding the Minimum Outgoing Edge

In *TestOut*, the result is only 1 or 0. So, a single bit suffices to return the result in the convergecast. Therefore, to find the minimum outgoing edge, KKT uses parallel *TestOut*'s (i.e. $O(\log n)$) to narrow down the search range more quickly. The first interval whose *TestOut* result is 1, becomes the range for the next search. Also, after each convergecast the result is verified using *HP-TestOut*. Below is the description of the algorithms *FindMin* and *FindMin-C*. The only difference is that *FindMin* repeats the loop for $O(\log n)$ iterations while *FindMin-C* continues for only the expected number of iterations, i.e., $O(\log n / \log \log n)$ (see [KKT15]).

Recall that in the following algorithm $w = O(\log n)$ is the word size. In other words, each message contains w bits. w is utilized to speed up the search for the outgoing edge.

$FindMin(x)$ [*FindMin-C*] finds the lightest edge in $(F_x, V \setminus F_x)$

1. $Count \leftarrow 0$.

2. x determines $maxWt(F_x)$ and $maxEdgeNumber(F_x)$ through one broadcast followed by a convergecast. Then, x computes $\epsilon(n)$.

3. x broadcasts an odd hash function $f : [1, maxEdgeNum(F_x)] \to \{0, 1\}$ and also j and k.

4. In parallel for $i = 0, 1, 2, \ldots w - 1$:
   Set $j_i \leftarrow j + i\lceil (k - j + 1)/w \rceil$; $k_i \leftarrow j + (i + 1)\lceil (k - j + 1)/w \rceil - 1$, and nodes in a convergecast return a word in which i^{th} bit is the result of $TestOut(x, j_i, k_i)$.

5. Upon receiving the result, x determines the index $min = \min\{i | TestOut(x, j_i, k_i) = 1\}$ and initiates $TestLow =$ *HP-TestOut*$(x, 0, j_{min} - 1)$ and $TestHigh =$ *HP-TestOut*(x, j_{min}, k_{min}).

6. Then,

   (a) If $TestLow = 0$ and $TestHigh = 1$ and if $j_{min} < k_{min}$, x sets $j = j_{min}$ and $k = k_{min}$; else, if $j_{min} = k_{min}$, then x broadcasts "stop" and returns j_{min}.

   (b) Else, if both return 0, x broadcasts "stop" and returns $\emptyset$.

   If none of (a) or (b) are satisfied, go to Step 7.

7. For $FindMin(x)$ [resp., *FindMin-C*]:
   If $Count < (c/q) \log n + (c/q) \log maxWt(F_x)/\log(w - 1)$, [resp., $Count < (2c/q) \log maxWt(F_x)/\log(w - 1)$], increment Count and repeat from Step 4. Else return $\emptyset$.

The following lemma is proven in [KKT15]:

Lemma 1 (FindMin, FindMin-C)**.** *Let c be any constant s.t. $c \geq 1$. With probability $1 - 1/n^c$, using asynchronous communication, FindMin(x) returns the lightest edge leaving a fragment F_x in expected time and messages $O(|F_x| \log n/\log\log n)$ (and worst case $O(\log n)$ time and messages). With probability $2/3 - 1/n^c$, FindMin-C(x) returns the lightest edge and with probability $1 - 1/n^c$ it returns the lightest edge or $\emptyset$ using worst case $O(|F_x| \log n/\log\log n)$ messages and time. If there is no edge leaving the tree, both procedures always return $\emptyset$. This assumes x knows a polynomial upper bound on n.*

Constructing the MST

The algorithm for constructing the MST follows from the implementation of the Borůvka's algorithm (Section 2.4.1) and using $FindMin$ to find the lightest outgoing

edges. We refer to this algorithm by *KKT*. Let $maxTimeMST(n)$ be the maximum time required to complete Steps (a) and (b) in a fragment of size n. Note that *time* in the algorithm refers to the value of the global clock.

KKT is executed by every node x and constructs the minimum spanning tree or forest:

1. $time \leftarrow 0$.
2. For $i = 1$ to $O(\log n)$:
   (a) If x is the leader, it initiates *FindMin-C*; else, x participates in *FindMin-C*.
   (b) If x is an endpoint of the edge $\{x, y\}$ which has been returned by *FindMin-C*, x sends $\langle AddEdge \rangle$ message to y across $\{x, y\}$.
3. While $time < i \cdot maxTimeMST(n)$ wait; while waiting, if any $\langle AddEdge \rangle$ message is received over an edge, mark that edge.

Since *FindMin-C* on disjoint fragments requires $O(n \log n / \log \log n)$ time and messages, KKT can construct the MST w.h.p. and in $O(n \log^2 n / \log \log n)$ time and messages.

Find Any Outgoing Edge

$FindAny(x)$ finds some edge leaving fragment F_x using an expected constant number of broadcasts and convergecasts. Therefore, it does not have the $\log n / \log \log n$ factor in $FindMin$. Let $\epsilon(n)$ be the error function and $\epsilon(n) < 1/(2n^c)$. Assume that $\epsilon^{-1}(n)$ is polynomial in n. Below is the description of $FindAny$ and *FindAny-C*. *FindAny-C* achieves the same goal but with constant probability.

$FindAny(x)$ [resp. *FindAny-C*] finds some outgoing in fragment F with leader x if any such edge exists:

1. $Count \leftarrow 0$.
2. x initiates *HP-TestOut* in F using $\epsilon(n)$ for error. If the result is false, then return $\emptyset$.
3. Determine the identity of an edge as follows:

(a) x broadcasts a random pairwise independent hash function $h : [1, maxEdgeNum(F)] \rightarrow [r]$ where r is a power of $2 >$ sum of degrees of nodes in F.

(b) Each node y hashes the edge numbers of its incident edges using h, and computes the vector $\overrightarrow{h}(y)$ s.t. $h_i(y)$ is the parity of the set of incident edges whose edge numbers hash to values in $[2^i]$ for $i = 1, \ldots, \log r$. If y has no incident edges then $\overrightarrow{h}(y) = \overrightarrow{0}$.

(c) The vector $\overrightarrow{h}(F) = \bigoplus_{y \in F} \overrightarrow{h}(y)$ is computed up the tree, in the convergecast and x aggregates the result. Then, x broadcasts $min = \min\{i | h_i(F) = 1\}$.

(d) Let $E(x)$ be the set of edge numbers of edges incident to x. Each node x computes $w(x) = \bigoplus\{e | e \in E(x) \wedge h(e) < 2^{min}\}$.

4. Test: x broadcasts $w(F)$ to obtain $Sum =$ the number of endpoints in F incident to the edge given by $w(F)$. Test succeeds iff $Sum = 1$.

5. If Test succeeds, return $w(T)$ else for *TestOut-C*, return $\emptyset$; for $TestOut$, if $Count \geq 16 \ln(\epsilon^{-1}(n))$ then return $\emptyset$ else increment $Count$ and repeat Steps 3-5.

We have the following lemma from [KKT15].

Lemma 2 (*FindAny*, *FindAny-C*)**.** *If there is no edge leaving F, then $FindAny(x)$ and FindAny-C(x) returns $\emptyset$. Otherwise,*

- *$FindAny(x)$ returns an edge leaving F w.h.p. It uses expected time and messages $O(n)$; and*
- *FindAny-C(x) returns and edge leaving F with probability at least $1/16$, else it returns $\emptyset$. It uses worst case time and messages $O(n)$.*

Constructing an ST

To construct an ST, the authors provide a modified version of KKT which we refer to by *ST-KKT*. First, they replace *FindMin-C* with *FindAny-C*. Then, they use three routines to "break" the possible cycles before the next phase starts. Note that this problem did not exist in the case of MST since we assumed that all edge weights are unique. The following routines are performed at the end of each phase:

Routines to handle cycles:

1. *Cycle detection:* Each leaf sends a message to its only neighbor; each node after receiving a message from all but one of its neighbors, sends a message to the neighbor not yet heard from. After time sufficient to hear from all but one neighbor in a worst-case tree, if a node has not heard from two neighboring nodes it has detected that it lies on a cycle and edges to those neighbors are edges on the cycle.

2. *Cycle breaking:* If a node is on a cycle and one or two of its neighboring edges on a cycle are newly marked, for each such edge, it sends a fair coin flip to the other endpoint of the edge. If both endpoints of the edge toss heads, then the edge is unmarked.

3. *Check and fix if necessary:* The cycle detection algorithm is again run to test if there is a cycle. If there still is a cycle, all the newly marked edges in the cycle are unmarked and not included as tree edges in the next phase.

There is an argument in [KKT15] that using this technique will not affect the asymptotic complexity of the algorithm. We restate that argument since we will refer to it in the next section.

Claim 1. *Let F be the number of fragments at the start of a phase. Let C be the probability that FindAny-C returns an edge. At the end of the phase there are no more than $(1 - C/8)F$ fragments with probability at least $\alpha = (1 - C/4)/(1 - C/8)$.*

Proof. Any edge returned by *FindAny-C* which is not in a cycle formed by edges chosen in a phase by *FindAny-C*, reduces the number of fragments by 1. Any such edge in a cycle reduces the number of fragments by 1 if its cycle is broken and it is not unmarked by both its endpoints. The probability that a newly marked edge e is unmarked by both its endpoints is $1/4$. Every cycle that is formed must have at least two newly marked edges, thus the probability that the cycle containing e is not broken is no greater than $1/2$. The probability that e is unmarked because of either of these events is thus no greater than $3/4$, by a union bound. It follows that the probability that a fragment finds an edge leaving using *FindAny-C* and that edge reduces the number of fragments by 1 is at least C/4. Let F' be the number of fragments at the end of the phase. Then $E[F'] \leq (1 - C/4)F$. By Markov's Inequality,

$$Pr[F' > (1 - C/8)F] \leq E[F']/(1 - C/8)F \leq (1 - C/4)/(1 - C/8).$$

□

ST construction succeeds w.h.p. and takes $O(n \log n)$ time and messages [KKT15].

