Distributed minimum spanning tree problem

Juho-Kustaa Kangas

24th November 2012

Abstract

Given a connected weighted undirected graph, the minimum spanning tree problem asks for a spanning subtree with the minimum sum of edge weights. We consider a distributed version of the problem in which each node of the graph is a processor knowing initially only the weights of incident edges. To solve the problem the nodes exchange messages with each other until each node knows, which incident edges are in the spanning tree. We present and analyze a classical algorithm solving the problem.

1 Introduction

A spanning tree of a connected, undirected graph is a connected acyclic subgraph that contains all nodes of the graph. For a graph with weighted edges, we define the minimum spanning tree (MST) as the spanning tree with the minimum sum over the weights of its edges. Finding the minimum spanning tree of a given graph is one of the oldest problems in computer science and a central problem in combinatorial optimization and distributed computing. Commonly used greedy centralized algorithms (e.g. Kruskal's [3] and Prim's [4] algorithms) solve the problem in time $\mathcal{O}(E \log V)$, where V and E are the sets of nodes and edges respectively (by convention, we write V and E short for $|V|$ and $|E|$ here). For a more general treatment of the problem and references to some faster algorithms, see e.g. [1].

In this report we consider the MST problem in a distributed setting. Specifically, we consider an asynchronous message passing model in which each node of the graph can do simple processing and send messages to its neighbors. In the beginning each node knows only the weights of its incident edges. The nodes must then cooperate by communicating with each other, following an identical local algorithm, until each node knows which of its incident edges are part of the MST. The cost of computation is measured in the amount of communication required. In general, we are interested in the number of messages that nodes must exchange (message complexity) as well as to what extent the process can run in parallel (time complexity).

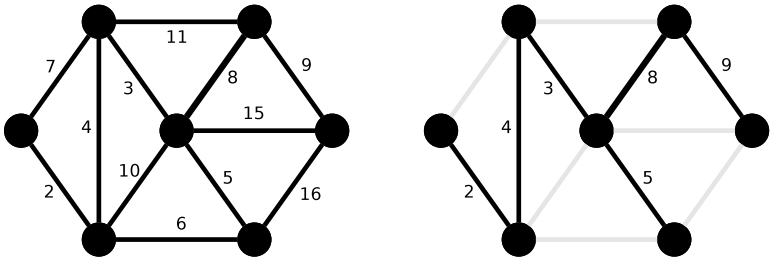

Figure 1: A weighted graph and its unique minimum spanning tree.

The paper is organized as follows: In section 2 we give a more formal definition of the problem and the model of computation. We also prove some general properties of minimum spanning trees to motivate a general approach to solving the problem. In section 3 we present a classical distributed algorithm by Gallager et al. solving the problem with $\mathcal{O}(E + V \log V)$ messages and $\mathcal{O}(V \log V)$ time. We address correctness issues and give an analysis of the complexity in sections 4 and 5, respectively. We assume the reader is familiar with basic terminology of graph theory.

2 Preliminaries

2.1 Definitions and notation

A *graph* is an ordered pair (V, E), where V is a finite set of *nodes* and $E \subseteq V^{(2)}$ is a set of *edges* between nodes. A sequence of nodes is called a *path* if there's an edge between all consecutive nodes. If there's also an edge between the first and last node, the sequence is called a *cycle*. A graph is *connected* if there's a path between all pairs of nodes. A *tree* is a connected graph without any cycles. If $E' \subseteq E$ and $T = (V, E')$ is a tree, then T is called a *spanning tree* of (V, E). In a *weighted* graph, we associate a *weight* $w(e)$ for each edge $e \in E$. For simplicity, we treat weights as being real numbers, though they can be any objects with similar properties.

Throughout the report we consider a given connected graph with weighted edges. If $T = (V, E)$ is a spanning tree of the graph, we say the weight of T is $w(T) = \sum_{e \in E} w(e)$, the sum of weights over the edges in E. If T has a minimum weight over all spanning trees of the graph, it's called a *minimum spanning tree* (figure 1). A minimum spanning tree isn't necessarily unique as several spanning edge sets might have the same weight. In section 2.2 we show that if edge weights are distinct, i.e., $w(e) \neq w(e')$ for all $e \neq e'$, then the MST is unique. If $F = (V', E')$ is a tree and $V' \subseteq V$ and $E' \subseteq E$, then we say F is a *fragment* of T. Further, any edge $\{u, v\}$ of the graph with $u \in V'$ and $v \notin V'$ is called an *outgoing edge* of F.

