Chapter 24

All-to-All Communication

In the previous chapters, we have mostly considered communication on a particular graph $G = (V, E)$, where any two nodes u and v can only communicate directly if $\{u, v\} \in E$. This is however not always the best way to model a network. In the Internet, for example, every machine (node) is able to "directly" communicate with every other machine via a series of routers. If every node in a network can communicate directly with all other nodes, many problems can be solved easily. For example, assume we have n servers, each hosting an arbitrary number of (numeric) elements. If all servers are interested in obtaining the maximum of all elements, all servers can simultaneously, i.e., in one communication round, send their local maximum element to all other servers. Once these maxima are received, each server knows the global maximum.

Note that we can again use graph theory to model this *all-to-all* communication scenario: The communication graph is simply the complete graph $\mathcal{K}_n := (V, \binom{V}{2})$. If each node can send its entire local state in a single message, then all problems could be solved in 1 communication round in this model! Since allowing unbounded messages is not realistic in most practical scenarios, we restrict the message size: Assuming that all node identifiers and all other variables in the system (such as the numeric elements in the example above) can be described using $\mathcal{O}(\log n)$ bits, each node can only send a message of size $\mathcal{O}(\log n)$ bits to all other nodes (messages to different neighbors can be different). In other words, only a constant number of identifiers (and elements) can be packed into a single message. Thus, in this model, the limiting factor is the amount of information that can be transmitted in a fixed amount of time. This is fundamentally different from the model we studied before where nodes are restricted to local information about the network graph.

In this chapter, we study one particular problem in this model, the computation of a minimum spanning tree (MST), i.e., we will again look at the construction of a basic network structure. Let us first review the definition of a minimum spanning tree from Chapter 2. We assume that each edge e is assigned a weight ω_e.

Definition 24.1 (MST)**.** *Given a weighted graph $G = (V, E, \omega)$. The MST of G is a spanning tree T minimizing $\omega(T)$, where $\omega(H) = \sum_{e \in H} \omega_e$ for any subgraph $H \subseteq G$.*

Remarks:

- Since we have a complete communication graph, the graph has $\binom{n}{2}$ edges in the beginning.
- As in Chapter 2, we assume that no two edges of the graph have the same weight. Recall that this assumption ensures that the MST is unique. Recall also that this simplification is not essential as one can always break ties by using the IDs of adjacent vertices.

For simplicity, we assume that we have a synchronous model (as we are only interested in the time complexity, our algorithm can be made asynchronous using synchronizer α at no additional cost (cf. Chapter 10). As usual, in every round, every node can send a (potentially different) message to each of its neighbors. In particular, note that the message delay is 1 for every edge e independent of the weight ω_e. As mentioned before, every message can contain a constant number of node IDs and edge weights (and $\mathcal{O}(\log n)$ additional bits).

Remarks:

- Note that for graphs of arbitrary diameter D, if there are no bounds on the number of messages sent, on the message size, and on the amount of local computations, there is a straightforward generic algorithm to compute an MST in time D: In every round, every node sends its complete state to all its neighbors. After D rounds, every node knows the whole graph and can compute any graph structure locally without any further communication.
- In general, the diameter D is also an obvious lower bound for the time needed to compute an MST. In a weighted ring, e.g., it takes time D to find the heaviest edge. In fact, on the ring, time D is required to compute any spanning tree.

In this chapter, we are not concerned with lower bounds, we want to give an algorithm that computes the MST as quickly as possible instead! We again use the following lemma that is proven in Chapter 2.

Lemma 24.2. *For a given graph G let T be an MST, and let $T' \subseteq T$ be a subgraph (also known as a fragment) of the MST. Edge $e = (u, v)$ is an outgoing edge of T' if $u \in T'$ and $v \notin T'$ (or vice versa). Let the minimum weight outgoing edge of the fragment T' be the so-called blue edge $b(T')$. Then $T' \cup b(T') \subseteq T$.*

Lemma 24.2 leads to a straightforward distributed MST algorithm. We start with an empty graph, i.e., every node is a fragment of the MST. The algorithm consists of phases. In every phase, we add the blue edge $b(T')$ of every existing fragment T' to the MST. Algorithm 24.3 shows how the described simple MST construction can be carried out in a network of diameter 1.