3.2 Algorithm for MSTs with Small Diameter

In this section, we prove Theorem 1:

Theorem 1. *The MST can be constructed w.h.p. in* $O(diam(MST)\frac{\log^2 n}{\log\log n})$ *time and* $O(n\frac{\log^2 n}{\log\log n}\log diam(MST))$ *messages, where* $diam(MST)$ *is the diameter of the MST.*

We provide a very simple algorithm using a binary search approach that is tailored towards the cases when the MST has a low diameter. Let $diam(MST)$ be the diameter of the minimum spanning tree. In the case of a disconnected network, $diam(MST)$ is the diameter of the connected component with maximum diameter. If $diam(MST)$ is asymptotically less than n, this algorithm achieves time complexity $o(n)$ and message complexity $o(m)$. This is the first algorithm that constructs an MST in time proportional to the diameter of the MST up to a logarithmic factor with $o(m)$ communication.

Our algorithm is a modification of KKT and takes $O(diam(MST)\frac{\log^2 n}{\log\log n})$ time and $o(m)$ messages. Our algorithms in this chapter are Monte Carlo and output a solution w.h.p. If the algorithm fails no solution is generated.

Each iteration of KKT allows the fragments $O(n)$ rounds to look for outgoing edges. However, we know that over the course of algorithm each fragment is actually a part of the final MST. This implies that if the diameter of the MST is low then the diameter of any fragment is low as well. In $O(diam(MST))$ time, fragments can find the lightest outgoing edges. However, we do not have prior knowledge of $diam(MST)$; therefore, we guess. In particular, we start from a constant estimate, and *simulate* a version of KKT assuming our estimation is an upper bound on diameter. We find the MST as soon as our estimation $\hat{D}$ for $diam(MST)$ becomes greater than $diam(MST)$. When we estimate $\hat{D}$, we prevent fragments from spending time more than $O(\hat{D}\log n/\log\log n)$.

Let *active fragments* be those allowed to look for the minimum outgoing edge, i.e., those with height less than or equal to $\hat{D}$. The algorithm is as follows.

The algorithm runs in iterations. Each iteration uses a threshold $\hat{D}$ for height. At the start of each iteration, all leaders deactivate themselves. Every leaf makes a *timer message* and sends it upwards. A *timer* is a message with the initial value equal to $\hat{D}$. Each time it passes a link its value is decreased by one, and once the value hits zero it will stop being transmitted. A timer message is very similar to the *exploration*

token used by Awerbuch in [Awe87]; however, here, a timer does not try to measure size of a fragment; it only cares about the height of a fragment which is the deciding factor for the time complexity of our algorithm.

Every internal node waits to receive the timers from all its children. Then, it picks the timer with minimum value, decrements it, and sends it to its parent. As a result, the leader of each fragment will receive the timers of all of its children iff the height of the fragment is $\leq \hat{D}$. In this case, the leader (and its fragment) will be activated. It can then start to look for the minimum outgoing edges. Algorithm 2 shows a pseudo-code of this activation process.

Algorithm 2 Activation

1: **procedure** ACTIVATE(T) `//Takes the value of threshold as input`
2: All of the leaves send up a timer with value T.
3: Every internal node that received timers from all of its children picks the minimum timer, decrements its value, and sends it up if the value is not zero.
4: Every leader that receives the timers from all of its children is activated.
5: **end procedure**

Diam-KKT (Algorithm 3) finds the MST. The algorithm starts by setting the estimation of $diam(MST)$, i.e. $\hat{D}$, to 1. Afterwards, before running an iteration of KKT it first activates only fragments whose height is $\leq \hat{D}$. Intuitively, we are assuming that $\hat{D}$ is the actual value of $diam(MST)$; hence, there is no point in allowing fragments with height larger than $\hat{D}$ to look for outgoing edges. Knowing this, if we were proved wrong and could not find the MST we will make a higher estimation by doubling $\hat{D}$. In case of a minimum spanning forest, it could be that some fragments are maximal while some others still need to increase their estimate. After performing enough iterations, i.e., $O(\log n)$, we need to see which fragments are maximal. To do this, we can test w.h.p. the existence of an outgoing edge using *HP-TestOut*. If *HP-TestOut* is false, the fragment is maximal. As soon as our estimate is good enough, i.e. $diam(MST) \leq \hat{D} < 2 \cdot diam(MST)$, w.h.p. all MST fragments will be maximal. When the minimum spanning tree or forest is found, all nodes in the network receive the message STOP and the algorithm successfully terminates.

Proof of Theorem 1. We prove that *Diam-KKT* uses $O(diam(MST)\frac{\log^2 n}{\log\log n})$ time and $O(n\frac{\log^2 n}{\log\log n} \cdot \log diam(MST))$ messages. W.h.p., the algorithm finishes as soon as $\hat{D} \geq diam(MST)$. Thus, we have a total of $\log diam(MST)$ estimations since each time we are doubling $\hat{D}$. Since each time we are doubling $\hat{D}$, the sum is a

geometric sequence and is dependent on the last term which is, w.h.p., at most twice the actual value of $diam(MST)$. Besides, we have $O(\log n)$ iterations, and a multiplicative factor of $O(\frac{\log n}{\log\log n})$ due to *FindMin-C*. Hence, the overall time complexity is $O(diam(MST)\frac{\log^2 n}{\log\log n})$. The message complexity will only increase by a factor of $\log diam(MST)$ and the theorem follows.

Algorithm 3 Modified KKT for MSTs with small diameter

1: **procedure** DIAM-KKT
2: Set $\hat{D} = 1$. `//`$\hat{D}$ `is the estimation of the MST's diameter.`
3: **while** $\hat{D} \leq 2n$ **do**
4: **for** $i = 1$ to $i < c \log n$ **do** `//`c `is a sufficiently large constant.`
5: All leaf nodes initiate ACTIVATE($\hat{D}$).
6: Active fragments use *FindMin-C*.
7: **end for**
8: Leaves initiate ACTIVATE($\hat{D}$). `//Start of verification`
9: Every active fragment uses *HP-TestOut* to determine the existence of an outgoing edge.
10: Every active fragment whose *HP-TestOut* result is false, is a maximal fragment of the MST. So, its leader broadcasts a STOP message.
11: If nodes did not receive a STOP message after sufficient time, they continue at Step 4 with $\hat{D} = 2\hat{D}$.
12: **end while**
13: **end procedure**

3.3 Construction of the MST in Linear Time

DiamKKT is a very fast algorithm if the diameter of the MST is small enough. However, if $diam(MST) = \omega(\frac{n \log\log n}{\log^2 n})$ then it will take $\omega(n)$ time. Here, we want to present an algorithm for MST construction which w.h.p. works in $O(n)$ time and and $o(m)$ messages, even if the MST has a large diameter. In particular, we prove Theorem 2:

Theorem 2. *The MST can be constructed w.h.p. in $O(n/\epsilon)$ time and using $O((1/\epsilon)n^{1+\epsilon}\log\log n)$ messages for all values of ϵ in $[\log\log n/\log n, 1]$.*

Let T be a threshold on the height which determines what fragments can be active. We show how to initialize and gradually increase this threshold to find the MST.

In fact, we want to get rid of the $\log_{\log n} n$ (or $\frac{\log n}{\log \log n}$) that appears in *FindMin-C* due to a $\log n$-ary search. To speed up the process, we narrow down the search range by a factor of n^ϵ each time, where $n^\epsilon > \log n$.

Similar to KKT, for the i^{th} interval ($[j_i, k_i]$) of the current search range we need one bit to be the result of $TestOut(x, j_i, k_i)$. Then, the leader will pick the first part that has an outgoing edge, i.e., the first interval whose $TestOut$ result is true, and again this new interval will be divided by n^ϵ.

To implement this, each node needs an array of n^ϵ bits. Sending an array with this magnitude will need $\frac{n^\epsilon}{\log n}$ consecutive messages. Therefore, we use a *pipelining technique* that works as follows. Every leaf sends these messages one after another without delay, and every internal node upon receiving the array from all its children calculates the sum of those arrays and sends it up. In fact, internal nodes will receive the parts of their children's arrays in order; therefore, they can compute the sum for the parts they have received (from all children) and send it up without waiting for the whole array to arrive.

It is easy to verify that the time rounds needed for the leader to receive the whole array from all its children is at most $T + \frac{n^\epsilon}{\log n} \leq 2T$.

Now we show how this pipelining technique helps in getting high probability in a single iteration. The idea is that instead of using only one hash function we use $O(\log n)$ pipelined hash functions. But again, we do not wait for the results of the first hash function before repeating the process with another one. We simultaneously apply $O(\log n)$ randomly chosen hash functions. The leader broadcasts all of the $O(\log n)$ hash functions in a pipelined manner. This $O(\log n)$ extra messages through each link will not affect the time complexity because the extra $O(\log n)$ time will be additive to T, which is relatively very large. This, however, affects the message complexity by a factor of $O(\log n)$. $FastFindMin$ (Algorithm 4) shows how to find the lightest outgoing edge in $O(T/\epsilon)$ time.