To simplify the manipulation of graphs, we extend common notations as follows: Let $T = (V, E)$ and $T' = (V', E')$ be subgraphs and $e = \{u, v\}$ an edge. The graph produced by adding e and its endpoints to T is denoted by $T \cup e = (V \cup \{u, v\}, E \cup \{e\})$. Removing e from T is denoted by $T \setminus e = (V, E \setminus \{e\})$. We write $e \in T$ if $e \in E$, otherwise $e \notin T$. Finally, the combination of T and T' is denoted by $T \cup T' = (V \cup V', E \cup E')$.

2.2 Properties of minimum spanning trees

Minimum spanning trees have several well known properties that are exploited by algorithms (see e.g. [1, 2]). We will now introduce some of them.

Lemma 1. *Let $T = (V, E)$ be a tree and e an edge in $V^{(2)}$ but not in E. Then $T \cup e$ contains a cycle. If e' is an edge in the cycle, then $(T \cup e) \setminus e'$ is a tree.*

Essentially the lemma says that adding an edge to a tree produces a cycle and removing any edge in the cycle produces a tree. This should be intuitively clear so we omit the proof.

Theorem 2. *Let F be a fragment of an MST and let e be a minimum-weight outgoing edge of F. Then, $F \cup e$ is also a fragment of an MST.*

Proof. Let T be an MST having F as a fragment. If $e \in T$, the claim follows trivially. Otherwise $T \cup e$ contains a cycle and an edge $e' \neq e$ in the cycle is an outgoing edge of F. By lemma 1 $T' = (T \cup e) \setminus e'$ is a spanning tree. Since by definition $w(e) \leq w(e')$, it follows that $w(T') = w(T) + w(e) - w(e') \leq w(T)$. Therefore T' is an MST and by construction $F \cup e$ is its fragment. □

Theorem 3. *If a graph has distinct edge weights, then its MST is unique.*

Proof. Assume to the contrary that there are two distinct MSTs, T and T'. Let e be the minimum-weight edge contained in exactly one of the trees. By symmetry assume $e \in T$. Since $e \notin T'$, the graph $T' \cup e$ necessarily contains a cycle. Since T is a tree, at least one of the edges in the cycle is not in T. Let e' be such an edge. By lemma 1 $T'' = (T' \cup e) \setminus e'$ is spanning tree. Since e' is in exactly one of T and T', and e has the minimum weight among such edges, it follows that $w(e) < w(e')$. Then, $w(T'') = w(T') + w(e) - w(e') < w(T')$, which is a contradiction since T' has minimum weight. □

2.3 Model of computation

We consider the problem of finding an MST in a common asynchronous message passing model. In this setting each node of the graph is a computation unit, having a simple state of a few variables and running a local algorithm identical to those of all other nodes. We assume that all nodes "wake up" simultaneously to start the algorithm. At this initial stage each node is only

aware of its incident edges and their weights. A node can communicate with its neighbors by sending and receiving short messages over these edges. By "short" we mean in the order of $\mathcal{O}(\log V)$ bits. The algorithm terminates once each node has determined which of its incident edges are part of the MST and will no longer send messages to other nodes.

Communication is asynchronous: Each node may send a message to any of its neighbors at any time, independently of other nodes. Several messages can be transmitted simultaneously on an edge, in both directions. Each message sent is received without error, after a delay that is unknown but bounded by some constant which does not depend on the graph. As a consequence, a message sent later than another message may still be received earlier. However, messages sent on the *same edge* in the *same direction* always arrive in the same order.

Each message received by a node is appended to a first-in-first-out queue, waiting to be processed. Whenever the queue is non-empty, the node picks the message at the head of the queue and does some processing according to its local algorithm. Depending on the message, the processing may consist of simple modifications to the node's internal state as well as sending new messages to its neighbors. If a node determines it cannot process the message yet, it may also reappend the message at the end of the queue.

While a node has no messages to process, it remains idle. The only spontaneous processing happens when all nodes simultaneously run some initiate procedure to start the algorithm. We assume this to be our initial state and are not concerned with how the nodes reach it.

2.4 Distinct edge weights

In order to solve the problem using the model described, we have to assume that the edge weights are distinct. This can be shown by considering a simple case with three nodes, all connected to each other and all edges having equal weights. Since the situation is symmetric, there is no deterministic algorithm to resolve which of the two edges should constitute the MST. Any solution can lead to all nodes continuously attempting to choose different edges, resulting in a cycle. If nodes are allowed to make random choices, then the expected number of attempts is likely to be low, but even then the process could go on indefinitely.

