Chapter 13

Global Problems II

We saw in Chapter 2 that building a minimum spanning tree (MST) in a distributed system can be done using the GHS algorithm. The algorithm needs $\mathcal{O}(n \log n)$ rounds in the worst case, assuming a node can only send $\mathcal{O}(\log n)$ bits through one edge per round.

In this chapter we will improve the MST's runtime to $\tilde{\mathcal{O}}(D + \sqrt{n})$, which is nearly optimal.

13.1 The diameter D

The GHS algorithm takes quasi-linear time in the worst time. Let us try to find a better algorithm. First, we will get a lower bound on the round complexity for computing a MST. The network diameter D, which does not take into account the edge weights, plays a central role in distributed algorithms. Notably, for this problem, one could think that computing a MST is harder than computing a BFS tree, and therefore requires $\Omega(D)$ rounds. Let us prove this formally.

Theorem 13.1 (MST is a global problem)**.** *There exists an n-node network of diameter $D < n/2$ in which any distributed MST algorithm requires $\Omega(D)$ rounds.*

Input. *Initially, each node v knows the IDs of its neighbors and the weights of its incident edges.*

Output. *The algorithm must elect an MST $T \subseteq E$ so that, upon termination, every node v has learned exactly which of its incident edges belong to T.*

Proof. Consider the following graph. Let $G = C_{2D}$ be the cycle whose antipodal edges are

$$(a, b) \quad \text{and} \quad (x, y).$$

See Figure 13.2. Attach the remaining $n-2D$ nodes as leaves anywhere onto the cycle with arbitrary weights (this does not increase the diameter beyond $D < n/2$ or affect the cycle-MST). Define two weight-functions $w^{(1)}, w^{(2)} : E(G) \to \mathbb{R}$ by

$$w^{(i)}(a, b) = 2, \quad w^{(i)}(x, y) = 2i-1, \quad w^{(i)}(e) = 1 \quad (e \notin \{(a, b), (x, y)\}),\ i = 1, 2.$$

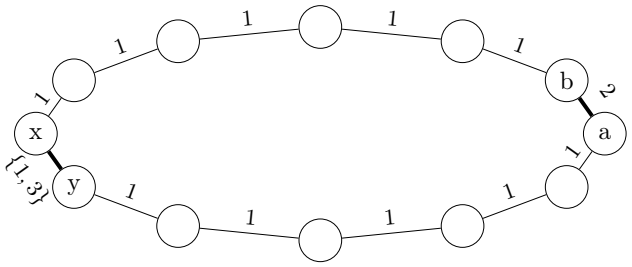

Figure 13.2: Construction of theorem 13.1.

In both cases the unique MST contains *all* edges of weight 1 and exactly one of $\{(a, b), (x, y)\}$:

- For $i = 1$: $w^{(1)}(x, y) = 1 < w^{(1)}(a, b) = 2$, so the MST contains (x, y) but not (a, b).
- For $i = 2$: $w^{(2)}(a, b) = 2 < w^{(2)}(x, y) = 3$, so the MST contains (a, b) but not (x, y).

Thus to decide at node b whether the edge (a, b) is in the MST, one must learn which case holds—i.e. whether $w(x, y) = 1$ or 3. That information is located at distance D from b, so any distributed MST algorithm requires $\Omega(D)$ rounds. □

13.2 Improving GHS when $D \ll n$

GHS reminder. The algorithm starts by having each node being its own fragment and then repeatedly merges fragments using their lightest outgoing edge, called the blue edge. Each phase finds a blue edge using flooding and echo, then merges fragments over it. This process repeats until only one fragment remains, which happens after at most $\mathcal{O}(\log n)$ rounds. The algorithm needs $\mathcal{O}(n \cdot \log n)$ time in the worst case.

Because of the $\Omega(D)$ lower bound, when the diameter of the network D is close to n, then we know that the GHS algorithm is nearly optimal. Indeed, ignoring logarithmic factors, GHS takes n rounds, which would be $\mathcal{O}(D)$ rounds and match the lower bound if $D = \Theta(n)$.

