Abstract

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

Keywords

Distributed computing • Minimum spanning tree • Broadcast tree

1 Introduction

We consider a distributed network as an undirected graph with $n$ nodes and $m$ edges, and the problem of finding a spanning tree and a minimum spanning tree (MST) with efficient communication. That is, we require that every node in the graph learns exactly the subset of its incident edges which are in the spanning tree or MST, respectively. A spanning tree enables a message to be broadcast from one node to all other nodes with only $n-1$ edge traversals. In a sensor or ad hoc network where the weight of a link between nodes reflects the amount of energy required to transmit a message along the link [23], the minimum spanning tree (MST) provides an energy efficient means of broadcasting. The problem of finding a spanning tree in a network has been studied for more than three decades, since it is the building block of solutions to many other fundamental problems such as counting, leader election, and deadlock resolution [3].

A spanning tree can be constructed by a simple breadth-first search from a single node using $m$ bits of communication. The tightness of this communication bound was a “folk theorem”, according to Awerbuch et al. [5]. Their 1990 paper defined the KT1 model where nodes have unique IDs and know only their neighbours. It showed, for a limited class of algorithms, a lower bound of $\Omega(m)$ messages in a synchronous KT1 network. In 2015, Kutten et al. [23] proved a lower bound for general randomized algorithms with $O(\log n)$ bit messages, in the KT0 model, where nodes do not know their neighbours. It showed, for a limited class of algorithms, a lower bound of $\Omega(m)$ messages in a synchronous KT1 network. In 2015, King, Kutten, and Thorup gave the first distributed algorithm (“KKT”) with $o(m)$ communication to build a broadcast tree and MST in the KT1 model. They devised Monte Carlo algorithms in the synchronous KT1 model with $O(n)$ communication [22]. This paper and a followup paper [25] left open the problem of whether a $o(n)$ bit communication algorithm in the asynchronous model was possible, for either the spanning tree or MST problem, when nodes know their neighbours’ IDs.

In an asynchronous network, there is no global clock. All processors may wake up at the start and send messages, but further actions by a node are event-driven, i.e., in response to messages received. The pioneer work of Gallager, Humblet,
and Spira [17] (“GHS”) presented an asynchronous protocol for finding the MST in the CONGEST model, where messages are of size $O(\log n)$. GHS requires $O(m + n \log n)$ messages and $O(n \log n)$ time if all nodes are awakened simultaneously. Afterwards, researchers worked on improving the time complexity of MST algorithms in the CONGEST model but the message complexity remained $\Omega(m)$.

In this paper, we provide the first algorithm in the KT1 model which uses $o(n)$ bits of communication for finding a spanning tree in an asynchronous network, specifically we show the following:

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

Note that throughout this work, we will not be discussing the time complexity of our algorithm since in the worst-case, it is asymptotically the same as its message complexity. Please see preliminaries section, for a brief discussion on this.

Trivially, once any spanning tree is constructed, broadcasting can be done using $O(n)$ messages from any of the nodes.

### 1.1 Techniques

Many distributed algorithms to find an MST use the Boruvka method: Starting from the set of isolated nodes, a forest of edge disjoint rooted trees which are subtrees of the MST are maintained. The algorithms run in phases: In a phase, in parallel, each tree $A$ finds a minimum weight outgoing edge, that is, one with exactly one endpoint in $A$ and its other endpoint in some other tree $B$. Then the outgoing edge is inserted to create the “merged” tree containing the nodes of $A$ and $B$. In what seems an inherently synchronous process, every tree (or a constant fraction of the trees) participates in some merge, the number of trees is reduced by a constant factor per phase, and $O(\log n)$ phases suffice to form a single tree. [3,17,22,25].

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

While $\text{FindAny}$ and $\text{FindMin}$ are asynchronous procedures, the Boruvka approach of [22] does not seem to work in an asynchronous model with $o(m)$ messages, as it does not seem possible to prevent only one tree from growing, one node at a time, while the other nodes are delayed, for a cost of $O(n^2)$ messages. The asynchronous GHS also uses $O(\log n)$ phases to merge trees in parallel, but it is able to synchronize the growth of the trees by assigning a rank to each tree. A tree which finds a minimum outgoing edge waits to merge until the tree it is merging with is of equal or higher rank. The GHS algorithm subtly avoids traversing the whole tree until a minimum weight outgoing edge to an appropriately ranked tree is found. This method seems to require communication over all edges in the worst case.

The technique for building a spanning tree in this paper is very different from the technique in [22] or [17]. 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 classifies itself as a low-degree or a high-degree node. Nodes that have degree less than $\sqrt{n} \log^{3/2} n$ are low-degree nodes. Otherwise, they are high-degree nodes. Also, each node selects itself to be a star node with probability $1/\sqrt{n \log n}$. (We use $\log n$ to denote $\log_2 n$.) This technique is inspired from [12], and provides a useful property that every node whose degree is at least $\sqrt{n} \log^{3/2} n$ is adjacent to a star node with high probability. Initially, star nodes (and low-degree nodes) send out messages to all of their neighbours. Each high-degree node which joins $T$ waits until it hears from a star node and then invites it to join $T$. In addition, when low-degree and star nodes join $T$, they invite their neighbours to link to $T$ via their incident edges. Therefore, with high probability, the following invariant for $T$ is maintained as $T$ grows:

**Invariant 1** $T$ includes all neighbours 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 $\text{Expand}$ (described later), and after each execution of $\text{Expand}$ the invariant is satisfied with high probability.

As mentioned earlier, dividing nodes into two groups based on their degree, i.e., low-degree and high-degree nodes, is crucial for our algorithm to succeed in saving messages. Therefore, it is important to mention that such division in an asynchronous network does not automatically enable us to use approaches that can be used in the synchronous model. For example, in the synchronous model, if low-degree nodes send messages to all of their neighbours, after only one round all nodes learn which of their neighbours are low-degree and which are high-degree. Then, they can construct a subgraph of higher degree nodes which consequently allows them to save messages. However, in the asynchronous model, a node, not hearing from its neighbour, does not know when to conclude that its neighbour is high-degree.

The challenge is for high-degree nodes in $T$ to find neighbours outside $T$. If in each phase, an outgoing edge from a high-degree node in $T$ to a high-degree node $x$ (not in
A phase, there are a total of \(O(x)\) complexity of the aforementioned papers is found and invited to join. As the number of star nodes is no more than \(O(\sqrt{n})\), 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 neighbour has high degree without receiving a message from its neighbour.

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

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

### 1.2 Related work

**Asynchronous networks** The first breakthrough for computing the MST was by Gallager, Humblet, and Spira [17], 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 \(n [3,8,15,16,32]\). 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. [18] were the first to give a sublinear time \(O(D + \sqrt{n} \log^{1/2} n)\)-round MST algorithm, where \(D\) is the hop diameter of the network. Kutten and Peleg [24] gave an algorithm with \(O(D + \sqrt{n} \log^{2} n)\) time complexity, and Elkin [9] 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 [4,10,23,29,31]. There are algorithms that match both lower bounds simultaneously up to a polylogarithmic factor [13,28].

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 [5]. 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. [22], provided an algorithm in the KT1 model with \(O(n)\) 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.

The Awerbuch et al. [5] lower bound on the number of messages holds only for (randomized) algorithms where messages may contain a constant number of IDs, and IDs are processed by comparison only and for general deterministic algorithms, where ID’s are drawn from a very large size universe.

Lower bounds on time for approximating the minimum spanning tree have been proved in the synchronous KT0 model In [10,31], Kutten et al. [23] show an \(\Omega(m)\) lower bound on message complexity for randomized general algorithms in the KT0 model. In this paper we show that in the KT1 CONGEST model and with asynchronous communication, it is possible to provide randomized algorithms to achieve \(o(m)\) message complexity.