If we modify the model to allow distinct identifiers for nodes, then it is possible to break ties by appending those identifiers to edge weights. Sending the node identifiers before starting the main algorithm requires at most $2E$ messages. There are also some modifications to the algorithm we describe in section 3 that work without doing the extra communication.

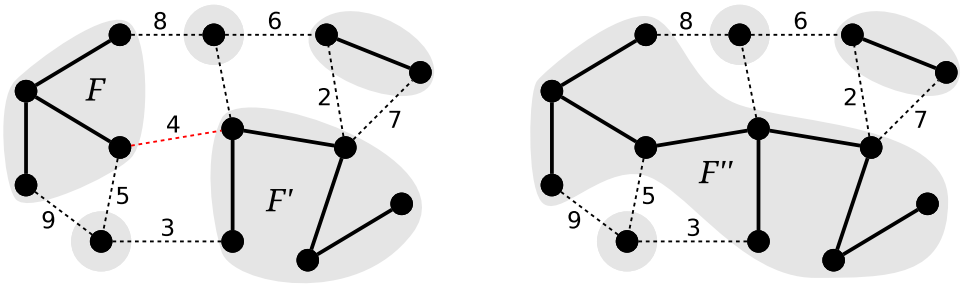

Figure 2: A collection of MST fragments (left). Following the generic scheme, we can choose an arbitrary fragment F and add its minimum-weight outgoing edge (in red) to the MST. By combining the fragments joined by the edge (right), we get a new larger fragment of the MST.

3 Algorithm

3.1 High level description

We present an algorithm by Gallager et al. [2] finding the MST under the specified model. The algorithm follows a greedy scheme in which known fragments of the MST are incrementally combined into larger fragments by adding minimum-weight edges between them. It can be formulated as:

1: For each $v \in V$, construct a trivial fragment $(\{v\}, \emptyset)$.
2: **while** there is more than one fragment **do**
3: Choose a fragment F. Denote by e the minimum-weight outgoing edge of F and by F' the fragment at the other end of e.
4: Combine F and F' into a new fragment $F'' = (F \cup e) \cup F'$.
5: **end while**
6: The single remaining fragment is the MST.

To see that this works, consider line 4. By Theorem 2, $F \cup e$ is a fragment of some MST. By Theorem 3 (since we assume edge weights to be distinct) the MST is unique. Since fragments $F \cup e$ and F' share a node, F'' is connected and therefore also a fragment of the MST (figure 2). The algorithm stops when there is only one fragment left that contains the whole MST.

The generic scheme is shared by many centralized algorithms, which differ mainly in how the fragment is chosen on line 3. Kruskal's algorithm [3], for example, always picks the minimum-weight outgoing edge over all fragments, while Prim's algorithm [4] keeps extending a single large fragment. By using suitable data structures both can be made to run in $\mathcal{O}(E \log V)$. However, neither of these admits an efficient distributed version since they would require excessive communication: Kruskal's algorithm requires global knowledge of the graph to find the edge, while Prim's algorithm requires

communication within a fragment of size $\mathcal{O}(V)$ after each expansion.

To avoid these problems we need to pick the edges so that they can be found locally but only a few searches are performed in large fragments. To achieve this, we assign to each fragment a property called *level*, an integer ranging from 0 to an upper bound. We define a fragment of one node to be at level zero and allow a larger fragment to be formed in two ways:

1. If two fragments have the same level L and the same minimum-weight outgoing edge, they *combine* to produce a new fragment of level $L + 1$.

2. If a fragment is at level L and the fragment at the other of its minimum-weight outgoing edge has level $L' > L$, then the lower level fragment *joins* the higher level fragment, which maintains level L'.

If a fragment cannot apply either of the two rules, then it must simply wait until the level at the other end of its minimum-weight outgoing edge increases substantially. Note that from the perspective of the generic scheme, there is no difference between fragments combining or joining.

In section 4 we will show that at least one fragment can always apply either of the rules and thus they cannot lead to a deadlock. As we will show formally in section 5, the rules guarantee that each combination at least doubles the fragment size, which intuitively means that there cannot be too many large fragments. From this fact we derive the claimed upper bounds for the time and message complexity.

The overall algorithm now works as follows: First a fragment is formed for each individual node. Every time a new fragment is formed, it identifies its minimum-weight outgoing edge – independently of other fragments – and then informs the fragment at the other end of the edge. At some point after this the two fragments form a larger fragment by applying one of the two rules. The cycle continues until there is only one fragment. By the earlier description it should be clear that this procedure works. It remains to show how to implement it with local algorithms at each node. Specifically, we have to elaborate how the nodes within a fragment communicate to find the minimum-weight outgoing edges and handle fragment combinations.