Theorem 24.4. *On a complete graph, Algorithm 24.3 computes an MST in time $\mathcal{O}(\log n)$.*

Proof. The algorithm is correct because of Lemma 24.2. Every node only needs to send a single message to all its neighbors in every phase (line 4). All other computations can be done locally without sending other messages. In particular,

Algorithm 24.3 Simple MST Construction (at node v)

```
1: // all nodes always know all current MST edges and thus all MST fragments
2: while $v$ has neighbor $u$ in different fragment do
3:     find lowest-weight edge $e$ between $v$ and a node $u$ in a different fragment
4:     send $e$ to all nodes
5:     determine blue edges of all fragments
6:     add blue edges of all fragments to MST, update fragments
7: end while
```

the blue edge of a given fragment is the lightest edge sent by any node of that fragment. Because every node always knows the current MST (and all current fragments), lines 5 and 6 can be performed locally.

In every phase, every fragment connects to at least one other fragment. The minimum fragment size therefore at least doubles in every phase. Thus, the number of phases is at most $\log_2 n$. □

Remarks:

- Algorithm 24.3 does essentially the same thing as the GHS algorithm (Algorithm 2.18) discussed in Chapter 2. Because we now have a complete graph and thus every node can communicate with every other node, things get simpler (and also much faster).
- Algorithm 24.3 does not make use of the fact that a node can send different messages to different nodes. Making use of this possibility will allow us to significantly reduce the running time of the algorithm.

Our goal is now to improve Algorithm 24.3. We assume that every node has a unique identifier. By sending its own identifier to all other nodes, every node knows the identifiers of all other nodes after one round. Let $\ell(F)$ be the node with the smallest identifier in fragment F. We call $\ell(F)$ the leader of fragment F. In order to improve the running time of Algorithm 24.3, we need to be able to connect every fragment to more than one other fragment in a single phase. Algorithm 24.5 shows how the nodes can learn about the $k = |F|$ lightest outgoing edges of each fragment F (in constant time!).

Given this set E' of edges, each node can locally decide which edges can safely be added to the constructed tree by calling the subroutine AddEdges (Algorithm 24.6). Note that the set of received edges E' in line 14 is the same for all nodes. Since all nodes know all current fragments, all nodes add the same set of edges!

Algorithm 24.6 uses the lightest outgoing edge that connects two fragments (to a larger super-fragment) as long as it is safe to add this edge, i.e., as long as it is clear that this edge is a blue edge. A (super-)fragment that has outgoing edges in E' that are surely blue edges is called *safe*. As we will see, a super-fragment $\mathcal{F}$ is safe if all the original fragments that make up $\mathcal{F}$ are still incident to at least one edge in E' that has not yet been considered. In order to determine whether all lightest outgoing edges in E' that are incident to a certain fragment F have been processed, a counter $c(F)$ is maintained (see line 2). If an edge incident to two (distinct) fragments F_i and F_j is processed, both $c(F_i)$ and $c(F_j)$ are decremented by 1 (see Line 8).