**FindAny and FindMin** which appear in the KKT algorithms build on ideas for sequential dynamic connectivity in [21]. A sequential dynamic ApproxCut also appeared in that paper [21]. Solutions to the sequential linear sketching problem for connectivity [1] share similar techniques but require a more complex step to verify when a candidate edge name is an actual edge in the graph, as the edges names are no longer accessible once the sketch is made (See Sect. 2.4).

The threshold detection problem was introduced by Emek and Korman [14]. It assumes that there is a rooted spanning tree \(T\) where events arrive online at \(T\)’s nodes. Given some threshold \(k\), a termination signal is broadcast by the root if and only if the number of events exceeds \(k\). We use a naive solution of a simple version of the problem here.

A synchronizer, introduced by Awerbuch [2] and studied in [6,7,11,30], is a general technique for simulating a synchronous algorithm on an asynchronous network using
communications along a spanning tree. To do this, the spanning tree must be built first. Using a general synchronizer imposes an overhead of messages that affect every single step of the synchronous algorithm that one wants to simulate, and would require more communication than our special purpose method of using our spanning tree to synchronize the modified GHS.

There are two related works in the synchronous CONGEST model [19,20] that provide MST algorithms with $O(n^{3/2})$ messages and in $O(D + \sqrt{n})$ time. The idea of selecting a small subset of nodes (which we will call star nodes in this paper) to help with reducing the message complexity is the shared among these works. More importantly, there is the need to treat nodes differently based on their degree. Roughly speaking, nodes with degree $> \sqrt{n}$ (called high-degree nodes) are treated differently from nodes whose degree is $\leq \sqrt{n}$ (called low-degree nodes). A node has to have a way to tell which of their neighbours are high-degree and which are not. This distinction can be made very easily if the communication is synchronous. It can be done with $O(1)$ time and $O(n^{3/2})$ messages. However, in an asynchronous network there is no efficient way to distinguish these nodes. Our paper is different from the aforementioned works in the techniques it provides to deal with the problems that arise in the presence of asynchrony. We achieve the same message complexity for the synchronous model while the time complexity of our algorithms is also $O(n^{3/2})$.

Note that during this journal publication, we have provided another result in asynchronous model that keeps the message complexity since the worst-case execution time if the delivery delays of any sent message can have an arbitrary delay before it is delivered. Another way of thinking about this is that the communication is event-driven; nodes take actions upon receiving messages (or waking up for the first time). In this work, our assumption is that all nodes wake up simultaneously and start to execute their protocols.

However, in an asynchronous network, there is no global clock. Any sent message can have an arbitrary delay before it is delivered. Another way of thinking about this is that the communication is event-driven; nodes take actions upon receiving messages (or waking up for the first time). In this work, our assumption is that all nodes wake up simultaneously and start to execute their protocols.

In an asynchronous network, time complexity is defined as the worst-case execution time if the delivery delays of all messages are normalized to be at most one time unit. Therefore, the time complexity is determined by the worst-case chain of events. We assume that the time for completing local computations within a node is negligible.

The time complexity of an asynchronous algorithm is trivially bounded by its message complexity since the worst case chain of events. We assume that the time for completing local computations within a node is negligible.

2 Preliminaries

2.1 Model

Let $c \geq 1$ be any constant. The communication network is the undirected graph $G = (V, E)$ over which a spanning tree or MST will be found. Edge weights are integers in $[1, n^c]$. IDs are assigned uniquely by the adversary from $[1, n^c]$. All nodes have knowledge of $c$ and $n$ which is an upper bound on $|V|$ (number of nodes in the network) within a constant factor. All nodes know their own ID along with the IDs of their neighbours (KT1 model) and the weights of their incident edges. Nodes have no other information about the network, e.g., they do not know $|E|$ or the maximum degree of the nodes in the network. If the edge weights are not unique they can be made unique by appending the ID of the endpoints to its weight, so that the MST is unique. Nodes can only send direct messages to the nodes that are adjacent to them in the network. Our algorithm is described in the CONGEST model in which each message has size $O(\log n)$. Note that in the literature CONGEST usually refers to a synchronous network; however, our algorithms are asynchronous and by CONGEST we only mean that the size of each message is bounded by $O(\log n)$. The KT1 CONGEST model has been referred to as the “standard model” [5].

Message cost is the sum over all edges of the number of messages sent over each edge during the execution of the algorithm. If a message is sent it is guaranteed to be eventually received.

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

We initially assume there is a special node (called leader) at the start and the graph is connected. These assumptions are dropped in Sect. 5.

2.2 Communication and time complexity

Networks can be synchronous or asynchronous. Communication in a synchronous network proceeds in rounds where at the beginning of each round the nodes that intend to send a message to a subset of their neighbours, send their messages. These messages are guaranteed to be received by the end of this round. It is like communication is being synchronized using a global clock that all nodes have access to. Time complexity in a synchronous network is the number of rounds that an algorithm needs to complete. This is also known as the round complexity of the algorithm.

However, in an asynchronous network, there is no global clock. Any sent message can have an arbitrary delay before it is delivered. Another way of thinking about this is that the communication is event-driven; nodes take actions upon receiving messages (or waking up for the first time). In this work, our assumption is that all nodes wake up simultaneously and start to execute their protocols.

In an asynchronous network, time complexity is defined as the worst-case execution time if the delivery delays of all messages are normalized to be at most one time unit. Therefore, the time complexity is determined by the worst-case chain of events. We assume that the time for completing local computations within a node is negligible.

The time complexity of an asynchronous algorithm is trivially bounded by its message complexity since the worst
chain of execution is if all messages are sequentially organized and there is no parallelism. In this paper, we rely on this fact and will not be discussing the time complexity of our algorithms.

2.3 Definitions and subroutines

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

The algorithm uses the following subroutines and definitions:

- **Broadcast($M$)**: Procedure whereby the leader of $T$ sends a message $M$ to its children and its children broadcast to their subtrees.

- **Expand**: A procedure for adding nodes to $T$ and preserving Invariant 1 after doing so.

- **FindAny**: Returns to the leader an outgoing edge chosen uniformly at random with probability $1/16$, or else it returns $\emptyset$. The leader then broadcasts the result. $\text{FindAny}$ requires $O(n)$ messages. We specify $\text{FindAny}(E')$ when we mean that the outgoing edge must be an outgoing edge in a particular subset $E' \subseteq E$.

- **FindMin**: is similarly defined except the edge is the (unique) minimum cost outgoing edge. This is used only in the minimum spanning tree algorithm. $\text{FindMin}$ requires $O(n \log^2 n/\log \log n)$ messages.

- **ApproxCut**: A function which w.h.p. returns an estimate in $[k/32, k]$ where $k$ is the number of outgoing edges from $T$ and $k > c \log n$ for $c$ a positive constant. It requires $O(n \log n)$ messages. $\text{FindAny}$ and $\text{FindMin}$ are described in [22] (The $\text{FindAny}$ is called $\text{FindAny-C}$ there.) $\text{FindAny-C}$ was used to find any outgoing edge in the previous paper. It is not hard to see that the edge found is a random edge from the set of outgoing edges; we use that fact here. The relationships among $\text{FindAny}$, $\text{FindMin}$ and $\text{ApproxCut}$ below are described in the next subsection.

- **FoundL($v$)**, **FoundO($v$)**: Two lists of edges incident to node $v$, over which $v$ will send invitations to join $T$ the next time $v$ participates in $\text{Expand}$. After this, the list is emptied. Edges are added to $\text{FoundL}(v)$ when $v$ receives $\langle \text{Low} - \text{degree} \rangle$ message or the edge is found by the leader by sampling and its external endpoint is low-degree. Otherwise, an edge is added to $\text{FoundO}(v)$ when $v$ receives a $\langle \text{Star} \rangle$ 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 $\langle \text{Low} - \text{degree} \rangle$ and $\langle \text{Star} \rangle$. This may cause an edge to be in both lists which is handled properly in the algorithm.