3.2 Maintaining fragments on node level

To maintain fragments on node level, each node keeps track of which of its incident edges are part of the MST. Initially, each edge is unlabeled. When an edge is found to be part of the MST, it's labeled `Branch` and the node will attempt to connect with the fragment at the other end. If an edge is found *not* to be part of the MST, it's labeled `Rejected` instead and the node will never consider the edge again. Note that both endpoints of an edge maintain their own state of the edge and these two states can be temporarily inconsistent.

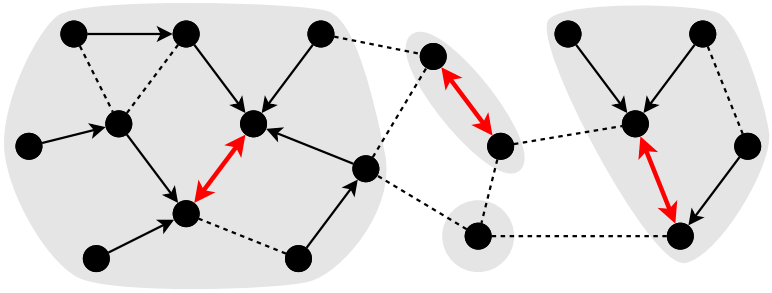

Figure 3: A possible state of the algorithm with four fragments. Dashed lines represent unlabeled edges. Other edges are labeled `Branch` and rejected edges are not depicted. Core edges are shown in bold red. An arrow from a node represents the inbound edge of the node. The nodes incident to a core have the core as their inbound edge, hence the double arrow.

In addition to the edge labels, each node contains a variable indicating the level of its fragment. Further, we require that each fragment with more than one node contain a special edge called the *core* of the fragment. When two fragments combine into a larger fragment, the edge joining the two original fragments always becomes the core of the new fragment.

The core has two purposes: Firstly, when nodes are searching for the minimum-weight outgoing edge of their fragment, they do not necessarily know which edges are outgoing: basically, they will need to query their neighbors as to whether they belong to the same or a different fragment. To answer such queries, each fragment must have a unique identifier known to all of its nodes. Since each fragment has a unique core edge, and since we assumed edge weights to be distinct, we can use the weight of the core edge as the unique identifier of its fragment. We will refer to the weight of the core edge simply as the *identity* of the fragment. Note that fragments at level zero do not need identities, since any query they receive necessarily comes from a different fragment.

Secondly, the core serves as a "hub" of computation within a fragment. When a new fragment is formed, the nodes incident to the core initiate the search for the minimum-weight outgoing edge by broadcasting the new level and fragment identity to all other nodes. The nodes will report their findings back to the core nodes, which then determine whether to connect with another fragment or if the algorithm has finished. To this end, all nodes must keep track of which one of their incident edges lead towards to core (figure 3). We will refer to this edge as the *inbound* edge of a node. All other edges within the fragment are called *outbound* edges (not to be confused with *outgoing* edges, which span two *different* fragments).

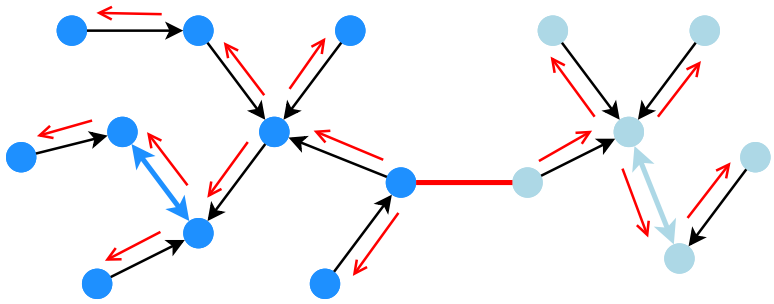

Figure 4: Two fragments combining by adding the red edge. The red arrows represent `Initiate` messages carrying the new identity to all nodes.