Algorithm 24.5 Fast MST construction (at node v)

```
1: // all nodes always know all current MST edges and thus all MST fragments
2: repeat
3:     $F :=$ fragment of $v$;
4:     $\forall F' \neq F$, compute min-weight edge $e_{F'}$ connecting $v$ to $F'$
5:     $\forall F' \neq F$, send $e_{F'}$ to $\ell(F')$
6:     if $v = \ell(F)$ then
7:         $\forall F' \neq F$, determine min-weight edge $e_{F,F'}$ between $F$ and $F'$
8:         $k := |F|$
9:         $E(F) := k$ lightest edges among $e_{F,F'}$ for $F' \neq F$
10:        send send each edge in $E(F)$ to a different node in $F$
                       // for simplicity assume that v also sends an edge to itself
11:    end if
12:    send edge received from $\ell(F)$ to all nodes
13:    // the following operations are performed locally by each node
14:    $E' :=$ edges received by other nodes
15:    AddEdges($E'$)
16: until all nodes are in the same fragment
```

An edge connecting two distinct super-fragments $\mathcal{F}'$ and $\mathcal{F}''$ is added if at least one of the two super-fragments is safe. In this case, the two super-fragments are merged into one (new) super-fragment. The new super-fragment is safe if and only if both original super-fragements are safe and the processed edge e is not the last edge in E' incident to any of the two fragments F_i and F_j that are incident to e, i.e., both counters $c(F_i)$ and $c(F_j)$ are still positive (see line 12).

The considered edge e may not be added for one of two reasons. It is possible that both $\mathcal{F}'$ and $\mathcal{F}''$ are not safe. Since a super-fragment cannot become safe again, nothing has to be done in this case. The second reason is that $\mathcal{F}' = \mathcal{F}''$. In this case, this single fragment may become unsafe if e reduced either $c(F_i)$ or $c(F_j)$ to zero (see line 18).

Lemma 24.7. *The algorithm only adds MST edges.*

Proof. We have to prove that at the time we add an edge e in line 9 of Algorithm 24.6, e is the blue edge of some (super-)fragment. By definition, e is the lightest edge that has not been considered and that connects two distinct super-fragments $\mathcal{F}'$ and $\mathcal{F}''$. Since e is added, we know that either $safe(\mathcal{F}')$ or $safe(\mathcal{F}'')$ is true. Without loss of generality, assume that $\mathcal{F}'$ is safe. According to the definition of *safe*, this means that from each fragment F in the super-fragment $\mathcal{F}'$ we know at least the lightest outgoing edge, which implies that we also know the lightest outgoing edge, i.e., the blue edge, of $\mathcal{F}'$. Since e is the lightest edge that connects *any* two super-fragments, it must hold that e is exactly the blue edge of $\mathcal{F}'$. Thus, whenever an edge is added, it is an MST edge. □

Theorem 24.8. *Algorithm 24.5 computes an MST in time $\mathcal{O}(\log \log n)$.*

Proof. Let β_k denote the size of the smallest fragment after phase k of Algorithm 24.5. We first show that every fragment merges with at least β_k other fragments in each phase. Since the size of each fragment after phase k is at least

Algorithm 24.6 AddEdges(E'): Given the set of edges E', determine which edges are added to the MST

```
Let $F_1, \ldots, F_r$ be the initial fragments
$\forall F_i \in \{F_1, \ldots, F_r\}, c(F_i) := \#$ incident edges in $E'$
Let $\mathcal{F}_1 := F_1, \ldots, \mathcal{F}_r := F_r$ be the initial super-fragments
$\forall \mathcal{F}_i \in \{\mathcal{F}_1, \ldots, \mathcal{F}_r\}, safe(\mathcal{F}_i) := true$
while $E' \neq \emptyset$ do
  $e :=$ lightest edge in $E'$ between the original fragments $F_i$ and $F_j$
  $E' := E' \setminus \{e\}$
  $c(F_i) := c(F_i) - 1$, $c(F_j) := c(F_j) - 1$
  if $e$ connects super-fragments $\mathcal{F}' \neq \mathcal{F}''$ and $(safe(\mathcal{F}')$ or $safe(\mathcal{F}''))$ then
    add $e$ to MST
    merge $\mathcal{F}'$ and $\mathcal{F}''$ into one super-fragment $\mathcal{F}_{new}$
    if $safe(\mathcal{F}')$ and $safe(\mathcal{F}'')$ and $c(F_i) > 0$ and $c(F_j) > 0$ then
      $safe(\mathcal{F}_{new}) := true$
    else
      $safe(\mathcal{F}_{new}) := false$
    end if
  else if $\mathcal{F}' = \mathcal{F}''$ and $(c(F_i) = 0$ or $c(F_j) = 0)$ then
    $safe(\mathcal{F}') := false$
  end if
end while
```