This leads to a key question in distributed computing for global problems: Can we design MST algorithms whose runtime is sublinear in n when the graph diameter $D \ll n$ is much smaller than the number of nodes?

The answer is indeed, yes. We now present an idea that leads to a $\mathcal{O}((D + \sqrt{n}) \log n)$ rounds algorithm. This runtime is optimal up to logarithmic factors.

The main idea. The main bottleneck in GHS was the flood/echo which can take $\mathcal{O}(n)$ rounds in worst case. This is not just theoretical—consider the case where some fragment forms a path of length, say, n/100n/100. The flood/echo within this fragment necessarily requires $\Omega(n)$ rounds while the diameter of the graph could be 22 if there was a node in another fragment connected to all the nodes along the path.

The main idea to work around this issue is to propagate fragment information through the BFS tree which has diameter $\mathcal{O}(D)$ and can be built at the

beginning in $\mathcal{O}(D)$ rounds. The BFS tree can easily be used replace flood/echo of a single fragment. Unfortunately, we cannot use this for all fragments since the BFS tree will become congested. Replacing flood/echo for a very large $x \gg D$ fragments through the BFS tree takes $\Theta(x)$ rounds in the best case scenario. However, only the largest n/x fragments have more than x nodes and the standard flood/echo on smaller fragments work well in $\mathcal{O}(x)$ rounds. We can strike a balance between x and n/x by choosing $x = \sqrt{n}$. This means we will do flood/echo for small fragments using the standard algorithm and for large (more than $\sqrt{n}$ nodes) using the BFS tree. The resulting MST algorithm will have a near-optimal $\mathcal{O}((D + \sqrt{n}) \log n)$ round worst-case runtime.

Algorithm outline. The algorithm we describe will be very similar to GHS, with $\mathcal{O}(\log n)$ phases of fragment merging during which we gradually grow a forest of fragments until we reach a spanning tree. However, we will be a little bit more careful during merging and finding blue edges to ensure the optimized runtime. We start with the trivial forest where each node forms its own fragment of the forest, that is, each node is one separate fragment in our partition of G.

In each phase, each fragment S_i will have a leader node $s_i \in S_i$, and moreover, the leader will know the size of its fragment. Each fragment S_i suggests a merge along the edge with exactly one endpoint in S_i that has the smallest weight among such edges. We call such an edge a *blue edge*. Recall from the second lecture (Lemma 2.17) that all such edges belong to MST. We soon explain how to compute these blue edges. Let us for now continue with the high-level explanation of how to use these edges to merge parts.

Let N be the current number of connected fragments of the forest. If $N = 1$, we are done already. Otherwise, each fragment suggests one merge edge. Each edge might be suggested by two of its endpoints, so we have at least $N/2$ suggested edges in total. We add these edges to the forest and thus, effectively, merge the connected fragments at their two endpoints. Hence, the number of fragments is reduced by at least half. After $\log n$ iterations, the number of connected fragments is down to 1, which means we have reached a spanning tree.

What remains is to explain how to find the merge edges, and how to perform the merges. We first discuss the process of computing minimum-weight edges of one phase in $\mathcal{O}(D + \sqrt{n})$ rounds. Then, we will discuss how to perform the merges in the same round complexity.

Computing Blue Edges in $\mathcal{O}(D + \sqrt{n})$ rounds

Our objective is to let each node know the the blue (minimum weight outgoing edge) of its fragment. More concretely, let each node v set $c(v)$ to be the minimum-weight outgoing edge among edges incident to v. The objective is that each node $v \in S_i$ learns the weight of the minimum-weight outgoing edge among edges all edges incident on fragment S_i. Notice that node v can easily find $c(v)$ by first receiving from all neighbors the fragment leader IDs of their fragments and then only considering the smallest of those edges having the other endpoint in a different fragment. We handle the fragments in two categories of

[0]This assumes that the edge weights are unique, which we can do without loss of generality by breaking ties depending on the identifier of each edge's endpoints.

small and large, depending on whether the fragment has at least $\sqrt{n}$ vertices or not.