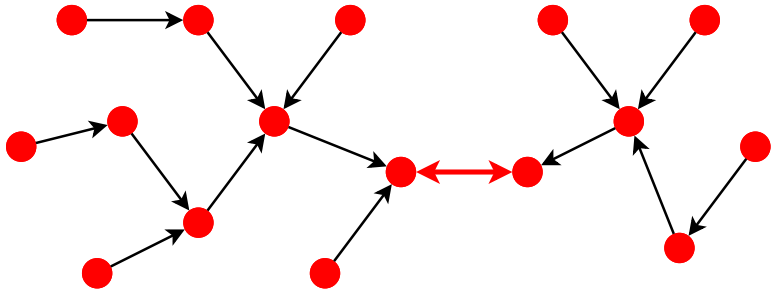

Figure 5: The combination of figure 4 completed. All nodes have updated their levels, fragment identities and inbound edges.

3.3 Finding the minimum-weight outgoing edge

We will now describe in detail how the fragments find their minimum-weight outgoing edges. First consider the simple case of a level zero fragment containing only one node. Trivially, the minimum-weight outgoing edge of such a fragment is just the minimum-weight incident edge of the node. With a simple local computation, the node identifies this edge, labels it as `Branch` and then informs the node at the other end by sending a message called `Connect` over the edge. This message carries the level of the fragment (zero) as a single argument. Once done, the node will wait for a response.

Next consider the case when two fragments at level L combine into a fragment of level $L + 1$ and the edge joining them becomes the core of the new fragment. Initially only the two nodes incident to the core are aware of the combination: the rest of the nodes in the two halves still remember the levels and identities of the two original fragments. The nodes incident to the core inform the other nodes by broadcasting a message called `Initiate` over their outbound edges (figure 4). The message carries the identity and the level of the new fragment as its arguments. Upon receiving the message, each node updates its own fragment identity and level according to the message. Each node also updates the inbound edge by marking it as the edge over

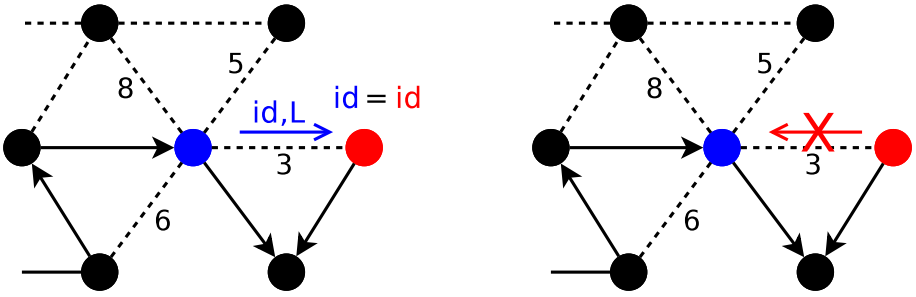

Figure 6: The blue node sends a `Test` message over its minimum-weight unlabeled edge (left). The red node receives the blue node's fragment identity, which matches to its own, so it responds with a `Reject` message (right).

which the `Initiate` message was received. Having updated its own state, each node then broadcasts the same message on its own outbound edges. In this fashion the message is ultimately propagated to all nodes of the new fragment (figure 5).

In addition to updating fragment information, the `Initiate` message initiates the search for the minimum-weight outgoing edge of the new fragment. This edge could be the minimum-weight outgoing edge of any node in the fragment. Therefore, having updated its values and propagated the `Initiate` message, each node proceeds to do a search in its local neighborhood.

All unlabeled incident edges of a node are possible candidates for the minimum-weight outgoing edge. However, the node cannot know beforehand if these edges are outgoing: the nodes at the other end could already belong to the same fragment. The solution is to send queries to these nodes to determine their fragments. The node picks the unlabeled edge of minimum weight and sends a message called `Test` over the edge. The message carries two arguments, the identity and the level of the node's own fragment. The node will then wait for a response from the other end.

A node receiving the `Test` message first compares its own fragment identity to the one in the message. If the identities are equal, then the two nodes must be in the same fragment, since two different fragments cannot have the same identity. Then, the receiving node labels the edge `Rejected` and sends a `Reject` message as a response (figure 6), to indicate that the other node should also reject the edge. As a small optimization, if the receiving node itself has sent a `Test` message to the other node, then it is not necessary to send the `Reject` message since receiving the `Test` message will already make the other node reject the edge.