β_k by definition, we get that the size of each fragment after phase $k+1$ is at least $\beta_k(\beta_k+1)$. Assume that a fragment F, consisting of at least β_k nodes, does not merge with β_k other fragments in phase $k+1$ for any $k \geq 0$. Note that F cannot be safe because being safe implies that there is at least one edge in E' that has not been considered yet and that is the blue edge of F. Hence, the phase cannot be completed in this case. On the other hand, if F is not safe, then at least one of its sub-fragments has used up all its β_k edges to other fragments. However, such an edge is either used to merge two fragments or it must have been dropped because the two fragments already belong to the same fragment because another edge connected them (in the same phase). In either case, we get that any fragment, and in particular F, must merge with at least β_k other fragments.

Given that the minimum fragment size grows (quickly) in each phase and that only edges belonging to the MST are added according to Lemma 24.7, we conclude that the algorithm correctly computes the MST. The fact that

$$\beta_{k+1} \geq \beta_k(\beta_k+1)$$

implies that $\beta_k \geq 2^{2^{k-1}}$ for any $k \geq 1$. Therefore after $1+\log_2\log_2 n$ phases, the minimum fragment size is n and thus all nodes are in the same fragment. □

Chapter Notes

There is a considerable amount of work on distributed MST construction. Table 24.9 lists the most important results for various network diameters D. In the above text we focus only on $D = 1$.

Upper Bounds

Graph Class	Time Complexity	Authors
General Graphs	$\mathcal{O}(D + \sqrt{n} \cdot \log^* n)$	Kutten, Peleg [KP95]
Diameter 2	$\mathcal{O}(\log n)$	Lotker, Patt-Shamir, Peleg [LPSP06]
Diameter 1	$\mathcal{O}(\log \log n)$	Lotker, Patt-Shamir, Pavlov, Peleg [LPPSP03]

Lower Bounds

Graph Class	Time Complexity	Authors
Diameter $\Omega(\log n)$	$\Omega(D + \sqrt{n}/\log n)$	Das Sarma, Holzer, Kor, Korman, Nanongkai, Pandurangan, Peleg, Wattenhofer [SHK+12]
Diameter 4	$\Omega\left((n/\log n)^{1/3}\right)$	Das Sarma, Holzer, Kor, Korman, Nanongkai, Pandurangan, Peleg, Wattenhofer [SHK+12]
Diameter 3	$\Omega\left((n/\log n)^{1/4}\right)$	Das Sarma, Holzer, Kor, Korman, Nanongkai, Pandurangan, Peleg, Wattenhofer [SHK+12]

Table 24.9: Time complexity of distributed MST construction

We want to remark that the above lower bounds remain true for randomized algorithms. We can even not hope for a better randomized approximation algorithm for the MST as long as the approximation factor is bounded polynomially in n. On the other hand it is not known whether the $\mathcal{O}(\log \log n)$ time complexity of Algorithm 24.5 is optimal. In fact, no lower bounds are known for the MST construction on graphs of diameter 1 and 2. Algorithm 24.5 makes use of the fact that it is possible to send different messages to different nodes. If we assume that every node always has to send the same message to all other nodes, Algorithm 24.3 is the best that is known. Also for this simpler case, no lower bound is known.

Bibliography

[KP95] Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 238–251. ACM, 1995.

[LPPSP03] Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. Mst construction in o (log log n) communication rounds. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 94–100. ACM, 2003.

[LPSP06] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed mst for constant diameter graphs. *Distributed Computing*, 18(6):453–460, 2006.

[SHK$^+$12] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.