We also modify $TestOut$ to take a fourth argument h_l which is the hash function that it will use. Later we show that when we are in the first phase of the algorithm we can have $O(\log n)$ iterations. Therefore, we will get high probability with applying only one hash function in *FastFindMin-C*. We do not give the pseudocode separately for *FastFindMin-C*. It is $FastFindMin$ but with one hash function instead of $O(\log n)$.

Lemma 3. *There exists a constant c for which $FastFindMin$ finds the lightest outgoing edge w.h.p.*

Algorithm 4 Faster version of FindMin

procedure FastFindMin(x, ϵ) `//Takes fragment leader` x `and` ϵ `as input.`
  Initialize the search interval to $[1, n^k]$.
  while size of the range is more than 1 **do**
    x broadcasts odd hash functions $h_1, h_2, \ldots, h_{c \log n}$ where $h_l : [1, n^k] \rightarrow \{0, 1\}$.
    Let $[j_i, k_i]$ be the i^{th} part of the current range. For $i = 1$ to n^ϵ and for $l = 1$ to $c \log n$ calculate $TestOut(x, j_i, k_i, h_l)$ in the fragment leader by pipelining.
    x determines, w.h.p., the first subinterval with an outgoing edge. This is done by picking the first interval $[j_{min}, k_{min}]$ for which there exists some hash function h_m such that $TestOut(x, j_{min}, k_{min}, h_m) = true$.
    Update the range to j_{min}, k_{min}.
  end while
  Return the minimum outgoing edge.
end procedure

Proof. Imagine I is the current interval under search. Let I_f be the first subinterval that has an outgoing edge. Since, as stated in [KKT15], each h_i is an odd hash function, the probability that an odd number of outgoing edges in I_f hash into 1 is at least 1/8. Note that non-outgoing edges will cancel each other's parity since they appear exactly twice in the sum. Therefore, the probability of this event not happening is less than 7/8 for one hash function. Using $c \log n$ hash functions this probability is reduced to $(\frac{7}{8})^{c \log n}$. However, we need to narrow down the interval a total of k/ϵ times, and the whole process fails if any of these narrowing downs fails. Hence, by union bound, the probability of not finding the lightest outgoing edge will be $\frac{k}{\epsilon} \cdot (\frac{7}{8})^{c \log n}$. Now, note that we always want $n^\epsilon = \Omega(\log n)$, because otherwise the whole array can be transmitted in one message; this implies that $\epsilon = \Omega(\frac{\log \log n}{\log n})$. Thus, for $\frac{k}{\epsilon} \cdot (\frac{7}{8})^{c \log n}$ to be less than $1/n^{c_1}$ it suffices that $(\frac{7}{8})^{c \log n} < \frac{1}{n^{c_1+3}}$ which happens for $c > \frac{c_1+3}{\log \frac{8}{7}}$. □

Having all the prerequisites, we can provide the $FastMST$ algorithm which w.h.p. finds the MST in $O(n/\epsilon)$ rounds and with $O(\frac{n^{1+\epsilon} \log \log n}{\epsilon})$ communication. $FastMST$ starts with setting the threshold value to $\frac{n}{\log^2 n}$. Before executing an iteration for finding outgoing edges the procedure *Activate* is initiated. For every value of threshold (say $\hat{D} = \frac{n}{(\log^{(i)} n)^2}$), $O(\log^{(i)} n)$ iterations are performed. The next lemma proves the time complexity.

Lemma 4. *Algorithm FastMST terminates in $O(n/\epsilon)$ rounds w.h.p.*

Algorithm 5 Synchronous MST algorithm with $O(n)$ time

```
1: procedure FastMST(ϵ) //Takes ϵ as input.
2:     Initialize F, set of all fragments, to be all of the singletons.
3:     Set threshold counter i = 1.
4:     while i ≤ log* n do
5:         Set the threshold T = n/(log^(i) n)^2.
6:         Set the iteration counter j = 1.
7:         while j ≤ 2⌈log^(i) n⌉ do
8:             Leaves initiate activate(T).
9:             if i = 1 then
10:                Call FastFindMin-C(x, ϵ) for every active leader x.
11:            else
12:                Call FastFindMin(x, ϵ) for every active leader x.
13:            end if
14:            Merge fragments using the lightest outgoing edges found.
15:            j ← j + 1.
16:        end while
17:        i ← i + 1.
18:    end while
19: end procedure
```

Proof. We know from Lemma 3 that *FastFindMin* fails probability $1/n^{c_1}$ for some constant c_1. Now, in any iteration, there are at most n fragments. Therefore, by union bound, w.h.p., the number of fragments is divided by at least 2 in each iteration. Hence, the number of iterations needed for each phase is log of the number of fragments. Furthermore, there are a total of $\sum_{i=1}^{i=\log^* n} O(\log^{(i)} n) = O(\log n)$ iterations and they all succeed w.h.p., again by union bound. The maximum edge weight is n^k for some constant k and since each time we are narrowing down the range by n^ϵ, a total of $\frac{k}{\epsilon}$ narrowing downs are needed; hence the factor $\frac{1}{\epsilon}$ in the time complexity.

When all of the fragments with height $\leq n/(\log^{(i)} n)^2$ are merged, the number of remaining fragments cannot exceed $(\log^{(i)} n)^2$. Therefore, for each phase where the threshold is updated from $n/(\log^{(i-1)} n)^2$ to $n/(\log^{(i)} n)^2$, we only need $O(\log(\log^{(i-1)} n)^2) = O(\log^{(i)} n)$ iterations. Moreover, in the last phase the threshold is exactly n and all of the remaining fragments will merge.

In each iteration, only active fragments are allowed to look for outgoing edges.

Thus, the overall complexity will be

$$\frac{1}{\epsilon}\sum_{i=1}^{i=\log^* n} O(\log^{(i)} n)\cdot O(\frac{n}{(\log^{(i)} n)^2}) = \frac{n}{\epsilon}\sum_{i=1}^{i=\log^* n} O(\frac{1}{\log^{(i)} n}).$$

The last term of this sum is a constant. For large enough value of n, each time the denominator is losing a log; so, we can say the denominator is at least doubled each time. Therefore, this sum can be bounded by a constant using geometric series, and the lemma follows. □

Lemma 5. *Algorithm FastMST requires* $O(\frac{n^{1+\epsilon}\log\log n}{\epsilon})$ *messages.*

Proof. As stated in Lemma 4, it takes $\frac{k}{\epsilon}$ narrowing downs before the lightest outgoing edge is found. For each of these narrowing downs every node needs to communicate an n^ϵ-bit array upward for each of the $O(\log n)$ hash functions. This is $O(n^\epsilon \log n)$ bits which can be pipelined in time $O(n^\epsilon)$. In total, every link will be used for $\frac{k}{\epsilon}\cdot n^\epsilon$ messages after the first phase. Besides, there will be $O(\log\log n)$ iterations over all phases $i \geq 2$. On the other hand, in the first phase, we use only one hash function but we have $O(\log n)$ iterations. Thus, considering k is constant, the total message complexity will be:

$$O(\frac{1}{\epsilon}\frac{n^{1+\epsilon}}{\log n}\log n) + O(\frac{1}{\epsilon}\frac{n^{1+\epsilon}}{\log n}\log n\log\log n) = O(\frac{n^{1+\epsilon}}{\epsilon}\log\log n).$$

□

Together, Lemma 4 and Lemma 5 prove Theorem 2.

3.4 Construction of an ST in Linear Time

The MST algorithm in the previous section trivially obtains a spanning tree, as well. However, the objective of this section is to significantly reduce the message complexity. We prove Theorem 3:

Theorem 3. *A spanning tree can be constructed w.h.p. in* $O(n)$ *time and using* $O(n\log n\log\log n)$ *messages.*

For constructing a spanning tree in linear time we again use a threshold T to only allow fragments with height less than or equal to T to look for outgoing edges. As in

the previous section we need to boost the probability when we cannot have $O(\log n)$ iterations. This will be done by running repetitions of each iteration in parallel.

The algorithm *FastST* is shown for finding the spanning tree in $O(n)$ rounds and using $O(n \log n \log\log n)$ messages. We do not give the pseudocode for *LogFindAny*. It does the same thing as *FindAny-C* except it uses $O(\log n)$ hash functions with the same pipelining technique to get high probability of success in finding outgoing edges.

Proof of Theorem 3: Over the course of the algorithm, we use $\log^* n$ phases and in each phase we have the threshold $T = n/(\log^{(i)} n)^2$. In any iteration *FindAny-C* and *LogFindAny* spend time proportional to the height of the fragment which is bounded by T. Note that using $O(\log n)$ hash functions in a pipelined manner in *LogFindAny* does not affect time complexity. Therefore, similar to the analysis of Algorithm *FastMST*, the time complexity is $O(n)$. Moreover, we are using $O(\log n)$ hash functions only after increasing the threshold for the first time. As in Lemma 5, the overall number of iterations after the phase is $O(\log\log n)$. Hence, the message complexity is $O(n \log n)$ for the first phase, and $O(n \log n \log\log n)$ for the rest of the phases which is $O(n \log n \log\log n)$ overall.

Algorithm 6 Synchronous ST with $O(n)$ time

```
1:  procedure FastST
2:      Initialize F, set of all fragments, to be all of singletons.
3:      Set the threshold counter i = 1.
4:      while i ≤ log* n do
5:          Set the threshold T = n/(log^(i) n)^2.
6:          Set the phase counter j = 1.
7:          while j ≤ 2⌈log^(i) n⌉ do
8:              Leaves initiate activate(T)
9:              if i = 1 then
10:                 Call FindAny-C(x) for every active fragment leader x.
11:             else
12:                 Call LogFindAny(x) for every active fragment leader x.
13:             end if
14:             Handle cycles.
15:             j ← j + 1.
16:         end while
17:         i ← i + 1.
18:     end while
19: end procedure
```

In order to handle cycles we use a similar strategy to the one in *ST-KKT*. However,

the difference is that we only allow nodes to spend T time units to hear from their neighbors where T is the current threshold in the algorithm. Therefore, when a node does not receive a message over some edge e, it means that either the node is on a cycle or simply its distance from a leaf node is more than T. But we prove that in either case using the three routines for handling cycles in Section 3.1.1, the number of fragments reduce by a constant factor in each phase and the asymptotic complexity will not be affected by this modification.