On the contrary, if the identities are different, it is *not* obvious that the nodes are in different fragments. This is because, when two fragments combine, it takes some time until the `Initiate` message carrying the new identity reaches all nodes in the fragment. Then it is possible that two nodes

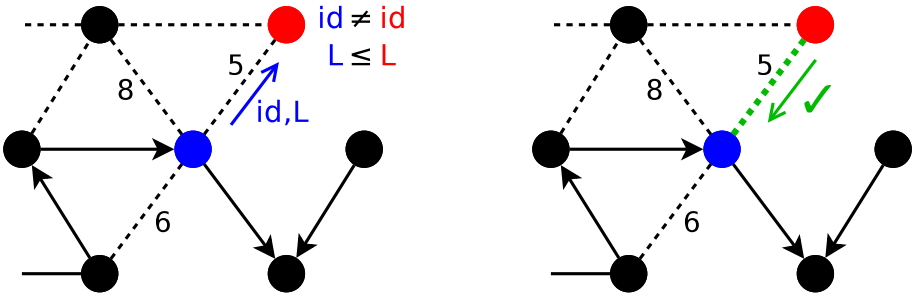

Figure 7: Having forgotten the rejected edge, the blue node sends a new `Test` message (left). This time identities don't match and the red node has a greater or equal level. Therefore, an `Accept` message is sent and the blue node marks the edge as its minimum-weight outgoing edge (right).

are in the same fragment but one of the nodes receiving a `Test` message from the other hasn't yet received the new identity. Note that in such a case the receiving node necessarily has a lower level than the other node (since levels increase only when identities are updated). Therefore the situation is resolved by comparing the fragment levels instead. If the receiving node has a level higher than or equal to that of the sending node, then it must be in a different fragment and an `Accept` message is sent as a response (figure 7). Otherwise the question cannot be decided yet and the receiving node will reappend the `Test` message to its message queue, to be processed once the node's level has increased enough.

Consider again the node that sent the `Test` message. If it receives a `Reject` message as a response (or a `Test` message with the same level), it labels the queried edge `Rejected` and proceeds to the next unlabeled edge of lowest weight, sending a new `Test` message. This procedure continues until an `Accept` message is received over one of the edges, which is then marked as the minimum-weight outgoing edge, or until the node no longer has any unlabeled edges to query, which means it has no outgoing edges.

Once each node has identified its minimum-weight outgoing edge (or lack thereof), the nodes must gather this information to the core so that the minimum-weight outgoing edge of the entire fragment can be determined. At first each leaf node, i.e., a node with no outbound edges, sends a message called `Report` over its inbound edge. The message carries a single argument, the weight of the minimum-weight outgoing edge, or a special value `None` if the node has no outgoing edges. Each non-leaf node waits until it has identified its own minimum-weight outgoing edge (if any) and received reports over all of its outbound edges. The node then determines the minimum weight w among the reported edges and its own minimum-weight outgoing edge and marks the edge leading towards it, i.e., either its own outgoing edge or one of the outbound edges. It then sends a `Report` message over its

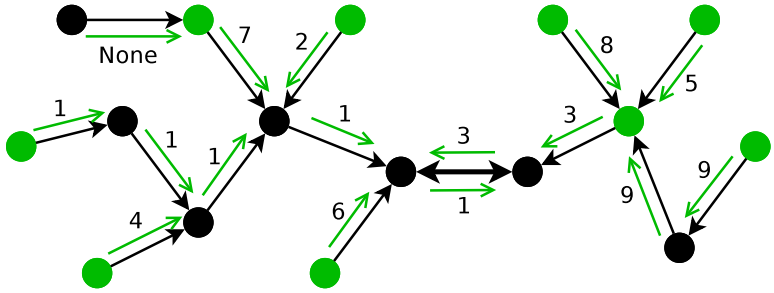

Figure 8: Nodes sending `Report` messages over their inbound edges. Each message carries the minimum weight of an outgoing edge in the corresponding subtree. Nodes shown in green send the weight of their own outgoing edge.

inbound edge carrying the weight w or `None` if neither the node nor any of the outbound neighbors had outgoing edges (figure 8).

In this way the nodes propagate the reports towards the core, marking the path leading towards the minimum-weight outgoing edge. Finally, the reports reach the nodes incident to the core and they both learn the minimum weight on their own side of the fragment. The core nodes then exchange `Report` messages, allowing both of them to determine the minimum weight of the entire fragment. Once determined, it only remains to inform the node incident to the minimum-weight outgoing edge.

To this end, the core node on the minimum-weight side sends a message called `Change-Core` over the edge marked as leading towards the minimum-weight outgoing edge. Any node receiving the message passes it on along the marked path until the node incident to the minimum-weight outgoing edge receives it (figure 9). This node then labels the edge as `Branch` and sends a `Connect` message over it, carrying the level of the fragment as an argument. If the core nodes both report `None` to each other, then the algorithm terminates.