- $T - \text{neighbour}(v)$: A list of neighbours of $v$ in $T$. This list, except perhaps during the execution of $\text{Expand}$, includes all low-degree neighbours of $v$ in $T$. This list is used to exclude from $\text{FoundL}(v)$ any non-outgoing edges.

- **ThresholdDetection($k$)**: A procedure that is used to inform the leader w.h.p. when the number of events experienced by the nodes in a fragment $T$ reaches the threshold $k/4$. The leader initiates this procedure and provides the parameter $k$. The nodes in $T$ experience no more than $k < n^2$ events w.h.p. Here, an event is the receipt of $\langle \text{Low} - \text{degree} \rangle$ over an outgoing edge. Following the completion of $\text{Expand}$, all edges $(u, v)$ that are in $\text{FoundL}(u)$ are events if $v \notin T - \text{neighbour}(u)$. The idea for this procedure is to approximate the number of outgoing edges from $T$, and using this approximation when we know that at least half the outgoing edges are to low-degree nodes. This provides a constant approximation on the number of events that are expected to be received. $O(|T| \log n)$ messages suffice to complete this procedure.

**Note** The reason we need the $T - \text{neighbour}$ data structure is that we assumed that messages may not be received in the same order they were sent. In particular, $\langle \text{Low} - \text{degree} \rangle$ messages from neighbours of a node in the fragment tree could be received very late. Therefore, we need to have this data structure so that when the fragment is waiting for events (in $\text{ThresholdDetection}$) we do not count these late messages as events. An alternative way to deal with this is to have nodes send back acknowledgment messages if they receive a $\langle \text{Low} - \text{degree} \rangle$ message. A low-degree node will only send its future messages only if it has received an acknowledgement for its initial $\langle \text{Low} - \text{degree} \rangle$ message. This guarantees that no $\langle \text{Low} - \text{degree} \rangle$ message from a node inside the fragment will contribute to the events.

2.4 Implementation of $\text{FindAny}$, $\text{FindMin}$ and $\text{ApproxCut}$

We briefly review $\text{FindAny}$ in [22] and explain its connection with $\text{ApproxCut}$. The key insight is that an outgoing edge is incident to exactly one endpoint in $T$ while other edges are incident to zero or two endpoints. If there was exactly one outgoing edge, the parity of the sum of all degrees in $T$ would be 1, and the bit-wise XOR of the binary representation of the names of all incident edges would be the name of the one outgoing edge.

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

Since each edge has the same probability of falling in \([1, 2^{min}]\), this procedure gives a randomly selected edge. Note also that the leader can instruct the nodes to exclude certain edges from the XOR, say incident edges of weight greater than some \( w \). In this way the leader can binary search for the minimal weight outgoing edge to carry \( FindMin \). Similarly, the leader can select random edges without replacement.

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

We show:

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

The proof is given in Sect. 3.2.

### 3 Asynchronous ST construction with \( o(m) \) messages

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

Initially, each node selects itself with probability \( 1/\sqrt{n \log n} \) as a star node. Low-degree and star nodes initially send out \( \langle Low - degree \rangle \) and \( \langle Star \rangle \) messages to all of their neighbours, respectively. (We will be using the \( \langle M \rangle \) notation to show a message with content \( M \).) A low-degree node which is a star node sends both types of messages.

At any point during the algorithm, if a node \( v \) receives a \( \langle Low - degree \rangle \) or \( \langle Star \rangle \) message through some edge \( e \), it adds \( e \) to \( Found_L(v) \) or \( Found_O(v) \) respectively.

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

1) **Expansion** Each phase is started with \( Expand \). \( Expand \) adds to \( T \) any nodes which are external endpoints of outgoing edges placed on a \( Found \) list of any node in \( T \) since the last time that node executed \( Expand \). In addition, it restores \( Invariant \) \( 1 \) for \( T \). Recall the invariant statement: \( T \) includes all neighbours of any star or low-degree node in \( T \). Each high-degree node in \( T \) is adjacent to a star node in \( T \).

**Implementation** \( Expand \) is initiated by the leader and broadcast down the tree. When a node \( v \) receives \( \langle Expand \rangle \) message for the first time (it is not in \( T \)), it joins \( T \) and makes the sender its parent. If it is a high-degree node and is not a star, and is receiving the \( \langle Expand \rangle \) message for the first time, it has to wait until it receives a \( \langle Star \rangle \) message over some edge \( e \), and then adds \( e \) to \( Found_O(v) \). At all times, if this node received any \( \langle Low - degree \rangle \) messages, it adds the corresponding neighbours to its \( Found_L(v) \) list. Once the node receives the \( \langle Star \rangle \) message, it proceeds by forwarding the \( \langle Expand \rangle \) over the edges in \( Found_L(v) \) or \( Found_O(v) \) and empties these lists. Note, that if the high-degree non-star node is not a first-timer (has received \( \langle Expand \rangle \) before), it just forwards the message to its children and over the edges in \( Found_L(v) \) and \( Found_O(v) \) lists.

If the first-timer node is low-degree or a star, it forwards \( \langle Expand \rangle \) to all of its neighbours.

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 \( T - neighbour(v) \). 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 \( T^0 \) consists of the leader node and as its Found lists contain all its neighbours, after the first execution of
**Expand**, if the leader is high-degree, $T_1$ satisfies Invariant 1. 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 $v \in T$, $T$-neighbour $(v)$ contains all the low-degree neighbours of $v$ in $T$.

Expand is called in line 6 of the main algorithm (Algorithm 2) The pseudocode is given in the Expand algorithm.

2) **Search for an outgoing edge to a high degree node**

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

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

Throughout this section we will be using the following fact from Chernoff bounds:

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

We show:

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

3) **Wait to hear from outgoing edges to low-degree external nodes**

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

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

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

The nodes $v \in T$ will eventually receive at least $k/2$ (Low − degree) messages over the outgoing edges. Note that these messages must have been received by $v$ after $v$ executed Expand and added to FoundL(v), for otherwise, these would not be outgoing edges.

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

Each event experienced by a node causes it to flip a coin with probability $\min(c \log n/r, 1)$. If the coin is heads, then a trigger message labelled with the phase number is sent up to the leader. The leader is triggered if it receives at least $(c/2) \log n$ trigger messages for that phase. When the leader is triggered, it begins a new phase. Since there are at least $r$ triggering events, the expected number of trigger messages eventually generated is $(c \log n/r) \geq c \log n$.

**Lemma 3** When the leader receives $(c/2) \log n$ messages with the current phase number, w.h.p., at least $1/128$ of the outgoing edges to low-degree nodes have been added to FoundL lists.

**Proof** 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 \(r/2\) 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 \(k/128\) of the outgoing edges to low-degree nodes have been found.

3.1 Proof of the main theorem

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

Lemma 4 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 1/128 factor (or the algorithm terminates with a spanning tree).

Proof By Lemma 2 there are three possible results from the Search phase. If a sampled outgoing edge to a high-degree node is found, this edge will be added to the \(\textit{Found}\) list of its endpoint in \(T\). If the Search phase ends in fewer than \(2c \log n\) edges found and none of them are to high degree nodes, then w.h.p. these are all the outgoing edges to low-degree nodes, these edges will all be added to some \(\textit{Found}\) lists. If there are no outgoing edges, the algorithm terminates and a spanning tree has been found. If the third possible result occurs, then there are at least \(2c \log n\) outgoing edges, half of which go to low-degree nodes. By Lemma 3, the leader will trigger the next phase and it will do so after at least \(1/128\) of the outgoing edges to low-degree nodes have been added to \(\textit{Found}\) lists.