Claim 2. *Let F be the number of fragments at the start of a phase. Let P be the probability that FindAny (or its variations) returns an edge. At the end of the phase there are no more than $(1 - P/8)F$ fragments with probability at least $\alpha = (1 - P/4)/(1 - P/8)$.*

Proof. Let e be on a newly marked edge found by $FindAny$ such that both endpoints of e have detected e as an edge which is either on a cycle or a path longer than T edges from a leaf node. If e is on a cycle, with a similar argument to that of Claim 1 the statement follows. But let us assume that a number of fragments have been connected together as a chain using the newly marked edges $e_1, e_2, \ldots, e = e_i$. If both endpoints of e decide to unmark it in the *cycle breaking* routine, it will be unmarked with probability of $1/2$. Otherwise, e will only be unmarked if after the cycle breaking step, it is still on a long path ($> T$) to a leaf node. But this happens only if $e = e_i$ and e_{i-1} both survive the cycle breaking step which happens with probability of $1/4$. So, the probability that e is unmarked is no more than $3/4$ by union bound. The rest of the argument is exactly the same as Claim 1.

□

Chapter 4

Asynchronous Algorithms

In this chapter, we prove the following theorems for the asynchronous model of communication:

Theorem 4. *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 $O(n^{3/2}\log^{3/2} n)$ time and messages in the KT1 CONGEST model, with high probability.*

The next theorem is an improvement to the time of theorem 4.

Theorem 5. *There exists an asynchronous algorithm in the KT1 CONGEST model that, w.h.p., computes the MST in $O(n)$ time and with $O(\min\{m, n^{3/2}\log^2 n\})$ messages.*

This result achieves sublinear communication, i.e., $o(m)$, and is optimal for time when the diameter is $\Theta(n)$. We also prove the following more general theorem.

Theorem 6. *Given an asynchronous MST algorithm with time $T(n,m)$ and message complexity of $M(n,m)$ in the KT1 CONGEST model, w.h.p., we can construct the MST with $O(n^{1-2\epsilon} + T(n, n^{3/2+\epsilon}))$ time and $\tilde{O}(n^{3/2+\epsilon} + M(n, n^{3/2+\epsilon}))$ messages, for $\epsilon \in [0, 1/4]$.*

Theorem 4 has been published in International Symposium on DIStributed Computing (DISC) 2018 [MK18]. Theorems 5 and 6 appeared in DISC 2019 [MK19a]; as a brief announcement. A full version of this paper can be found in [MK19b].

4.1 Asynchronous MST with $o(m)$ Messages

We provide the first asynchronous distributed algorithms in the KT1 model (initial knowledge of neighbors' IDs) to compute broadcast and minimum spanning tree with $o(m)$ bits of communication, in a sufficiently dense graph. Our algorithm is randomized Monte Carlo. Again note that with a small probability the algorithm will wait in a infinite loop and never terminate. However, if it does terminate, it will generate the correct output with high probability.

As a first step, we provide an algorithm for computing a spanning tree with $O(n^{3/2}\log^{3/2} n)$ messages. Given a spanning tree, we compute the MST with only $\tilde{O}(n)$ messages. Our results in this chapter imply that even with asynchronous communication, if we initially know the ID of neighbors and can use randomness, we can break the $\Omega(m)$ barrier of message complexity.

While *FindAny* and *FindMin* are asynchronous procedures, the Borůvka approach of [KKT15] 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. If a fragment grows fast and repeatedly merges with other (smaller) fragments, the result will be $\Theta(n^2)$ 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 *level* 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 level. The GHS algorithm subtly avoids traversing the whole tree until a minimum weight outgoing edge to an appropriately leveled tree is found. However, this method seems to require communication over all edges in the worst case.

Our algorithms in this chapter separate nodes based on their degree using a threshold. Nodes are classified either as low-degree or high-degree. Asynchrony precludes approaches that can be used in the synchronous model. For example, in the synchronous model, if low-degree nodes send messages to all their neighbors in one round, then all nodes learn which of their neighbors are not low-degree, and therefore they can construct the subgraph of high-degree nodes. In the asynchronous model, a node, not hearing from its neighbor, does not know when to conclude that its neighbor is high-degree.

The technique for building a spanning tree in our work is very different from the

technique in [KKT15] or [GHS83]. 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*. This technique is inspired from [Elk17a], 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. From now on, we call nodes with degree $\geq \sqrt{n} \log^{3/2} n$ *high-degree* nodes, and all other nodes *low-degree*.

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 as well. In addition, when low-degree and star nodes join T, they invite all of their neighbors to join T. 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. Each high-degree node in T is adjacent to a star node in T.

To be more accurate, this invariant is not initially true. We design a subroutine named *Expand* (described later), and after each execution of *Expand* the invariant is satisfied with high probability.

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. Since 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 is 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 we will discuss later on.

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

Note: For now, we assume that the graph is connected and we deal with disconnected case in Section 4.5. The original method that we used to handle the disconnected case in [MK18] is a bit complex; therefore, we decided to discuss an easier approach that appears in [MK19b].

4.2 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). For clarity, throughout this chapter, we use $\langle M \rangle$ to denote a message with content M. 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.

- $ApproxCut$: 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.

- $Found_L(v)$, $Found_O(v)$: 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 $\langle$*Low-degree*$\rangle$ message or the edge is found by the leader by sampling and its external endpoint is low-degree. Otherwise, an edge is added to $Found_O(v)$, i.e., when v receives a $\langle$*Star*$\rangle$ 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 $\langle$*Low-degree*$\rangle$ and $\langle$*Star*$\rangle$. This may cause an edge to be in both lists which will be 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 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 $\langle$*Low-degree*$\rangle$ 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). This procedure requires $O(|T| \log n)$ messages.

The implementation of *ApproxCut* is given in Algorithm 9. In Section 4.3.2 we will later prove the following lemma about *ApproxCut*:

Lemma 6 (*ApproxCut*). *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.*

4.3 Spanning Tree 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. 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.

FindST-Leader (Algorithm 10) runs in phases. Each phase has three parts:

1. **Expansion** of T over found edges since the previous phase (or the start of the algorithm if it is the first phase) and the restoration (or establishment, resp.) 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 starts with an *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 (Algorithm 11) is initiated by the leader and broadcast down the tree. When a node v receives the $\langle Expand \rangle$ 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 $\langle Expand \rangle$ requests to before it contacts its parent.

Let T^i denote T after the execution of *Expand* in phase i. Initially, $T = T^0$ consists of the leader node and its Found lists contain all its neighbors. After the first execution of *Expand*, T_1 satisfies the invariant regardless of whether the leader is a low-degree, high-degree, or a star node. 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.*

Argument of termination of *Expand***:** The only randomness in *Expand* is when a high-degree node is waiting to hear a $\langle Star \rangle$ message. The convergecast will also proceed w.h.p. since each high degree node will eventually receive a star message and the leaves will eventually receive what they need to communicate to their parents and each parent will eventually receive what they need in order to

complete the convergecast. Therefore, with high probability, *Expand* will terminate.

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. For simplicity, in this chapter, we use *FindAny* to refer to *FindAny-C*. 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.

Lemma 7 proves that w.h.p. one of these three cases always happen. If the first two cases occur, the phase ends. Otherwise, after a period of waiting, the process repeats until either Case (1) or Case (2) is satisfied.

Implementation: Endpoints of the 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 of these edges are added to the corresponding Found lists 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 after a period of waiting. Waiting is Part (3) of the algorithm and is described below.

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 approximate the number of 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 at least $k/2$.

The nodes $v \in T$ will eventually receive at least $k/2$ messages over outgoing edges of the form $\langle$*Low-degree*$\rangle$. Note that these messages must have been received by v after v executed *Expand*, otherwise, these might not be outgoing edges. To keep only the outgoing edges, each execution of *Expand* empties the Found lists. Therefore, any edge added to Found lists after the list is emptied, is outgoing.

The leader initiates a *ThresholdDetection* procedure in which there is an event for every node v that received a $\langle$*Low-degree*$\rangle$ message over an outgoing edge. 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 of $\min\{(c \log n)/r, 1\}$, where $r = q/2$ is in $[k/64, k/2]$. 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.

Lemma 8 proves that w.h.p. the leader is triggered. It also proves that when it is triggered, w.h.p, at least 1/4 of the outgoing edges to low-degree nodes have been added to $Found_L$ lists.

To provide the proof of the next two lemmas, 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 $X_1, \ldots, X_T$.

Lemma 7. *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 *(input to the algorithm): 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 $1/2^{2c \log n} < 1/n^{2c}$. □

Lemma 8. *W.h.p. the leader receives* $(c/2) \log n$ *messages with the current phase number. When this happens, w.h.p, at least 1/4 of the outgoing edges to low-degree nodes have been added to* $Found_L$ *lists.*

Proof. 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. □

4.3.1 Proof of Theorem 4 for Spanning Trees

In this section, we prove Theorem 4 as it applies to computing the spanning tree of a connected network with a preselected leader. Recall Theorem 4:

Theorem 4. *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* $O(n^{3/2} \log^{3/2} n)$ *time and messages in the KT1 CONGEST model, with high probability.*

Lemma 9. *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 7, there are three cases from the **Search** phase. We argue that in each case either we will have a phase of type A or type B.