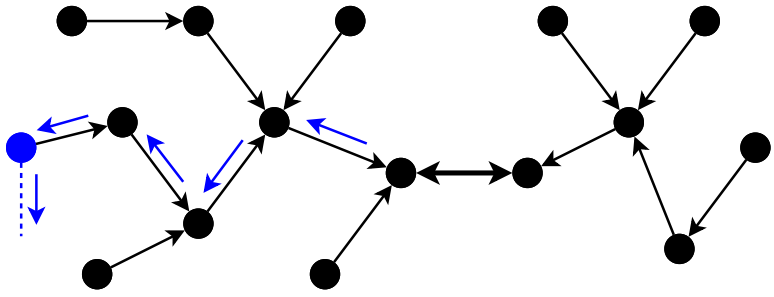

Figure 9: Nodes propagating the `Change-Core` message to the node incident to the minimum-weight outgoing edge (blue), which then sends a `Connect` message over the edge.

3.4 Handling connect messages

We have now described how each fragment identifies its minimum-weight outgoing edge and finally sends a `Connect` message on this edge. To complete the cycle, we will now describe how the `Connect` messages are handled to initiate a new search.

First consider the case when two nodes with the same fragment level and minimum-weight outgoing edge send `Connect` messages to each other. As per the rules described in section 3.1, their fragments will combine. Upon sending and receiving the messages, the two nodes initiate the fragment combination by incrementing their levels and marking the core as their inbound edge. From this point on everything proceeds as described in the section 3.3, starting with the two nodes sending an `Initiate` message over their outbound edges.

Next consider the case when a node receives a `Connect` message from a node with a lower level. In this case the lower level fragment will join the higher level fragment. To this end, the receiving node immediately sends an `Initiate` message as a response, carrying its level and identity. The other node will then propagate the message to all other nodes in the lower level fragment. Each node receiving the message updates its fragment information and (with one exception) starts searching for the minimum-weight outgoing edge, in the same manner as when fragments combine.

The exception is that if the node receiving the `Connect` message has already sent its `Report` message to its current core, it will respond with a variant of the `Initiate` message instead. The only difference is that a node receiving such a message will *not* start searching for its minimum-weight outgoing edge, nor will the nodes to which the message is propagated. This is because the `Connect` message was sent over the minimum-weight outgoing edge of the lower level fragment. Since the receiving node reported without considering that edge, the reported edge must have a lower weight than any outgoing edge of the lower level fragment. Therefore it is not necessary for the nodes in the lower level fragment to participate in the search.

In all other cases a node receiving the `Connect` message will simply reappend it to the end of the queue to be processed as soon as either of the two cases holds.

4 Correctness

We will now establish the correctness of the algorithm, i.e., that it always halts and really finds the minimum spanning tree. By the description in section 3.1, it should be clear that generic scheme is correct as long as either of the two rules is always applicable. Further, from section 3.3 it should be evident that if an edge is marked as `Branch` and a `Connect` message is sent over it, then it is the minimum-weight outgoing edge of the fragment.

It remains to show that deadlocks do not exist, i.e., each fragment with outgoing edges does eventually send a `Connect` over *some* edge and each such message leads to fragments combining or joining.

First note that `Initiate`, `Report` and `Change-Core` messages are always propagated without delay (easily shown by induction). Likewise, receiving a `Reject` or an `Accept` message will always trigger another `Test` or `Report` message. The only case where a potential deadlock could arise is the handling of `Test` and `Connect` messages: neither will be answered while the receiving node has a higher level than the sending node.

To show that there are no deadlocks, assume some state of the algorithm with an arbitrary collection of fragments. If there is only one fragment then all `Test` messages are rejected without delay and no `Connect` messages are sent. Therefore the algorithm will terminate. Assume now that there are at least two fragments. We show that the collection of fragments will change. Among the fragments with the lowest level, consider a fragment with a minimum-weight outgoing edge of lowest weight. Since the fragment has the lowest level, any `Test` message from it will be answered without delay. Therefore the fragment will proceed to send a `Connect` message. If the fragment at the other end has a higher level, the lower level fragment will join it. Otherwise the fragment at the other end has the same level and same minimum-weight outgoing edge (since it is the minimum among fragments of the lowest level). Therefore the two fragments will combine. Either way, the collection of fragments will change. Therefore there is no deadlock.

5 Analysis

In this section, we write V and E short for $|V|$ and $|E|$. We show that any run of the algorithm requires at most $5V \log_2 V + 2E$ messages to be sent between nodes. Further, assuming that local computation is instant and the transmission of each message requires one unit of time, we show that at most $5V \log_2 V$ time units are required.