By Observation 1, all the endpoints of the edges on the \(\textit{Found}\) lists will be added to \(T\) in the next phase, and there is at least one such edge or there are no outgoing edges and the spanning tree has been found. When Expand is called in the next phase, \(T\) acquires a new high degree node in two possible ways, either because an outgoing edge on a \(\textit{Found}\) list is to a high-degree node or because the recursive Expand on outgoing edges to low-degree edges eventually leads to an external high-degree node. In either case, by Invariant 1, \(T\) will acquire a new star node as well as a high-degree node. Also by Invariant 1, all outgoing edges must come from high-degree nodes. Therefore, if no high-degree nodes are added to \(T\) by Expand, then no new outgoing edges are added to \(T\). On the other hand, \(1/128\) of the outgoing edges to low-degree nodes have become non-outgoing edges as their endpoints have been added to \(T\). So we can conclude that the number of outgoing edges to low-degree nodes have been decreased by \(1/128\) factor.

It is not hard to see:

Lemma 5 The number of phases in the algorithm is bounded by \(O(\sqrt{n} \log^{1/2} n)\).

Proof By Lemma 4, 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 1/128; hence, there are no more than \(\log_{4/3} n^2 = O(\log n)\) phases of type B between phases of type A.

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

Lemma 6 The message complexity of the algorithm is \(O(n^{3/2} \log^{3/2} n)\).

Proof The initialization requires \(O(\sqrt{n} \log^{3/2} n)\) messages from \(O(n)\) low-degree nodes and \(O(n)\) messages from each of \(O(\sqrt{n} / \log^{1/2} n)\) stars. In each phase, Expand requires a number of messages which is linear in the size of \(T\) or \(O(n)\), except that newly added low-degree and star nodes send to their neighbours when they are added to \(T\), but this adds just a constant factor to the initialization cost. \(\text{FindAny}\) is repeated \(O(\log n)\) times for a total cost of \(O(n \log n)\) messages. \(\text{ApproxCut}\) requires the same number. The Threshold Detector requires only \(O(\log n)\) messages to be passed up \(T\) or \(O(n \log n)\) messages overall. Therefore, by Lemma 5 the number of messages over all phases is \(O(n^{3/2} \log^{3/2} n)\).

Theorem 1 for spanning trees in connected networks with a pre-selected leader follows from Lemmas 5 and 6.

3.2 Proof of ApproxCut Lemma

Proof Let \(W\) be the set of the outgoing edges. 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 \([2^i], i = 0, \ldots, l\). \(h_z,i(T)\). Also, \(\overrightarrow{h_z}(T) = \bigoplus_{y \in T} \overrightarrow{h_z}(y)\) is computed in the leader. For a fixed \(z\) and \(i\), we have:

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

This probability is at least 1/16 for \(i = l - \lfloor \log |W| \rfloor - 2\) (Lemma 5 of [22]). Note that the value of \(i\) here differs from \(i = l - \lfloor \log |W| \rfloor - 1\) used in [22]. This is because the proof of Lemma 5 in there can be tighter, so the probability of 1/16 still holds even if we decrease the value by one. Remember that \(l = \lfloor 2\log n \rfloor\). Therefore, since \(X_j = \sum_{z=1}^{c \log n} h_{z,j}\) (as defined in the pseudocode and not to be confused with Bernoulli trials), \(E[X_j] = \sum E[h_{z,j}] \geq c \log n / 16\), where \(j = l - \lfloor \log |W| \rfloor - 2\).
Note that \( j = l - \left\lceil \log |W| \right\rceil - 2 \) means that \( \frac{2^j}{2^j + 2} < |W| < \frac{2^j}{2^j + 1} \). Consider \( j = -4 \). Since the probability of an edge being hashed to \( [2^{j-4}] \) is \( \frac{2^j}{2^j + 1} \), we have

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

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:

\[
Pr(h_{z,j-4}(T) = 1 \lor \ldots \lor h_{z,0}(T) = 1) \leq Pr(\exists e \in Whashed to [2^{j-4}] or \ldots or [0])) = Pr(\exists e \in Whashed to [2^{j-4}]) = \frac{1}{32}.
\]

Let \( y_z \) be 1 if \( h_{z,j-4}(T) = 1 \lor \ldots \lor h_{z,0}(T) = 1 \), 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 \( a \),

\[
Pr(X_{j-4} > a \lor \ldots \lor X_0 > a) \leq Pr(Y > a).
\]

From Chernoff bounds:

\[
Pr(X_j < (3/4)c \log n/16) = Pr(X_j < (3/4)E[X_j]) < 1/e^c
\]

and,

\[
Pr(X_{j-4} > (3/2)c \log n/16 \lor \ldots \lor X_0 > (3/2)c \log n/16) < Pr(Y > (3/2)c \log n/16) < \frac{c \log n}{32} < Pr(Y > (3/2)c \log n/32) < Pr(Y > (3/2)E[Y]) < 1/e^c.
\]

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, \( 2|W| \leq 2^{j-\min} \leq 64|W| \). Therefore,

\[
|W|/32 \leq 2^j/64 \leq |W|.
\]

Furthermore, as explained in [22] (Sect. 2 on random odd hash functions), the hash functions can be defined using only two numbers; a multiplier and a threshold which can fit in one message. Therefore, broadcasting each of the \( O(\log n) \) hash functions and computing the corresponding vector takes \( O(n) \) messages; so, the lemma follows.

\[ \square \]

### Algorithm 1 Initialization of the spanning tree algorithm

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

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

1: procedure FINDST-LEADER
2: \quad Leader initially adds all of its incident edges to its \( \text{Found}_L \) list. // By exception leader does not need to differentiate between \( \text{Found}_L \) and \( \text{Found}_O \)
3: \quad \quad \quad \( i \leftarrow 0 \)
4: \quad \quad \quad \( \text{do (Phase} \ i\text{)} \)
5: \quad \quad \quad \quad \( i \leftarrow i + 1 \)
6: \quad \quad \quad Leader calls Expand(). // Expansion
7: \quad \quad \quad \quad \text{// Search and Sampling:}
8: \quad \quad \quad \quad \( \text{counter} \leftarrow 0 \), \( A \leftarrow \emptyset \). \( A \) is the set of sampled outgoing edges.
9: \quad \quad \quad \quad \text{while} \ (\text{counter} < 48c \log n) \ \text{do}
10: \quad \quad \quad \quad \quad \text{if} \ FindAny(\text{\textbackslash} A) \text{ then}
11: \quad \quad \quad \quad \quad \quad \text{leader sends message} \text{ to} \ v \text{ to query} \ v \text{’s degree}, \text{ and sends it to leader.}
12: \quad \quad \quad \quad \quad \quad \text{leader adds} \ (u, v) \text{ to either} \ \text{Found}_L(u) \text{ or} \ \text{Found}_O(u) \text{ based on} \ v \text{’s degree.}
13: \quad \quad \quad \quad \quad \quad \text{end if}
14: \quad \quad \quad \quad \quad \quad \text{counter} \leftarrow \text{counter} + 1.
15: \quad \quad \quad \quad \text{end while}
16: \quad \quad \quad \text{if} \ (|A| = 0) \text{ then}
17: \quad \quad \quad \quad \quad \quad \text{terminate the algorithm} \text{ as there are no outgoing edges.}
18: \quad \quad \quad \quad \text{else if} \ (|A| < 2 \log n \ \text{few edges}) \text{ or } 3(u, v) \in A \ s.t. \ v \text{ is high-degree then}
19: \quad \quad \quad \quad \quad \quad \text{leader starts a new phase to restore Invariant 1.}
20: \quad \quad \quad \quad \quad \quad \text{else (at least half of the outgoing edges are to low-degree nodes) \quad // Wait:}
21: \quad \quad \quad \quad \quad \quad \text{r} \leftarrow \text{ApproxCut}(i/2).
22: \quad \quad \quad \quad \quad \quad \text{leader calls ThresholdDetection}(r).
23: \quad \quad \quad \quad \quad \quad \text{leader waits to trigger and then starts a new phase.}
24: \quad \quad \quad \quad \quad \quad \text{end if}
25: \quad \quad \quad \quad \text{while(true) \quad // the loop breaks only if the algorithm terminates on Line 17}
26: \quad \quad \quad \quad \text{end procedure}

### 3.3 Pseudocode

#### 4 Finding MST with \( o(m) \) asynchronous communication

The MST algorithm implements a version of the GHS algorithm which grows a forest of disjoint subtrees (“fragments”) of the MST in parallel. We reduce the message complexity of GHS by using \( \text{FindMin} \) to find minimum weight outgoing edges without having to send messages across every edge.
Algorithm 3 Given $r$ at phase $i$, this procedure detects when nodes in $T$ receive at least $r/4$ (Low − degree) messages over outgoing edges. $c$ is a constant.