If a sampled outgoing edge to a high-degree node is found, the phase is of type A.

If the **Search** phase ends with fewer than $2c \log n$ outgoing edges found and none of them are to high-degree nodes, then these are all the outgoing edges to low-degree nodes, and will all be added to $Found_L$ lists. It is not hard to see that if we explore the whole cutset $(T, V \setminus T)$ either we will eventually hit a high-degree or star node (type A), or the whole network is spanned (type B).

If there are at least $2 \log n$ outgoing edges in the sample, and all go to low-degree nodes, w.h.p., half the edges in the cut go to low-degree nodes. Using *ApproxCut* and *ThresholdDetection*, the number of outgoing edges to low-degree edges will decrease by a factor of 1/4 (type B). □

As for the complexity we have the following two lemmas:

Lemma 10. *The number of phases is bounded by* $O(\sqrt{n} \log^{1/2} n)$.

Proof. By Lemma 9, 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. □

Lemma 11. *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, i.e., $O(n)$, except for the newly added low-degree and star nodes that send to all of their neighbors when they are added to T. But this only adds 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. *ThresholdDetection* requires only $O(\log n)$ messages to be passed up T or $O(n \log n)$ messages overall. Therefore, by Lemma 10 the number of messages over all phases is $O(n \log^{3/2} n)$. □

Theorem 4 for spanning trees in connected networks with a preselected leader follows from Lemmas 11 and 10.

4.3.2 Proof of Lemma 6 (ApproxCut)

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 [KKT15]). 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

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

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:

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

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

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,

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

Since X_j is the sum of independent random variables we can use Chernoff bounds:

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

and,

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

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

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. □

4.3.3 Pseudocodes

Algorithm 7 Initialization of the asynchronous MST construction

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. (They may be star nodes at the same time.)
4: Star nodes send $\langle Star \rangle$ messages to all of their neighbors.
5: Low-degree nodes send $\langle Low\text{-}degree \rangle$ messages to all of their neighbors.
6: **end procedure**

Algorithm 8 Algorithm to detect when the number of events pass a threshold

1: **procedure** THRESHOLDDETECTION
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.
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 $\langle Trigger, i \rangle$ message.
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 9 Approximating the size of cut

1: **procedure** ApproxCut(T)

Approximates the number of outgoing edges. c is a constant.

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**

Algorithm 10 Protocol for constructing the ST that is executed by the leader

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** $FindAny$ finds an edge (u, v) ($u \in T$ and $v \notin T$) **then**
11: u queries 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 11 Expansion algorithm

1: **procedure** EXPAND

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.

2: When node x receives an ⟨*Expand*⟩ message over an edge (x, y):

3: x adds y to *T-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 and 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**

4.4 Constructing the MST

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 level 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 fragments with higher level slow down. Our protocol works in phases where in each phase only fragments that are currently at minimum level continue to grow while other fragments wait. A spanning tree is used to control the fragments based on their level.

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 level zero. Let the precomputed 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 $\langle$*level-request*$\rangle$ to learn the current minimum level among all fragments after this broadcast. Leaves of T send up their fragment level and the minimum level of all fragments is computed via a convergecast.

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

1. $R < R'$: In this case, v' answers immediately to v by sending back an $\langle Accept\rangle$ 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 level R'.

2. $R = R'$: v' responds $\langle Accept \rangle$ 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 level to $R' + 1$.
   Otherwise, v' does not respond to the message until F''s level increases. Once F' increased its level, it responds via an $\langle Accept \rangle$ message, fragments merge, and the new fragment will update its level to R'.

The key point here is that fragments at minimum level 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 level 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 level 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 level.

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

As proved in Lemma 12, this algorithm uses $\tilde{O}(n)$ messages.

Lemma 12. *FindMST uses $O(n \log^3 n / \log\log n)$ messages and finds the MST w.h.p.*

Proof. All fragments start at level zero. Before a phase begins, two broadcasts and convergecasts are performed to only allow fragments at minimum level 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 ID of the higher level fragment over their own fragment. As a result, each phase takes $O(n \log^2 n / \log\log n)$ messages.

A fragment at level R must contain at least two fragments with level $R-1$; therefore, a fragment with level R must have at least 2^R nodes. So, the level of a fragment never exceeds $\log n$. Also, each phase increases the minimum level 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 12, Theorem 4 for minimum spanning trees follows.

Algorithm 12 Asynchronous MST construction

1: **procedure** FINDMST
2: All nodes are initialized as fragments at level 0.
`// Start of a phase`
3: r calls Broadcast($\langle$*level-request*$\rangle$), and *minlevel* is found via a convergecast.
4: r calls Broadcast($\langle Proceed, minlevel \rangle$).
5: Fragment leaders at level *minlevel* that have received the $\langle Proceed, minlevel \rangle$ 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 $\langle Proceed, minlevel \rangle$, a node v does the following:
If it is a leaf in T at level *minlevel*, sends up $\langle Done \rangle$ after increasing its level.
If it is a leaf in T with a level higher than *minlevel*, 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 level.
7: r waits for $\langle Done \rangle$ from all of its children, and starts a new phase at step 3.
8: **end procedure**

4.5 Improving the Time Complexity

In this section, we prove Theorems 5 and 6:

Theorem 5. *There exists an asynchronous algorithm in the KT1 CONGEST model that, w.h.p., computes the MST in $O(n)$ time and with $O(\min\{m, n^{3/2}\log^2 n\})$ messages.*

Theorem 6. *Given an asynchronous MST algorithm with time $T(n,m)$ and message complexity of $M(n,m)$ in the KT1 CONGEST model, w.h.p., we can construct the MST with $O(n^{1-2\epsilon} + T(n, n^{3/2+\epsilon}))$ time and $\tilde{O}(n^{3/2+\epsilon} + M(n, n^{3/2+\epsilon}))$ messages, for $\epsilon \in [0, 1/4]$.*

We provide an asynchronous algorithm that computes the minimum spanning tree or forest (MST) in time proportional to the number of nodes and communication sublinear in m when the network is dense. To the best of our knowledge, our time matches that of the fastest asynchronous MST algorithms in this model which use communication $\tilde{O}(m)$.

In fact, our algorithm is more general. We can convert any asynchronous MST algorithm with time $T(n,m)$ and message complexity of $M(n,m)$ to an algorithm with time $O(n^{1-2\epsilon} + T(n, n^{3/2+\epsilon}))$ and message complexity of $\tilde{O}(n^{3/2+\epsilon} + M(n, n^{3/2+\epsilon}))$, for $\epsilon \in [0, 1/4]$. Picking $\epsilon = 0$ and using Awerbuch's algorithm [Awe87], this results in an MST algorithm with time $O(n)$ and message complexity $\tilde{O}(n^{3/2})$. However, if there were an asynchronous MST algorithm that took time sublinear in n and required messages linear in m, by picking $\epsilon > 0$ we could achieve sublinear time (in n) and sublinear communication (in m), simultaneously. However, to the best of our knowledge, there is no such algorithm.

Ghaffari and Kuhn [GK18] provided an algorithm with $\tilde{O}(D + \sqrt{n})$ round complexity and $\tilde{O}(\min\{m, n^{3/2}\})$ message complexity, in the synchronous network. In other words, they achieved sublinear time in n and sublinear communication in m, simultaneously. In the previous section, we ([MK18]) achieved the same message complexity in the asynchronous model. However, the time complexity of our algorithm was $\tilde{O}(n^{3/2})$.

We use the ideas of the previous sections and [GK18], to introduce a new MST algorithm that achieves $O(n)$ time and $\tilde{O}(n^{3/2})$ messages in the asynchronous network. This algorithm matches the time of the fastest asynchronous MST algorithms that

require $O(m)$ communication. Table 4.1 gives a summary of the MST algorithms that require $o(m)$ communication.

Table 4.1: MST algorithms with $o(m)$ communication

Authors	Synchrony	Time	Messages
King, Kutten, and Thorup [KKT15]	Sync.	$\tilde{O}(n)$	$\tilde{O}(n)$
Mashreghi and King [MK17]	Sync.	$O(\frac{n}{\epsilon})$	$\tilde{O}(\frac{n^{1+\epsilon}}{\epsilon})$
Mashreghi and King [MK18]	Async.	$\tilde{O}(n^{3/2})$	$\tilde{O}(\min\{m, n^{3/2}\})$
Ghaffari and Kuhn [GK18]	Sync.	$\tilde{O}(D + \sqrt{n})$	$\tilde{O}(\min\{m, n^{3/2}\})$
Mashreghi and King [MK19b]	Async.	$O(n)$	$\tilde{O}(\min\{m, n^{3/2}\})$

The main technical contribution of our algorithm here is to find, in time linear in n, a special subgraph with $\tilde{O}(n^{3/2})$ edges that contains all edges of the final MST. Applying Awerbuch's algorithm [Awe87] to this subgraph results in an MST algorithm with time $O(n)$ and message complexity $\tilde{O}(n^{3/2})$.

Again, we divide the nodes into high-degree and low-degree but with a slightly different threshold for the purpose of optimization. Also, the sampling probability for picking star nodes is different.

We define a subgraph G' (from [GK18]) as the subgraph on G induced by all high-degree and star nodes. The heart of our algorithm is to construct a minimum spanning forest on G'. Then, the set of all edges in the minimum spanning forest along with the edges that have at least one low-degree endpoint (edges in $G \setminus G'$) is the special subgraph we use for computing the MST.