First observe by induction that a fragment of level L has at least 2^L nodes. A zero level fragment by definition has one node. Assuming the claim holds for fragments of level L, when two such fragments combine, the resulting fragment of level $L + 1$ has at least $2^L + 2^L = 2^{L+1}$ nodes. Since a fragment may only grow when other fragments join it, the claim will continue to hold. Consequently, (since $2^L \leq V$) the highest level is at most $\log_2 V$.

5.1 Message complexity

Since the total number of messages sent equals the total number of messages received, we can count each message type from either perspective: we use the sender's point of view for `Connect`, `Test`, `Reject`, `Report` and `Change-Core` and the receiver's for `Initiate` and `Accept`.

First, all edges not part of the MST are rejected at some point. The rejection of an edge requires two messages, either one `Test` message followed by a `Reject` message or two independent `Test` messages. Hence, $2E$ is an upper bound for the number of `Reject` or `Test` messages sent over rejected edges. This does not cover *successful* `Test` messages, i.e., those sent over accepted edges.

At level zero a node may send at most one `Connect` message and receive at most one `Initiate` message. At the last level each node may send at most one `Report` message. Together these add at most $3V$ messages. Excluding the zeroth and the last, each node can go through at most $\log_2 V - 1$ levels. At each such level a node may send at most one successful `Test` message and receive at most one `Accept` message as a response. It may send at most one `Report` message and either a `Change-Root` or a `Connect` message. Finally, it may receive at most one `Initiate` message. These add at most 5 messages per node and level, a total of $5V(\log_2 V - 1)$ messages.

Summing all these up, we get at most $2E + 3V + 5V(\log_2 V - 1) < 2E + 5V \log_2 V$ messages. Also note that the most complex messages contain only a few bits to indicate the type of the message, one fragment level using at most $\lceil \log_2 V \rceil$ bits and one edge weight of some constant size.

5.2 Time complexity

First observe that if all fragments are at level L and send their `Connect` messages, it takes at most V time units until all nodes advance to level $L+1$: Two nodes sending a `Connect` message to each other will immediately exchange `Initiate` messages and after that at least one node receives an `Initiate` message by propagation during each time unit.

We show by induction that for each $L \leq \log_2 V$, once $5VL - 3V$ time units have passed since the start of the algorithm, each node is at level L or higher. By the previous observation all nodes are at level 1 or higher by V units so the claim holds for $L = 1$.

Consider now $L > 1$ and assume that $5VL - 3V$ time units have passed. It suffices to show that any nodes still at level L will be at level $L + 1$ by $5V$ additional units (since by then $5V(L + 1) - 3V$ units will have passed). By now all nodes at level L will have started sending their `Test` messages. By the assumption, all nodes are at level L or higher and therefore the `Test` messages are answered without delay. Each node can send at most V `Test` messages so it takes at most $2V$ units until all of them are answered (one unit for `Test` and another for the response). After this it takes at most V messages until the `Report` messages are propagated and at most V more until `Change-Core` is propagated. Summing these up, after $2V+V+V = 4V$ units all fragments at level L will have sent a `Connect` message. By our first observation, it takes at most V time units after this until the nodes at level L advance to level $L+1$. Therefore all nodes at level L will reach level $L+1$

after at most $4V + V = 5V$ units, which we wanted to show.

As a result, all nodes will reach the last level by at most $5V \log_2 V - 3V$ time units. At the last level all `Test` messages are answered in at most $2V$ units and `Report` messages propagated in at most V units. Therefore, the final number of time units required is no greater than $5V \log_2 V - 3V + 3V = 5V \log_2 V$.

6 Conclusion

We have presented a distributed algorithm for finding the minimum spanning tree in an asynchronous model of computation. Starting out with only the knowledge of their own incident edges, all nodes learn which of these edges are part of the MST. The algorithm works by maintaining known fragments of the MST and combining them into larger fragments by adding minimum-weight edges between two fragments iteratively. All fragments find these edges locally and to some extent in parallel. By adding the edges in a suitable order, the algorithm uses $\mathcal{O}(E + V \log V)$ messages and $\mathcal{O}(V \log V)$ time. The restriction of requiring distinct edge weights can be broken by probabilistic means or unique node identifiers.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3. edition, 2009.

[2] 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*, 5(1):66–77, 1983.

[3] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proceedings of the American Mathematical Society*, volume 7, pages 48–50, 1956.

[4] Robert C. Prim. Shortest connection networks and some generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.