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

Algorithm 4 Leader initiates Expand by sending (Expand) to all of its children. If this is the first Expand, leader sends to all of its neighbours. Here, $x$ is any non-leader node.

1: procedure EXPAND
2: 2: When node $x$ receives an (Expand) message over an edge $(x, y)$:
3: 3: $x$ adds $y$ to $T$-neighbour$(x)$.
4: 4: if $x$ is not in $T$ then
5: 5: The first node that $x$ receives (Expand) from becomes $x$'s parent. // $x$ joins $T$
6: 6: else if $x$ is a high-degree node and $x$ is not a star node then
7: 7: $x$ waits to receive a (Star) over some edge $e$, then adds $e$ to $\text{Found}_0(x)$.
8: 8: It forwards (Expand) over edges in $\text{Found}_1(x)$ and $\text{Found}_0(x)$ (only once in case an edge is in both lists), then removes those edges from the Found lists.
9: 9: else ($x$ is a low-degree or star node)
10: 10: It forwards the (Expand) message to all of its neighbours.
11: 11: end if
12: 12: else ($x$ is already in $T$)
13: 13: if the sender is not its parent, it sends back (Done-by-reject).
14: Else, it forwards (Expand) to its children in $T$, over the edges in $\text{Found}_1(x)$ and $\text{Found}_0(x)$.
15: then removes those edges from the Found lists.
16: 16: end if
17: // 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.
18: 18: 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.
19: 19: The algorithm terminates when the leader receives (Done) from all of its children.
20: end procedure

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

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

But, in order to do this, we require the use of a spanning tree to help synchronize the growth of the fragments.

Note that GHS nodes send messages along their incident edges for two main purposes: (1) to see whether the edge is outgoing, and (2) to make sure that fragments with higher rank are slowed down and do not impose a lot of time and message complexity. Therefore, if we use FindMin instead of having nodes to send messages to their neighbours, we cannot make sure that higher ranked fragments are slowed down. Our protocol works in phases where in each phase only fragments with smallest ranks continue to grow while other fragments wait. A spanning tree is used to control the fragments based on their rank. (See [17] for the original GHS.)

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

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

Then, $r$ broadcasts the message ⟨Proceed, minRank⟩ where minRank is the current minimum rank among all fragments. Any fragment leader that has rank equal to minRank, proceeds to finding minimum weight outgoing edges by calling FindMin on its own fragment tree. These fragments then send a ⟨Connect⟩ message over their minimum weight outgoing edges. When a node $v$ in fragment $F$ (at rank $R$) sends a ⟨Connect⟩ message over an edge $e$ to a node $v'$ in fragment $F'$ (at rank $R'$), since $R$ is the current minimum rank, two cases may happen: (Ranks and identities are updated here.)
1. \( R < R' \): In this case, \( v' \) answers immediately to \( v \) by sending back an \( \langle \text{Accept} \rangle \) message, indicating that \( F \) can merge with \( F' \). (Note that \( v' \) might receive more than one such message at the same time from different nodes, but it still treats them in the same way.) Then, \( v \) initiates the merge by changing its fragment identity to the identity of \( F' \), making \( v' \) its parent, and broadcasting \( F'' \)'s identity over fragment \( F \) so that all nodes in \( F \) update their fragment identity as well. Also, the new fragment (containing \( F \) and \( F' \)) has rank \( R' \).

2. \( R = R' \): \( v' \) responds \( \langle \text{Accept} \rangle \) immediately to \( v \) if the minimum outgoing edge of \( F' \) is \( e \), and also \( F'' \)'s ID is larger than \( F' \)'s ID. The requirement for \( F'' \)'s ID being larger is only to break symmetry in case both fragments send/receive their \( \langle \text{Connect} \rangle \) message at the same time. Then, \( F \) merges with \( F' \) as mentioned in rule 1, and the new fragment will have \( F'' \)'s identity. Also, both fragments increase their rank to \( R' + 1 \).

Otherwise, \( v' \) does not respond to the message until \( F'' \)'s rank increases. Once \( F' \) increased its rank, it responds via an \( \langle \text{Accept} \rangle \) message, fragments merge, and the new fragment will update its rank to \( R' \).

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

When all fragments finish their merge at this phase they have increased their rank by at least one. Now, it is time for \( r \) to 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, \( \langle \text{Done} \rangle \) messages are convergecast in \( T \). Nodes that were at minimum rank send up to their parent in \( T \) a \( \langle \text{Done} \rangle \) message only after they increased their rank and received \( \langle \text{Done} \rangle \) messages from all of their children in \( T \).

As proved in Lemma 7, this algorithm uses \( \tilde{O}(n) \) messages.

Lemma 7 Having access to a spanning tree, \( \text{FindMST} \) uses \( O(n \log^3 n / \log \log n) \) messages and finds the MST w.h.p.

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

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

From Lemma 7, Theorem 1 for minimum spanning trees follows.

5 Minimum spanning forest

In this section we provide an algorithm to find a minimum spanning forest when the input graph is disconnected. The algorithm still computes an MST if the graph is connected. Unlike the old algorithm in Sect. 3, the algorithm presented here does not require a pre-selected leader. In other words, this algorithm can also be used to find the MST when a leader is not known ahead of time.

In order to simplify the description of the algorithm, we use the alternative way for counting events as mentioned in Sect. 2.3. In fact, we assume that nodes send back acknowledgment messages when they receive a \( \langle \text{Low-degree} \rangle \) message. Therefore, we will not need the \( T \)-neighbour data structure.

The main idea for our algorithm is to deal with components based on their sizes. A component is considered big if it has at least \( \sqrt{n} \log^{3/2} n \) nodes, and small otherwise. With high
probability, there is a star node inside all big components. However, a small component does not necessarily contain a star node. Moreover, small components cannot have a high-degree node since a node is high-degree if it has at least \( \sqrt{n \log^{3/2} n} \) neighbours.

Nodes, however, do not know the size of the component they belong to. To overcome this obstacle, we are going to have two types of nodes that run different algorithms as follows.

- Star nodes run the FindST-leader protocol.
- Low-degree nodes that have not been selected as a star-node (called low-degree non-star nodes) run the GHS algorithm.

Note that all nodes still send the initial \( \langle \text{Low-degree} \rangle \) and \( \langle \text{Star} \rangle \) messages.

Having nodes that run different algorithms in the same component will eventually result in a situation where a node is asked to participate in more that one protocol (initiated by different nodes). We call such an event a collision. We will later explain how we can handle these collisions so that a node can decide which protocol it should be a part of.

The intuition behind running different protocols is as follows. If a component is small, all of its nodes are low-degree. Moreover, as mentioned before, a small component may not have a star node. Therefore, a small component should be able to compute the MST on its own. Since there is no high-degree node in a small component, the total number of edges in the component is \( O(n^{3/2} \log^{3/2} n) \). Therefore, all nodes can run the GHS algorithm which uses messages proportional to the number of edges in the component, and find the MST directly. However, to ensure that the message complexity still remains \( o(n) \) we will not allow any high-degree node to participate in a GHS protocol. Therefore, the GHS protocol will complete successfully if and only if it is running in a component with no high-degree and no star node.