To this end, we first construct a spanning forest on G', where the sum of the diameter of all trees in this forest is $O(\sqrt{n})$. We do this in $O(n)$ time and with $\tilde{O}(n^{3/2})$ messages using the waiting technique of [MK18] and an idea of [GK18]. Then, we use this spanning forest to construct the minimum spanning forest.

Before describing the algorithm, we make the following definitions:

- *High-degree and low-degree nodes*: A node is high-degree if its degree is at least $\sqrt{n}\log^2 n$. Otherwise, it is a low-degree node.

- *Star nodes*: A node selects itself to be a star node with probability of $\frac{c}{\log n\sqrt{n}}$.

We assume that our algorithm computes the MST with probability $1 - 1/n^{c_p}$, where constant c_p is a parameter of the algorithm, and c in self-selecting star nodes is another constant depending on c_p. An edge belongs to the subgraph G' if and only if both

of its endpoints are either high-degree or star nodes. Note that G' is not necessarily connected.

The algorithm has three parts:

1. We compute a spanning forest F on G'. We use the ideas [MK18] to compute a spanning tree in each connected component of G'. We also ensure that the sum of the heights of the obtained trees is $O(\sqrt{n})$. This is covered in Section 4.5.1.

2. We compute the minimum weight spanning forest F_{min} on G'. We use an idea of [GK18], along with the fact that the obtained trees in part (1) all have diameter of $O(\sqrt{n})$. This is done using the low-diameter trees in F, to simulate the MST algorithm of [GK18] in the asynchronous environment on each connected component of G'. This part is covered in Section 4.5.2.

3. In this part we only consider the edges in F_{min} and the edges in $G \setminus G'$ which all have at least one low-degree endpoint. This sparse spanning subgraph is called S_{min}. We run an asynchronous MST algorithm with $O(n)$ time and $O(m)$ message complexity (e.g. [Awe87]) on S_{min}. The result is the MST of G. This is covered in Section 4.5.3.

The challenge in part (1) is to modify the algorithm of [MK18], to have star nodes grow fragments of the spanning forest only on G' (not including any low-degree node). In part (2), we observe that the algorithm of [GK18] consists of a number of Borůvka style phases where the computations in each phase are inherently asynchronous. We show that using the low-diameter trees from part (1), we need to synchronize the nodes only at the beginning of each such phase. This allows us to run, without any asymptotic overhead on the complexity, the MST algorithm of [GK18] in the asynchronous network. We run the algorithm on each connected component of G'. This results in the minimum spanning forest for part (2) of our algorithm.

For the last part, the challenge is that there is no global clock to announce the beginning of these three parts to nodes. Therefore, we have all nodes start running the MST protocol of part (3) while part (1) and (2) are being computed. To do this, the MST protocol is delayed by the high-degree or star nodes if they have not yet computed their corresponding tree in F_{min}. We show, however, that this approach in coordinating the protocols does not affect the asymptotic time and message complexity of the MST protocol.

4.5.1 Constructing a Spanning Forest F on G'

Initially, similar to the idea of [GK18], we form a number of height-one fragments around the star nodes. Star nodes send a message to their neighbors. We observe that w.h.p. each high-degree node is adjacent to a star node. Each high-degree node considers itself to be the child of the first star node it has heard from. This gives us the *initial fragments* of the spanning forest, where each fragment is formed by a star node and a subset of its neighbors. Since a star node is self-selected with probability of $\frac{c}{\sqrt{n}\log n}$, w.h.p., there are at most $O(\sqrt{n}/\log n)$ star nodes.

Now, we only need to connect these initial fragments using $O(\sqrt{n}/\log n)$ edges. However, finding outgoing edges to other high-degree or star nodes is a bit hard since there may be many outgoing edges that go to low-degree nodes. Therefore, we use the waiting technique of our MST algorithm in Section 4.4 to explore such edges and exclude them from the search. We have the following lemma.

Lemma 13. *The sum of the diameters of all trees in the spanning forest F is $O(\frac{\sqrt{n}}{\log n})$.*

Proof. There are $O(\sqrt{n}/\log n)$ initial fragments. Since in each of these fragments, zero or more high-degree nodes are connected to a star node, the height of each fragment is one. Therefore, the height of any maximal tree in F, obtained by adding edges between the initial fragments, is no more than two times the final number of star nodes in that tree; therefore, the sum of the heights is $O(\sqrt{n}/\log n)$. □

Implementation

We now describe the details of constructing the spanning forest.

A - Initialization

All nodes first run the Initialization procedure (Algorithm 13). Star nodes self-select themselves with probability of $\frac{c}{\sqrt{n}\log n}$. Star nodes then send a $\langle Star\rangle$ message to all of their nodes. All nodes immediately respond to a $\langle Star\rangle$ message. If a node is high-degree, is not a star node, and this is the first time it hears from a star node, it becomes that star's child and sends back $\langle Child\rangle$ to let the star node know this, as well (line 3). If the node receiving the $\langle Star\rangle$ does not satisfy the conditions for sending back a $\langle Child\rangle$ message, it responds with a $\langle$*Not-Child*$\rangle$ message unless it is star node itself which should respond with a $\langle$*Star-node*$\rangle$ message and will not become a child of the star it has heard from. These messages form the initial height-one fragments.

Upon waking up, low-degree nodes that are not a star send a ⟨*Low-degree*⟩ message to all of their neighbors.

- Fragment IDs: Each fragment has an ID equal to that of the star node leading the fragment. When new nodes join a new fragment they all take that fragment's ID. Each high-degree and star node x has a variable xID which keeps the ID of the fragment it belongs to. We use xID to refer both to the fragment ID and the fragment tree of node x. Note that after initialization all high-degree nodes belong to a fragment.

- Local information of a high-degree node: Whenever a high-degree node receives a ⟨*Low-degree*⟩ message, it remembers that incident edge so it can later exclude it from the search for outgoing edges.

B - Computing Maximal Trees of F

Each star node runs the $MaximalTree$ protocol (Algorithm 15). Each phase of the $MaximalTree$ protocol starts with calling $FindAny$ $16c \log n$ times to sample a set of outgoing edges without replacement (while loop at line 4). Then, the degree and the star status of the external endpoint of the sampled edges is queried. The results are given to the fragment leader.

After sampling, three cases might happen:

1. **There is an edge going to a high-degree or star node:** We can use this edge to connect to another fragment. So, a merge request message is sent over the edge. This message is handled in the procedure $ReceiptOfMerge$ (Algorithm 14).

2. **The size of the sample is strictly less than** $2 \log n$**:** By Chernoff Bounds, if there are at least $2 \log n$ outgoing edges, after sampling $16c \log n$ times without replacement, w.h.p., the size of the sample must be at least $2 \log n$. Therefore, in this case, it must be that we have explored all outgoing edges and none of them went to a high-degree or star node. So, the fragment tree is maximal in G' and the algorithm terminates.

3. **The size of the sample is equal to** $2 \log n$**, and all sampled edges go to low-degree nodes:** Since $FindAny$ finds an outgoing edge uniformly at random from the set of edges used for search, this means that w.h.p. at least

half the outgoing edges are to low-degree nodes. In this case, the fragment waits for more $\langle$*Low-degree*$\rangle$ messages to be received from across the cut. To do the waiting, *ApproxCut* and *ThresholdDetection* are used.

C - Merging Fragments

A fragment (say A) after finding an outgoing edge sends a merge request message along the edge with its fragment ID to the other fragment (say B). This merge message is handled in procedure ReceiptOnMerge (Algorithm 14). This procedure allows fragments with lower IDs to join fragments with higher IDs. If A has a lower ID, its request is accepted, and it should update the ID of all of its nodes to B's ID. However, if A's ID is greater, its response is delayed until B's ID becomes greater than or equal to A's ID. Moreover, if the two have equal IDs, then they are already merged so the request is rejected. As a result, the leader looks for another outgoing edges. We have the following lemma.

Lemma 14. *Computing the maximal trees of the spanning forest F requires $O(n)$ time and $O(n^{3/2}\log^2 n)$ messages w.h.p.*

Proof. In the initialization, there are at most n low-degree non-star nodes that send to at most $O(\sqrt{n}\log^2 n)$ of their neighbors. Also, there are at most $O(\sqrt{n}/\log n)$ star nodes that send to all of their neighbors and receive a response. This overall takes $O(n^{3/2}\log^2 n)$ messages and $O(1)$ time.

To find an outgoing edge, each fragment T runs *FindAny* $O(\log n)$ times. This results in $O(|T|\log n)$ messages and $O(height(T)\log n)$ time. Moreover, finding an outgoing edge to a high-degree or star node may require $O(\log n)$ phases of sampling and waiting. Therefore, each fragment requires $O(|T|\log^2 n)$ messages and $O(height(T)\log^2 n)$ time to find an outgoing edge. There are at most $O(\sqrt{n}/\log n)$ merges that have to be performed using the outgoing edges on the initial height-one fragments. Also, height of a fragment is always $O(\sqrt{n}/\log n)$ (Lemma 13). Therefore, the overall time will be $O((\sqrt{n}/\log n)^2 \cdot \log^2 n) = O(n)$. The overall message complexity is $O(n \cdot \log^2 n \cdot \sqrt{n}/\log n) = O(n^{3/2}\log n)$. □

The following lemma proves the correctness of the algorithm.

Lemma 15. *With high probability, MaximalTree protocol always makes progress. Also, when it terminates, the obtained trees are maximal trees in F.*

Proof. When the fragments find their outgoing edges, the policy for merging always allows the fragment with lowest ID to be merged with another fragment. So, in terms of merging, the algorithm always makes progress. We should only argue that if an outgoing edge to a high-degree or star node exists, it will be found w.h.p. We call such an edge a valid outgoing edge for the sake of this proof.