Small fragments: Consider a single small fragment S_i, which means this fragment has no more than $\sqrt{n}$ vertices. Then, in this fragment, each node v starts with its own smallest weight-outgoing edge and its weight $c(v)$. Then, we perform a flood/echo (or simply minimum flooding) on the BFS tree of this fragment S_i. This flood/echo goes from the leaves to the root, maintaining the minimum value seen, and thus eventually delivering the minimum-weight outgoing edge to the fragment leader. The information about this edge can be delivered to all nodes of the fragment by a broadcast from the root to the leaves.

Large fragments: There are at most $n/\sqrt{n} = \sqrt{n}$ large parts, as each of them has at least $\sqrt{n}$ vertices. Since the number of large fragments is relatively small, we can handle all these fragments by performing their communications on the BFS of the whole graph G, simultaneously. Using standard pipelining techniques (see Lemma 13.3, proved in exercise 1 of the lecture), we can compute the minimum weight outgoing edges of all these $\sqrt{n}$ fragments in $\mathcal{O}(D + \sqrt{n})$ rounds.

Lemma 13.3 (Pipelining in a Tree)**.** *Let G be a network graph and T a spanning tree of depth D in G. Given k subsets of nodes (fragments) $S_1, S_2, \ldots, S_k$ (not necessarily connected or disjoint), one can perform a flood/echo operation within each fragment simultaneously in $\mathcal{O}(D + k)$ rounds.*

Concretely, each node x initially knows some unique $\mathcal{O}(\log n)$-bit fragment $ID(x)$ and some private $\mathcal{O}(\log n)$-bit integer x_v. Upon completion, each node v learns the min, max, and sum of the inputs in its fragment $\{x_w \mid ID(w) = ID(v)\}$.

Merging fragments

We need to slightly change how we merge the fragments in order to enable each fragment leader to know its component size. We want to design low-depth merges. Specifically, we restrict the merging fragments to be star shaped, using a simple random coin idea. Each fragment (leader) tosses a random coin and we then allow only merges centered on head-parts. Each head-part accepts incoming suggested merge-edges from tail-parts. The leader of this head-fragment becomes the leader of the merged new part. In exercise 2 of today's lecture, we will see that this probabilistic merging process is able to compute a MST within $\mathcal{O}(\log n)$ phases, with high probability.

What we need to compute for a merge: We need to make all nodes learn, besides the minimum-weight outgoing edge of their fragment, two extra things: (1) the coin tossed by their fragment leader, (2) the ID of their new fragment leader, (3) the size of the new fragment. We next explain how to perform each of these steps in $\mathcal{O}(D + \sqrt{n})$ rounds.

- Item (1)-which is to let each node know the coin toss of its fragment leader-is by a simple small change to the messages sent in computing the blue edge: now the message starting at the root also carries the random bit flipped by the leader.

- Item (2)-which is to let each node know its new fragment leader ID-is performed as follows: We define the fragment leader ID to be the leader

of the center fragment of the merge, who had a head coin. This ID is already delivered to the physical endpoint of the merge edge in the tail part.

Hence, within the tail fragment, all that we need to do is that one node knows the ID of the new leader and we want all nodes to have it. If the fragment was small, we can do it directly in $\mathcal{O}(\sqrt{n})$ rounds, inside the fragment. For large fragments, which there are only at most $\sqrt{n}$ of them, we can broadcast the their new ID leader, which is known to the physical endpoint of their merge edge, to all nodes of the graph in $\mathcal{O}(D + \sqrt{n})$ rounds.

- Item (3)-which is to let each node know the size of its fragment-can be performed similar to (2). First, in the center-of-merge head fragment, the physical endpoints of the merge can receive from their other endpoints the sizes of the merging tail fragments. Then, within this head fragment, we can compute the new fragment size by performing a simple converge-case, if the fragment is small, and by doing it through the global BFS tree, for large fragments. At the end, this information can be passed to all vertices of the new fragment, by first delivering it to all nodes of the head fragment, then passing it through the physical edges to the tail fragments, and then spreading it in the tail fragments.