On the other hand, if a component is big it has a star node. Furthermore, for such a component, we can have a star node that acts as the leader of that component. Following the Find-ST protocol, the nodes of a big component will first find a spanning tree over the component’s subgraph, and then compute the minimum spanning tree.

As for the message complexity, in case of small components the total of edges is bounded roughly by \( O(n) \) and the worst case is to have \( O(\sqrt{n}) \) small components. (By adding to the number of components the total of the edges in such components will drop.) Therefore, using GHS on small components, which has message complexity proportional to the number of edges, is harmless. Moreover, if we are dealing with big components, the message complexity is bounded by our previous algorithm for MST.

In order to implement this idea, we need a set of rules to handle the collision between protocols. Here, we do not consider an exchange of messages between two nodes that are running the GHS protocol to be a collision. The GHS protocol can deal with these cases itself.

In this section, we assume that fragments which grow using the FindST-leader, each have a fragment ID. This ID is the node ID of the star node that initiated the protocol. In order to resolve collisions, there are three cases that we should consider. Imagine that node \( a \) sends a message regarding the protocol it is running to node \( b \). The collision resolution rules are as follows:

1. When \( a \) is running the FindST-leader protocol and \( b \) is running the GHS protocol: in this case \( b \) will stop participating in the GHS protocol and will become a part of the FindST-leader protocol that \( a \) is running.
2. When \( a \) is running GHS and \( b \) is running FindST-leader: \( b \) will never respond to the message; therefore, \( a \)’s GHS protocol will never terminate.
3. When \( a \) and \( b \) are both running the FindST-leader protocols with different fragment IDs: in this case if \( a \)’s fragment ID is higher than \( b \)’s fragment ID, it can proceed and take over node \( b \). Otherwise, \( b \) tells \( a \) that \( a \)’s fragment should stop running the protocol. (We will elaborate on this later.)

Having this set of rules, we can always preserve the following invariant:

**Invariant 2:** In a component with at least one star node, the star node with maximum ID (among all star nodes in that component) will become responsible for computing the MST.

**Description of the algorithm:** First, all nodes follow the initialization protocol (Algorithm 1) as in the old algorithm. Then, each star node runs the FindST-Leader protocol to expand its fragment tree via expansion, search and sampling, and waiting where all of the messages exchanged during these three phases are labeled by the fragment ID, i.e., the ID of the star node that is the leader of the fragment.

All low-degree non-star nodes run the GHS protocol. Note that our assumption is that all nodes are awakened simultaneously.

If a component is composed of only low-degree non-star nodes, these nodes, which are all running the GHS protocol, will find the minimum spanning tree in that component directly. However, in order to obtain sublinear message complexity, we ensure that no high-degree node participates in a GHS protocol. To this end, when a high-degree node receives a message from a node running the GHS protocol, it will never respond to the message; hence, delaying the protocol forever. Note that existence of a high-degree node guarantees
the existence of a star node w.h.p. Therefore, a star node will eventually find the MST in this component.

As mentioned before, in order to resolve collisions we require each fragment to have an ID. Initially, each star node belongs to a singleton fragment with an ID equal to that of the star node itself. Although initially non-star nodes do not belong to any fragment, we assume that they belong to a fragment with an ID of 0. This assignment allows fragments that are lead by a star node to take over the nodes that have not joined a fragment yet, using the third collision resolution rule (prioritizing higher fragment IDs).

It is worth mentioning that a node may be contacted by a node in a different fragment through initialization messages, messages for querying the degree, or expansion messages. We do not label initialization and degree-querying messages. Therefore, a node that receives these types of messages will not consider it a collision. This will not affect the analysis of the algorithm.

In order to enforce the collision resolution rules and to keep the message complexity sublinear, we use a new method for the expansion phase. Expand-MultiLeader (Algorithm 7) allows the fragments IDs to be updated without using a lot of messages. Expand-MultiLeader will replace the old Expand algorithm in Sect. 3.

Implementation of Expand-Multileader The expansion messages used in Expand-MultiLeader are accompanied by the fragment identities ((Expand, tID)). Every node v keeps a variable vID which is the ID of the fragment it belongs to at the moment. We also use vID to refer to the whole fragment whose ID is equal to vID. As mentioned before, for all non-star nodes vID is initialized to 0. We define the reject list data structure as follows:

Reject(v): Contains the list of edges over which v rejected an expansion message from another fragment for the first time. An edge is only added to this list if the reason of rejection is lower ID value of the other fragment. We emphasize that the edge is added only on the first occurrence of this event for each fragment.

When node x (currently in fragment xID) receives an expansion message of (Expand, tID) from some node t in fragment tID, the algorithm handles this message by comparing xID and tID. There are three cases:

1. tID < xID: Fragment xID refuses to join fragment tID, and signals fragment tID to stop running the FindST-Leader protocol. To this end, x responds to t’s expansion message by sending back ⟨Rejected-lower-ID, tID⟩ (line 4). The leader of fragment tID will eventually be notified that its fragment has contacted a fragment with higher ID when the (Rejected-lower-ID,tID) message reaches the leader via a convergecast (line 28). This causes fragment tID to stop running the FindST-Leader protocol (line 27).

Moreover, fragment xID should take over the fragment tID. However, since fragment xID may not be running an expansion at the moment, node x has to remember to take over fragment tID in the next expansion. For this, x adds the edge (x, t) to Reject(x) only if this is the first time that x sees an expansion message with identity tID. The reason for the first time condition is that since all nodes of fragment tID are already connected via a tree structure, having access to at least one of those tree nodes would suffice to allow fragment xID to take over the whole fragment later. During the next expansion, x will forward the expansion messages over the edges in Reject(x), as well.

Notice that in this case x responds to an expansion message immediately even if it is participating in another expansion at the moment. This prevents fragments from waiting in a loop for each others’ expansion to finish.

2. tID > xID: In this case, x joins fragment tID by updating its fragment identity from xID to tID and setting the first node from which it has received (Expand, tID) as its parent (line 8).

Notice that unlike the previous case, x, if participating in an expansion, first waits to hear the response to all expansion messages it has sent, then updates its children, and only after that it handles the (Expand, tID) message. This will ensure that the expansion messages are always forwarded over a tree structure when it is updating the ID of another fragment. Figure 1 shows that without this wait, the algorithm needs Ω(n^2) messages to finish. Imagine that in Fig. 1 the nodes have IDs v_1 > v_2 > ... > v_n. Let C be the set of nodes {v_{n/2}, ..., v_n} and all nodes in C are low-degree and have √n edges to other nodes in C. There have to be Θ(√n log n) star nodes on the path from v_1 to v_{n/2}, w.h.p. Let us name these star nodes s_1 to s_k from left to right, where k = Θ(√n log n). Assume s_k’s expansion message reaches to v_{n/2} and is forwarded to the nodes in C, but before any expansion message is answered, s_{k-1}’s expansion message reaches to v_{n/2}. Therefore, if v_{n/2} does not wait for the last expansion to finish, it has to forward (Expand, s_{k-1}) to nodes in C again. Besides, these are all low-degree nodes and will forward the message over all of their incident edges which requires Ω(n^{3/2}) messages. Repeating this for s_{k-2} to s_1 will result in Ω(n^2) messages.

Now, there are two sub-cases for Case 2, based on whether or not x is part of some fragment upon receiving (Expand, tID):

2.a If x is part of some other fragment X (line 9), fragment X should also become a part of fragment tID. So, x forwards (Expand, tID) to all of its neighbours in X except the one that it has received the expansion message from. This will result in all neighbours of x to do the same recursively and all nodes in fragment X will
update their fragment identity to $tID$ and become part of fragment $tID$. Moreover, $x$ forwards the expansion message over edges in $Found(x)$ and $Reject(x)$.