If there is a valid outgoing in the sample, we are done. Otherwise, the algorithm explores all low-degree non-star nodes outside the fragment by waiting for them to send their $\langle$*Low-degree*$\rangle$ messages. This happens w.h.p. since we use *ApproxCut* and *ThresholdDetection*. *ApproxCut* approximates the cut within a constant factor and *ThresholdDetection* only signals the leader when a constant fraction of the $\langle$*Low-degree*$\rangle$ messages are received. Therefore, each time these two subroutines are applied, a constant fraction of the messages that we expect to be received, will be received with high probability. So, after repeating this $O(\log n)$ times, the number of $\langle$*Low-degree*$\rangle$ messages that have not been received becomes so low that they will not interfere with finding a valid outgoing edge. In the worst-case, when the majority of such messages are received and their corresponding edges are excluded from the search, *FindAny* finds a valid outgoing edge.

If there is no valid outgoing edge, the fragment tree is maximal. After most edges to low-degree nodes are explored, the set of sampled edges becomes very small (possibly empty), and the algorithm terminates. □

Algorithm 13 Initialization of the improved MST algorithm

1: **procedure** INITIALIZATION
2: Every node selects itself to be a *star* node with probability of $\frac{c}{\log n\sqrt{n}}$.
3: Star nodes send a $\langle Star \rangle$ message to all of their neighbors and wait for the response of each message. In response, if a node is high-degree and this is the first star it has hear from it sends back $\langle Child \rangle$. Otherwise, the node responds $\langle Not\text{-}Child \rangle$. If the node receiving the $\langle Star \rangle$ message is also a star, it responds by $\langle Star\text{-}node \rangle$.
4: Low-degree nodes that are not a star send $\langle Low\text{-}degree \rangle$ messages to all of their neighbors.
5: **end procedure**

Algorithm 14 Handling merge requests

1: **procedure** RECEIPTOFMERGE($\langle Merge, tID \rangle$)
2: When node x receives the message $\langle Merge, tID \rangle$ from node t:
3: If x is a high-degree node it waits to hear from at least one star node. Else, if x is a star node it waits to hear the response of its initialization messages.
4: **if** $tID < xID$ **then**
5: x immediately responds by $\langle Accept, xID \rangle$, and considers t as a child.
6: **else if** $tID > xID$ **then**
7: x delays the response until xID becomes greater than or equal to ($\geq$) tID.
8: **else** $tID = xID$
9: x rejects the merge by a $\langle Reject \rangle$ message.
10: **end if**
11: **end procedure**

Algorithm 15 Algorithm to form a maximal tree

1: **procedure** MAXIMALTREE(x)
This algorithm finds a maximal tree of the spanning forest F in G'. x is any star node that is also the leader of this fragment.
2: **repeat** `//beginning of a phase`
3: $counter \leftarrow 0, A \leftarrow \emptyset$.
4: **while** $counter < 16c \log n$ **do**
5: Leader calls $FindAny(E \setminus A)$, where nodes exclude from the search the edges that they have received a $\langle$*Low-degree*$\rangle$ message from, i.e., nodes that are low-degree but not star.
6: **if** $FindAny$ finds an edge (u, v) ($u \in T$ and $v \notin T$) **then**
7: $A = A \cup (u, v)$.
8: u sends a message and asks for v's degree, and whether or not it is a star node. u then sends up the result to the leader.
9: **end if**
10: $counter \leftarrow counter + 1$.
11: **end while**
12: **if** $\exists (u, v) \in A$ s.t. v is high-degree or star **then**
13: Leader chooses an edge to a high-degree or star node arbitrarily, and sends a $\langle Merge, ID \rangle$ message over it. (ID is this fragment's ID.)
14: If accepted, this fragment is updated to be a subtree of fragment vID rooted at node v.
15: If rejected, the leader starts a new phase.
16: **else if** $|A| < 2 \log n$ **then**
17: Leader terminates the algorithm.
18: **else** `// waiting`
19: $r \leftarrow ApproxCut()/2$. Then leader calls $ThresholdDetection(r)$.
20: Leader waits to receive a trigger message and then starts a new phase.
21: **end if**
22: **until**
23: **end procedure**

4.5.2 Constructing the Minimum Spanning Forest F_{min} on G'

It is easier to describe the algorithm for constructing F_{min} as an MST algorithm on each connected component of G'. Formally, let us fix a connected component C in G'. Let T be the low-diameter tree computed on C from part (1). Now, we describe how to compute T_{min}, the minimum weight spanning tree of C, using T whose diameter is bounded by $O(\sqrt{n})$ (we omit the $\log n$ division as it is not needed in this part).

We simulate the synchronous MST algorithm of [GK18] in the asynchronous model, on C. So, our algorithm in this part closely follows their algorithm. Our simulation is in a way that results in no asymptotic overhead on the time and message complexity of [GK18]. Our algorithm has two parts:

Part A - Computing fragments of F_{min} with low diameter: Let r be the root of T. This part is consisted of $O(\log n)$ Borůvka phases. An important point is that in this part merging is done differently from the standard Borůvka algorithm. In each phase, before fragments start merging, each fragment flips a fair coin. Then, if a fragment A sends a merge request to a fragment B, the merge is only accepted if A has a Tail coin, and B has a Head coin. This rule keeps the height low by preventing fragments to merge with each other in a long chain.

Moreover, not all fragments look for minimum outgoing edges. In fact, fragments whose diameter is strictly less than $\sqrt{n}$ are *active*, and the other fragments are *inactive*. Only active fragments can look for the minimum outgoing edge. However, *inactive* fragments may still accept the merge request of other fragments, given that the active fragment has a Tail coin and the inactive fragment has a Head coin.

The objective is to grow $O(\sqrt{n})$ MST fragments where the diameter of each fragment is bounded by $O(\sqrt{n})$ (in particular between $\sqrt{n}$ and $5\sqrt{n}$). The steps are as follows:

1. Initially, each node is a fragment.

2. Begin the search: r tells all nodes (via a broadcast) to begin the search for the minimum outgoing edges. Then, each fragment computes its minimum outgoing edge using $FindMin$. When nodes in a fragment know that the minimum outgoing edge is computed, they let r know using a convergecast. Note that they just let r know that they have finished the computation and do not send the found edge. r waits until all fragments compute their minimum outgoing edges.

3. Begin the merge: r tells all nodes to merge using the recently found minimum outgoing edges. All fragments then send merge requests over their minimum outgoing edges. If the merge is accepted, the fragment IDs of the nodes will be updated. Once fragments finished this part, they let r know by a convergecast.

4. Begin to truncate: r tells all fragments to make sure that their height stays in $[\sqrt{n}, 5\sqrt{n}]$. Once the fragments merge, it is possible that the height of the resulting fragments exceeds $5\sqrt{n}$. However, because of the coin flips, it is guaranteed the height will not exceed $5\sqrt{n}$ by more than a constant factor. To make sure that all fragments have height in $[\sqrt{n}, 5\sqrt{n}]$ we do as follows. The leader broadcasts a message to all nodes so that all nodes know their distance from the leader. Whenever the distance of a node from the fragment leader becomes a multiple of $\sqrt{n}+1$, the edge between that node to its parent is discarded. In case that the height of the remaining subtree gets below $\sqrt{n}$ that operation is undone. Once truncating is done, nodes notify r via a convergecast.

5. Check the phase: r computes the minimum height of all fragments via a broadcast followed by a convergecast. If the minimum height is $< \sqrt{n}$, r goes to Step 2 and announces the beginning of a new phase. Otherwise, r announces the beginning of part (B).

We call the fragments obtained in part (A) *old fragments*. Also, we call the leader of these fragments *old leaders*.

Once r announces the beginning of part (B), no more synchronization is required as the computations in part (B) are inherently asynchronous.

Part B - Completing F_{min}: Since the diameter of F_{min} can be as large as $\Theta(n)$, in this part, nodes communicate through a different network. In fact, nodes communicate via the old fragments and through the spanning tree T. The key point in this part is to grow fragments in a way that each fragment is consisted of a number of old fragments.

To find the remaining $O(\sqrt{n})$ minimum outgoing edges, and merge the old fragments, fragments rely on r to do the computation. In particular, nodes first send the necessary information to their old leaders. Then, the old leaders send their information (via T) to r. So, each message travels $O(\sqrt{n})$ hops.

Then, r computes the minimum outgoing edges and sends the instructions regarding the merges back to the nodes. Once the minimum outgoing edges are computed

and r knows how the fragment IDs should be updated, information regarding the updates can be passed down to all nodes by reversing the direction of the convergecast messages.

Note: Since each fragment in part (B) is consisted of a number of old fragments, r should pass the ID updates only to the old leaders. Then, the old leaders can pass the updates to all nodes in the old fragment.

Lemma 16. *Computing F_{min} requires $\tilde{O}(n)$ messages and $\tilde{O}(\sqrt{n})$ time.*

Proof. In part (A) there are $O(\log n)$ phases since the fragments are synchronized via T and in each phase a constant fraction of all fragments can be merged. In each phase, fragments use $FindMin$ which takes $\tilde{O}(n)$ messages and $\tilde{O}(\sqrt{n})$ time. Merging and truncating in each phase needs a constant number of broadcasts and convergecasts which take $O(n)$ messages and $O(\sqrt{n})$ time. Therefore, overall, part (A) takes $\tilde{O}(n)$ messages and $\tilde{O}(\sqrt{n})$ time.