13.3 A more general treatment: Shortcuts

The above division to small and large categories, and then the rules for where each of these should communicate, leads to an $\mathcal{O}(D + \sqrt{n})$-round algorithm. This is nearly-optimal in the worst case. However, in many graphs families of interest, this would be quite far from desirable bounds. In the following, and mostly as side remarks, we introduce a graph-theoretic notion of low-congestion shortcuts and briefly outline how this notion leads to more efficient algorithms, as well as a simple and clean unification of many methods.

Definition 13.4 (Low-Congestion Shortcuts)**.** *Consider a graph $G = (V, E)$ and a partition of V into disjoint subsets $S_1, \ldots, S_N \subset V$, each inducing a connected subgraph $G[S_i]$. We define an α-congestion shortcut with dilation β to be a set of subgraphs $H_1, \ldots, H_N \subset G$, one for each set S_i, such that:*
(1) For each i, the diameter of the subgraph $G[S_i] + H_i$ is at most β.
(2) For each $e \in E$, the number of subgraphs $G[S_i] + H_i$ containing e is at most α.

To better understand the definition, consider the following simple claim.

Lemma 13.5. *For any graph $G = (V, E)$ and any partition into V disjoint subsets $S_1, S_2, \ldots, S_k$, each inducting a connected subgraph, there always exists a $\tilde{O}(\sqrt{n})$-congestion shortcut with dilation $\tilde{O}(D + \sqrt{n})$.*

Proof. Let $n := |V|$. Consider a subset S_i. If $|S_i| \geq \sqrt{n}$ (large subsets), assign $H_i := E$ (the BFS tree works too). Otherwise (for small subsets), let H_i be the internal edges between S_i.

Note that the diameter of $G[S_i] + H_i$ for large subsets is D since we assign all edges to it, while the diameter of $G[S_i] + H_i$ for small subsets is $\sqrt{n}$ since the

diameter of a connected graph cannot be larger than the number of nodes in it. For congestion, node that each edge can be included in only one small subset (due to disjointness of the subsets) and every large subset. However, there are at most *sqrtn* large subsets (since there can be at most $n/\sqrt{n} \leq \sqrt{n}$ of them), hence the congestion is $\tilde{O}(\sqrt{n})$ □

Low-congestion shortcuts can be directly used to construct efficient distributed algorithms for the MST.

Theorem 13.6. *Suppose that the graph family $\mathcal{G}$ is such that for each graph $G \in \mathcal{G}$, and any partition of G into vertex-disjoint connected subsets $S_1, \ldots, S_N$, we can find an α-congestion β-dilation shortcut such that $\max\{\alpha, \beta\} \leq K$. Here, K can be a function of the family $\mathcal{G}$, and it can depend on n and D. Moreover, we assume such a low-congestion shortcut can be found in $\tilde{\mathcal{O}}(T)$ rounds.*

Then, there is a randomized distributed MST algorithm that computes an MST in $\tilde{\mathcal{O}}(T) + \mathcal{O}\left(\alpha \log n + \beta \log^2 n\right) = \tilde{\mathcal{O}}(T + K)$ rounds, with high probability, in any graph from the family $\mathcal{G}$.

Thus, one can reconstruct our $\tilde{O}(D + \sqrt{n})$ round MST algorithm for general graphs. There are a number of graph families where the question, and especially its graph-theoretic aspects, becomes much more interesting:

- For planar graphs and a few generalizations (e.g. bounded-genus graphs), it has been shown[MH16] that there always exists an α-congestion β-dilation shortcut such that $\max\{\alpha, \beta\} \leq \tilde{O}(D)$ and moreover, such a shortcut can be found in $\tilde{\mathcal{O}}(D)$ rounds. This leads to an $\tilde{\mathcal{O}}(D)$-round algorithm for MST in planar and near-planar graphs.
- In Erdős-Renyi random graph $G_{n,p}$, where each of the possible $\binom{n}{2}$ edges is included with probability $p \geq \Omega(\log n/n)$ (that is, above the connectivity threshold), it has been shown that there always exists a low-congestion shortcut with $\max\{\alpha, \beta\} \leq \mathit{poly} \log n \log n$, and such a shortcut can be found in $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$ rounds. That leads to an $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$ round algorithm for MST in Erdős-Renyi random graphs. See [GKS17] for this result and extensions to much broader graph families (those with small random walk mixing time).