2.b If $x$ did not belong to any fragment before (line 11), the expansion message is handled just like the old Expand algorithm. Note that in this case $Reject(x)$ is an empty list. If the node is high-degree and is not a star node (line 12), it waits to receive a star message, updates its $Found(x)$ list, and then forwards the message to the edges in its Found lists. Else, if $x$ is a low-degree or star node, it forwards the message to all of its neighbours in $G$.

3. $tID = xID$: In this case, $x$ and $t$ are in the same fragment. If $t$ is $x$’s parent in fragment $tID$, $x$ forwards the expansion message to its children in the fragment, over the edges in Found lists and also over the edges in $Reject(x)$. Otherwise, the sender of the message is not $x$’s parent; therefore, $x$ responds back $(Reject\text{-}same\text{-}tree, tID)$.

Once the fragment identities are compared and the expansion message is handled, $x$ waits to hear back from all the nodes it has forwarded to. Then, $x$ removes from $Reject(x)$ and the Found lists any edges it has forwarded over. Afterwards, $x$ updates its children to be all of the nodes that it has received $(Accept, tID)$ from (line 26) in this expansion. A node only sends up an $(Accept, tID)$ message in response to $(Expand, tID)$ if either it is a leaf in the fragment tree or it has not received any $(Rejected\text{-}lower\text{-}ID, tID)$ from the nodes it has sent to in this expansion. If a node receives a $(Rejected\text{-}lower\text{-}ID, tID)$, it sends up $(Rejected\text{-}lower\text{-}ID, tID)$ to its parent. As a result, if any node in the fragment is rejected because of its low ID, the leader will be notified and will stop running the FindST-Leader protocol (line 27).

### 5.2 Analysis

As before, the overall number of initialization messages is still bounded by $O(n^{3/2} \log^{3/2} n)$. Also, GHS spends asymptotically the same number of messages as the number of edges in the graph running the protocol. Since no high-degree node participates in a GHS, the overall number of messages used for the GHS protocol is bounded by $O(n^{3/2} \log^{3/2} n)$. Now, we show that the overall number of messages used for running FindST-Leader is also bounded by $O(n^{3/2} \log^{3/2} n)$.

We define a successful expansion to be one that completes without any node receiving a $(Rejected\text{-}lower-ID)$ message, which does not cause the leader to stop the FindST-Leader protocol.

**Lemma 8** The number of successful expansions over all fragments is $O(\sqrt{n} \log^{1/2} n)$.

**Proof** Any fragment can have at most $O(\log n)$ successful expansions before it causes the number of fragments to reduce, i.e., it causes two fragments with different identities to merge. The reason is that as we know from Lemma 5, $O(\log n)$ expansions suffice for a fragment to find a new star (and a new fragment) with high probability, which will cause at least one fragment to stop running the protocol. Since the initial number of fragments is $O(\sqrt{n} \log^{1/2} n)$, the overall number of successful expansions is bounded by $O(\sqrt{n} \log^{1/2} n)$. □
In any fragment, each successful expansion is followed by one search and sampling and at most one wait where each of which these parts uses \( O(n \log n) \) messages (proof in Lemma 6). Therefore, from Lemma 8, the overall number of messages used in the algorithm apart from expansion is bounded by \( O(n^{3/2} \log^{3/2} n) \).

Finally, we prove in the following claims that the overall number of messages used for expansion is bounded by the same amount.

**Claim 1** The number of forwards over edges in the Found lists, over all nodes, is bounded by \( O(n^{3/2} \log^{3/2} n) \).

**Proof** After an \( \langle \text{Expand}, t ID \rangle \) is forwarded over an edge in the Found list, regardless of the result, that edge is removed from the Found list (line 24). Therefore, we only need to show that the overall number of edges added to Found lists of all nodes is bounded by \( O(n^{3/2} \log^{3/2} n) \). An edge is added to the Found list of a node either because of a \( \langle \text{Low-degree} \rangle \) message, a \( \langle \text{Star} \rangle \) message, or an edge found by FindAny in FindST-Leader. The first two types of messages are \( O(\sqrt{n}) \) successful expansions over all fragments and each of them is followed by at most \( O(\log n) \) calls to FindAny. Therefore, FindAny is responsible for at most \( O(\sqrt{n}) \) edges in the Found lists and the claim follows. \( \square \)

**Claim 2** The number of forwards over edges in the Reject lists, over all nodes, is bounded by \( O(n^{3/2} / \log^{1/2} n) \).

**Proof** A node \( x \) that receives \( \langle \text{Expand}, t ID \rangle \) over edge \( e \), and rejects it because \( t ID < x ID \), may only add \( e \) to its Reject list if this is the first time it receives an expansion message with identity \( t ID \). Therefore, a node \( x \) adds \( O(\sqrt{n} / \log^{1/2} n) \) (the initial number of fragments) edges to its Reject list over the course of the algorithm. Also, upon forwarding over some edge \( e \) in the Reject list, the node removes \( e \) from the list; therefore, the overall number of such forwards is \( O(n^{3/2} / \log^{1/2} n) \). \( \square \)

**Claim 3** The overall number of expansion messages forwarded, when a node receives an expansion message for the first time, is bounded by \( O(n^{3/2} \log^{3/2} n) \) over all nodes.

**Fig. 2** Example of worst case time and message complexity

\[
P_L = P_{n/4}, \quad \ldots, \quad v_{\frac{n}{4} + 2}, \quad v_{\frac{n}{4} + 1}, \quad v_{\frac{n}{4}}, \quad \ldots, \quad v_{\frac{n}{4} - 1}, \quad v_1, \quad \ldots, \quad v_{\frac{n}{2}}, \quad v_{\frac{n}{2} - 1}, \quad \ldots, \quad v_{\frac{n}{2} + 2}, \quad P_R = P_{n/4}
\]

\( K_{n/2} \)

Finally, the following claim will bound the number of forwards after the first time and allows us to bound the message complexity of the algorithm.

**Claim 4** The number of forwards over branches of the old fragment in case that \( t ID > x ID \) (line 10), and also the number of forwards over branches of the current fragment in case that \( t ID = x ID \) (line 21) is bounded by \( O(n^{3/2} \log^{1/2} n) \).

**Proof** Forwards over incident edges in the old fragment when \( t ID > x ID \): Over the whole algorithm, \( O(\sqrt{n} / \log^{1/2} n) \) leaders may have grown their fragments to a tree of size \( O(n) \). Assume that each node had a set of incident edges (including the one to its parent) in each of the fragments it belonged to over the course of the algorithm. Let \( C \) be the collection of all of the incident edges of all nodes in these fragments. Note that in \( C \), the same edge is repeated twice for each of its endpoints and could also be repeated up to \( O(\sqrt{n} / \log^{1/2} n) \) times as part of different fragments. Since there are \( O(\sqrt{n} / \log^{1/2} n) \) fragment trees and each of them has size at most \( n \), size of \( C \) is the sum of the degrees of all nodes in these trees which is bounded by \( O(n^{3/2} / \log^{1/2} n) \).

Consider node \( x \) that is part of a fragment \( X \) with identity \( x ID \). When \( x \) receives \( \langle \text{Expand}, t ID \rangle \) it forwards this message to its neighbours in \( X \) except the one it has received the expansion message from (line 10). Whether the result of the forward is accept or reject, this is the last time that \( x \) forwards over a part of the fragment \( X \), as a node in \( X \). The reason is that right after this \( x \) joins fragment \( T \) and updates its children (line 26). Now, any future expansion message that \( x \) receives, will be forwarded as a part of fragment \( T \) (and not \( X \)). Therefore, the overall number of expansion messages that nodes forward over their incident edges, immediately after updating their identity, could not exceed the size of \( C \) which is \( O(n^{3/2} / \log^{1/2} n) \).

**Forwards to children when \( t ID = x ID \):** In this case, \( x \) only forwards the message if the sender is its parent in \( T \). In fact, if
A minimum spanning forest can be constructed using O(\(\sqrt{n} \log \frac{1}{n}\)) the overall number of forwards over incident edges when \(tID = xID\) is O(n^{3/2} \log^{1/2} n).