In part (B), we only broadcast the beginning of part (B). Afterwards, all computations are performed exactly in the same way as [GK18]. Part (B) has also $O(\log n)$ phases. However, we do not need to synchronize phase by phase. In each phase, all nodes give the necessary information to the old leaders. This takes $\tilde{O}(n)$ messages and $\tilde{O}(\sqrt{n})$ time. Then, the old leaders give the necessary information to r. This also takes $\tilde{O}(n)$ messages and $\tilde{O}(\sqrt{n})$ time since there are $O(\sqrt{n})$ messages that have to pipelined and travel $O(\sqrt{n})$ hops.

Over all connected components of G', the algorithm takes $\tilde{O}(n)$ messages and $\tilde{O}(\sqrt{n})$ time, and the lemma follows. □

The correctness of part (2) follows from the correctness of the synchronous algorithm of [GK18]. The computations in each of steps in part (A) and the whole part (B) can be simulated asynchronously since they do not depend on a global clock to be correct. Therefore, such simulation does not affect the correctness of the protocol.

4.5.3 Constructing the MST

We now have constructed the desired spanning subgraph S_{min} on G. This subgraph is consisted of all edges that have at least one low-degree endpoint, and the edges of the minimum spanning forest F_{min}. Therefore, S_{min} has $\tilde{O}(n^{3/2})$ edges.

To constructed the final MST we can use any $O(n)$ time asynchronous MST algorithm that requires no more than $O(m)$ messages on dense graphs. We use the

algorithm of [Awe87].

However, the only catch is that in an asynchronous network we cannot first complete parts (1) and (2) of the algorithm and then move to part (3). To solve this, we have all low-degree nodes run the MST protocol, right after they sent their initialization messages. High-degree and star nodes only begin to participate in the MST protocol once they have computed their corresponding tree in F_{min}. Computing F_{min} requires $O(n)$ time (for parts (1) and (2)). Therefore, the overall delays that high-degree and star nodes can cause is no more than $O(n)$, which is the same as the time complexity of the MST protocol. We now prove the following lemma which implies that running an MST algorithm on S_{min} computes the correct MST of G.

Lemma 17. *All edges of the final MST must be in subgraph S_{min}.*

Proof. Suppose on the contrary that there is some edge in the MST that is not in the set of edges of the subgraph S_{min}. This edge cannot have a low-degree endpoint because all such edges belong to the subgraph. Therefore, it must be an edge that is in the edges of G' but does not appear in F_{min}. However, adding this edge to the corresponding tree in F_{min} creates a cycle. By removing the maximum weight edge on that cycle we obtain a lighter tree which contradicts the fact that F_{min} is the minimum spanning forest of G'. □

Correctness of this part follows from the correctness of the MST protocol that is used. The only thing we do in this part is that we let high-degree and star nodes participate in the protocol only when they finished computing their trees in F_{min}. This will only cause an additive delay on the MST protocol and will not affect the complexity.

Theorem 5 follows from Lemmas 14, 15, 16, and 17.

4.5.4 Theorem 6 and Sublinear Time

The only parts in our algorithm that require linear time in n are part (1) and part (3). Part (3) is linear due to the asynchronous MST algorithm that we use on S_{min}. However, in part (1), the time complexity comes directly from the number of star nodes. If we have $n^{1/2-\epsilon}$ star nodes, the time complexity of this part is $\tilde{O}(n^{1-2\epsilon})$. Since part (2) has a time complexity of $\tilde{O}(\sqrt{n})$, we only consider $\epsilon \in [0, 1/4]$. However, we have to increase the threshold for low-degree nodes to $n^{1/2+\epsilon}$, as well. This will result in a message complexity of $\tilde{O}(n^{3/2+\epsilon})$ which is interesting if the input graph

has asymptotically more edges. For instance, by selecting $n^{1/4}$ ($\epsilon = 1/4$) star nodes, we have $\tilde{O}(n^{1/2})$ time and $\tilde{O}(n^{7/4})$ message complexity.

Notice that we presented our algorithm for $\epsilon = 0$; however, we optimized the $\log n$ factor to make sure that the time complexity remains $O(n)$. Overall, the time and message complexity of parts (1) and (2) of our algorithm will be $\tilde{O}(n^{1-2\epsilon})$ and $\tilde{O}(n^{3/2+\epsilon})$, respectively. And the subgraph S_{min} will have $\tilde{O}(n^{3/2+\epsilon})$ edges. Therefore, by applying an asynchronous MST algorithm with time $T(n, m)$ and message complexity of $M(n, m)$, we get an algorithm with time $O(n^{1-2\epsilon} + T(n, n^{3/2+\epsilon}))$ and message complexity of $\tilde{O}(n^{3/2+\epsilon} + M(n, n^{3/2+\epsilon}))$. This proves Theorem 6.

Chapter 5

Conclusions and Open Problems

For the synchronous case, we have shown that time can be brought down to linear in n while maintaining a communication complexity of $\tilde{O}(\frac{n^{1+\epsilon}}{\epsilon})$, where $\epsilon = \Omega(\frac{\log\log n}{\log n})$, for the MST and $O(n \log n \log\log n)$ for the spanning tree problem. We have also provided the first minimum spanning tree construction algorithm which uses time approximately proportional to the diameter of the MST without paying an excessive cost for communication. Ghaffari and Kuhn [GK18] have also provided an algorithm with $\tilde{O}(D + \sqrt{n})$ round complexity and $\tilde{O}(\min\{m, n^{3/2}\})$ message complexity. Therefore, even sublinear time is achievable while keeping the message complexity at $o(m)$. An intriguing problem is to see whether it is possible for a synchronous algorithm to remove the $\sqrt{n}$ factor from time when D is small. For example, is there an algorithm that can achieve $O(D)$ time and $o(m)$ communication complexity when D is constant?

We have also 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 in the asynchronous model, as well. Open problems that are raised by these results are: (1) Does the asynchronous KT1 model require substantially more communication that the synchronous KT1 model? Or is it possible to achieve $\tilde{O}(n)$ message complexity in the presence of asynchrony? (2) Can we still improve the time complexity (e.g. to $\tilde{O}(D + \sqrt{n})$) of the algorithm while maintaining the message complexity? We know from Theorem 6 that this can be done if there exists an asynchronous MST algorithm that takes sublinear time if the diameter of the network is low, and has $\tilde{O}(m)$ message complexity. To the best of our knowledge no such algorithm exists.

Similarly, is it possible to have an asynchronous breadth-first-search (BFS) algorithm that requires $\tilde{O}(D)$ time and $\tilde{O}(m)$ messages? Having such algorithm would help in synchronizing the network and would allow us to achieve sublinear time by following the same strategy as part two of our algorithm on the whole graph (Section 4.5).

Bibliography

[AGV87] Baruch Awerbuch, Oded Goldreich, and Ronen Vainish. *On the message complexity of broadcast: A basic lower bound*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.

[AGVP90] 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.

[AK93] Sudhanshu Aggarwal and Shay Kutten. Time optimal self-stabilizing spanning tree algorithms. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 400–410. Springer, 1993.

[AKM+93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 652–661. ACM, 1993.

[AKM+07] 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.

[AP90] 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.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

[Awe87] 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.

[BK95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM (JACM)*, 42(1):269–291, 1995.

[BK07] Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *International Symposium on Distributed Computing*, pages 92–107. Springer, 2007.

[CT85] F Chin and HF Ting. An almost linear time and o (nlogn+ e) messages distributed algorithm for minimum-weight spanning trees. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 257–266. IEEE, 1985.

[Elk04] 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.

[Elk06] 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.

[Elk17a] 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.

[Elk17b] Michael Elkin. A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. *arXiv preprint arXiv:1703.02411*, 2017.

[FM95] 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.

[FM04] Michalis Faloutsos and Mart Molle. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing*, 17(2):151–170, 2004.

[Gaf85] Eli Gafni. Improvements in the time complexity of two message-optimal election algorithms. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 175–185. ACM, 1985.

[GHS83] 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.

[GK18] Mohsen Ghaffari and Fabian Kuhn. Distributed mst and broadcast with fewer messages, and faster gossiping. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[GKP98] 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.

[KKT15] 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.

[KP95] 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.

[KPP+15] 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.

[KPS97] Shay Kutten and Boaz Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 149–158. ACM, 1997.

[KPS98] Shay Kutten and Boaz Patt-Shamir. Asynchronous time-adaptive self stabilization. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, page 319. ACM, 1998.

[MK17] 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.

[MK18] Ali Mashreghi and Valerie King. Broadcast and Minimum Spanning Tree with o(m) Messages in the Asynchronous CONGEST Model. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[MK19a] Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous MST and low diameter tree construction with sublinear communication. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary.*, pages 49:1–49:3, 2019.

[MK19b] Ali Mashreghi and Valerie King. Faster asynchronous MST and low diameter tree construction with sublinear communication. *CoRR*, abs/1907.12152, 2019.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36, 2001.

[Pel00] David Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5, 2000.

[PR00] David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, May 2000.

[Pri05] Distributed Coordination Primitives. *The Price of Locality: Exploring the Complexity of.* PhD thesis, ETH Zürich, 2005.

[PRS16] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. *arXiv preprint arXiv:1607.06883*, 2016.

[SB95] Gurdip Singh and Arthur J Bernstein. A highly asynchronous minimum spanning tree protocol. *Distributed Computing*, 8(3):151–161, 1995.

[SHK+12] 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.

[Tho18] Mikkel Thorup. Sample (x)=(a* x¡= t) is a distinguisher with probability 1/8. *SIAM Journal on Computing*, 47(6):2510–2526, 2018.