Putting together Claims 1 to 4, we obtain the following lemma.

\textbf{Lemma 9} Expand-MultiLeader will result in no more than O(n^{3/2} \log^{3/2} n) forwards of (Expand, 1ID) messages over all fragments, before a spanning forest is constructed.

Therefore, we have the following theorem on building a minimum spanning forest in general input graphs.

\textbf{Theorem 2} A minimum spanning forest can be constructed using O(n^{3/2} \log^{3/2} n) messages with asynchronous communication.

\textbf{A note on analysis:} Here, we provide an example that shows that even when all star nodes are running the protocol in parallel, the time and the message complexity could still be as high as \(\Theta(n^{3/2} \log^{3/2} n)\). Since we do not assume that we know initially whether the graph G is a connected graph or not, our example here is a connected graph. Consider the graph in Fig. 2 where a complete graph \(K_{n/2}\) is connected to two path graphs \(P_L/4\) on the left and on the right, named \(P_L\) and \(P_R\) respectively. Assume that node IDs are \(v_1 > v_2 > \ldots > v_n\). There have to be \(\Theta(\sqrt{n} \log \frac{1}{n})\) star nodes in \(P_L\) and in \(K_{n/2}\) w.h.p. As a result, \(\Theta(n^{3/2} \log^{3/2} n)\) messages are guaranteed since the star nodes in \(K_{n/2}\) send initialization messages to all of their neighbours.

We show that time complexity is \(\Theta(n^{3/2} \log^{3/2} n)\), as well. Let the star nodes in \(P_L\) be \(s_1, s_2, \ldots, s_k\) from left to right, where \(k = \Theta(\sqrt{n} \log \frac{1}{n})\). Let \(\delta = 1\) time step be the max delay. Suppose \(s_k\)’s expansion messages go all the way to the right in one time step, and span \(K_{n/2}\) and \(P_R\). Now, \(s_k−1\)’s expansion reaches to \(s_k\) and updates the identity of all of the nodes on the right in \(O(n)\) time steps. Meanwhile, \(s_k−2\) is expanding to the right but according to the algorithm (line 6) \(s_k−1\) waits to finish its expansion before passing on the expansion of \(s_k−2\), so \(s_k−2\) waits \(O(n)\) time steps. Similarly each \(s_k−i\) (\(1 \leq i \leq k−1\)) must wait \(O(n)\) time for \(s_k−i+1\) to finish its expansion for a cost of \(O(n)\). Moreover, each of these expansions is followed by a search and sampling that takes \(O(n \log n)\) time; hence, time complexity of \(\Theta(n^{3/2} \log^{3/2} n)\).

\textbf{6 Conclusion}

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

1. Ahn, K.J., Guha, S., McGregor, A.: Graph sketches: sparsification, spanners, and subgraphs. In: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 5–14. ACM (2012)

2. Awerbuch, B.: Complexity of network synchronization. J ACM 32(4), 804–823 (1985)

3. Awerbuch, B.: 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, pp. 230–240. ACM (1987)

4. Awerbuch, B., Goldreich, O., Vainish, R.: On the Message Complexity of Broadcast: A Basic Lower Bound. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge (1987)

5. Awerbuch, B., Goldreich, O., Vainish, R., Peleg, D.: A trade-off between information and communication in broadcast protocols. J ACM 37(2), 238–256 (1990)

6. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: A time-optimal self-stabilizing synchronizer using a phase clock. IEEE Trans Dependable Comput 4(3), 180–190 (2007)

7. Awerbuch, B., Peleg, D.: Network synchronization with polylogarithmic overhead. In: 31st Annual Symposium on Foundations of Computer Science, 1990. Proceedings., pp. 514–522. IEEE (1990)

8. Awerbuch, B., Ting, H.: An almost linear time and o (nlogn+e) messages distributed algorithm for minimum-weight spanning trees. In: 26th Annual Symposium on Foundations of Computer Science, pp. 257–266. IEEE (1985)

9. Elkin, M.: A faster distributed protocol for constructing a minimum spanning tree. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 359–368. Society for Industrial and Applied Mathematics (2004)

10. Elkin, M.: An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. SIAM J. Comput. 36(2), 433–456 (2006)

11. Elkin, M.: Synchronizers, spanners. In: Kao, Ming-Yang (ed) Encyclopedia of Algorithms, pp. 1–99. Springer (2008)

12. Elkin, M.: Distributed exact shortest paths in sublinear time. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, pp. 757–770. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3055399.3055452

13. Elkin, M.: A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. arXiv preprint arXiv:1703.02411 (2017)

14. Emek, Y., Korman, A.: Efficient threshold detection in a distributed environment. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 183–191. ACM (2010)

15. Faloutsos, M., Molle, M.: Optimal distributed algorithm for minimum spanning trees revisited. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 231–237. ACM (1995)

16. Gafni, E.: Improvements in the time complexity of two message-optimal election algorithms. In: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 175–185. ACM (1985)

17. Gafni, E.: Improvements in the time complexity of two message-optimal election algorithms. In: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 175–185. ACM (1985)

18. Garay, J.A., Kutten, S., Peleg, D.: A sublinear time distributed algorithm for minimum-weight spanning trees. SIAM J. Comput. 27(1), 302–316 (1998)

19. Ghaffari, M., Kuhn, F.: Distributed MST and broadcast with fewer messages, and faster gossiping. In: Schmid, U., Widder, J. (eds.) 32nd International Symposium on Distributed Computing (DISC 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 121, pp. 30:1–30:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.DISC.2018.30. http://drops.dagstuhl.de/opus/volltexte/2018/9819

20. Gnyr, R., Pandurangan, G.: Time-Message Trade-Offs in Distributed Algorithms. In: Schmid, U., Widder, J. (eds.) 32nd International Symposium on Distributed Computing (DISC 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 121, pp. 32:1–32:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.DISC.2018.32. http://drops.dagstuhl.de/opus/volltexte/2018/9821

21. Kaplan, B.M., King, V., Mountjoy, B.: Dynamic graph connectivity in polylogarithmic worst case time. In: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1131–1142. Society for Industrial and Applied Mathematics (2013)

22. King, V., Kutten, S., Thorup, M.: 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, pp. 71–80. ACM (2015)

23. Kutten, S., Pandurangan, G., Peleg, D., Robinson, P., Trehan, A.: On the complexity of leader election. J. ACM 62(1), 7 (2015)

24. Kutten, S., Peleg, D.: Fast distributed construction of k-dominating sets and applications. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 238–251. ACM (1995)

25. Mashreghi, A., King, V.: Time-communication trade-offs for minimum spanning tree construction. In: Proceedings of the 18th International Conference on Distributed Computing and Networking, p. 8. ACM (2017)

26. Mashreghi, A., King, V.: Brief announcement: faster asynchronous mst and low diameter tree construction with sublinear communication. In: 33rd International Symposium on Distributed Computing (DISC 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)

27. Mashreghi, A., King, V.: Faster asynchronous mst and low diameter tree construction with sublinear communication. arXiv preprint arXiv:1907.12152 (2019)

28. Pandurangan, G., Robinson, P., Squizzato, M.: A time-and-message-optimal distributed algorithm for minimum spanning trees. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, pp. 743–756. ACM (2017)

29. Peleg, D., Rubinovich, V.: A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. SIAM J. Comput. 30(5), 1427–1442 (2000). https://doi.org/10.1137/S0097539700369740

30. Peleg, D., Ullman, J.D.: An optimal synchronizer for the hypercube. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 77–85. ACM (1987)

31. Sarma, A.D., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed verification and hardness of distributed approximation. SIAM J. Comput. 41(5), 1235–1265 (2012)

32. Singh, G., Bernstein, A.J.: A highly asynchronous minimum spanning tree protocol. Distrib. Comput. 8(3), 151–161 (1995)