15-850: Advanced Algorithms **CMU, Fall 2018**
Lecture #1: Deterministic MSTs August 27, 2018
Lecturer: Anupam Gupta Scribe: C.J. Argue and Xinyu Wu

1 Minimum Spanning Trees: History

The minimum spanning tree problem is classic: given a weighted undirected graph, $G = (V, E)$ with n nodes and m edges, where the edges have weights $w(e) \in \mathbb{R}$, find a spanning tree in the graph with the minimum total edge weight. As a classic (and important) problem, it's been tackled many times. Here's a brief, not-quite-comprehensive history of its optimization, all without making any assumptions on the edge weights other that they can be compared in constant time:

- Borůvka [Bor26] gave the first MST algorithm in 1926; it was rediscovered by Choquet, Sollis, and others. Jarník [Jar30] gave his algorithm in 1930, and it was rediscovered by Prim ('57) and Dijkstra ('59), among others. Kruskal [Kru56] gave his algorithm in '56. All these can be easily implemented in $O(m \lg n)$ time.
- Yao's algorithm [Yao75] in '75 achieved a runtime of $O(m \lg \lg n)$.
- In 1984, Fredman and Tarjan [FT87] gave an $O(m \lg^* n)$ time algorithm. This was soon improved by Gabow, Galil, Spencer, and Tarjan [GGST86] ('86) to get $O(m \lg \lg^* n)$.
- In 1995 Karger, Klein and Tarjan [KKT95] got the holy grail of $O(m)$ time! ...but it was a randomized algorithm, so the search for a deterministic linear-time algorithm continued.
- In 1997, Chazelle [Cha00] gave an $O(m\alpha(n))$ time deterministic algorithm. Here $\alpha(n)$ is the inverse Ackermann function [see appendix].
- In 1998, Pettie and Ramachandran [PR02] gave an optimal algorithm, though we don't know its runtime. (The way this works is this: if there exists an algorithm which uses $MST^*(m, n)$ comparisons on all graphs with m edges and n nodes, the Pettie-Ramachandran algorithm will run in time $O(MST^*(m, n))$.)

In this lecture, we'll go through the three classics (Prim's, Kruskal's, and Borůvka's), and then talk about Fredman and Tarjan's algorithm, and the Karger, Klein, and Tarjan randomized algorithm.

For the rest of this lecture, we will assume that the edge weights are *distinct*. This does not change things in any essential way, but it ensures that the MST is unique, and hence simplifies some of the statements. We also assume the graph is simple, and hence $m = O(n^2)$.

1.1 The Two Basic Rules

Most of these algorithms rely on two rules: the *cut rule* (known in Tarjan's notation as the blue rule) and the *cycle rule* (or the red rule).

Cut Rule. *The cut rule states that for any cut of the graph (a cut is a partition of the vertices into two sets), the minimum-weight edge that crosses the cut must be in the MST. This rule helps us determine what to add to our MST.*

Proof. Let $S \subsetneq V$ be any nonempty proper subset of vertices, let $e = uv$ be the minimum-weight edge that crosses the cut defined by $S, \bar{S}$ (WLOG $u \in S$, $v \notin S$), and let T be a spanning tree not containing e. Then $T \cup \{e\}$ contains a unique cycle C, and since C crosses the cut $[S, \bar{S}]$ once (namely at e), it must cross also at another edge e'. $w(e') > w(e)$, so $T' = (T - \{e'\}) \cup \{e\}$ is a lower-weight tree than T. In particular, T is not the MST, and since T was an arbitrary spanning tree not containing e, the MST must contain e. □

Cycle Rule. *The cycle rule states that if we have a cycle, the heaviest edge on that cycle cannot be in the MST. This helps us determine what we can remove in constructing the MST.*

Proof. Let C be any cycle, let e be the heaviest edge in C. For a contradiction, let T be an MST that contains e. Dropping e from T gives two components. Now there must be some edge e' in $C \setminus \{e\}$ that crosses between these two components, and hence $T' := (T - \{e'\}) \cup \{e\}$ is a spanning tree. (Make sure you see why.) By choice of e we have $w(e') < w(e)$, so T' is a lower-weight spanning tree than T, a contradiction. □

2 The Classical Algorithms

2.1 Kruskal's Algorithm

For Kruskal's Algorithm we first sort all the edges such that $w(e_1) < w(e_2) < \cdots < w(e_m)$. This takes $O(m \lg m) = O(m \lg n)$ time. We then iterate through the edges, adding an edge if and only if it connects two vertices which are not currently in the same component. Figure 1.1 gives an example of how we add the edges.

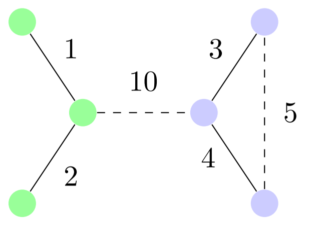

Figure 1.1: Dashed lines are not yet in the MST. Note that 5 will be analyzed next, but will not be added. 10 will be added. Colors designate connected components.

We can keep track of which component each vertex is in using a *disjoint set union-find* data structure. This has three operations:

- `makeset`$(elem)$, which takes an element $elem$ and creates a new singleton set for it,
- `find`$(elem)$, which finds the canonical representative for the set containing the element $elem$, and
- `union`$(elem_1, elem_2)$, which merges the two sets that $elem_1$ and $elem_2$ are in.

There is an implementation of this which allows us to do m operations in $O(m\alpha(m))$ *amortized* time, where $\alpha(\cdot)$ is an inverse of the Ackermann function, and hence a very slow-growing function. Since this $O(m\alpha(m))$ term is dominated by the $O(m \lg n)$ we get from sorting, so the overall runtime is $O(m \lg n)$.

2.2 Prim's Algorithm

For Prim's algorithm we first take an arbitrary root vertex r to start our MST T. At each iteration we take the cheapest edge connecting of our current tree T to some vertex not yet in T, and add this edge to the T—thereby increasing the number of vertices in T by one. Figure 1.2 below shows an example of how we add the edges.

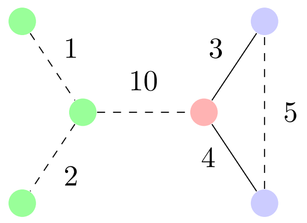

Figure 1.2: Dashed lines are not yet in the MST. We started at the red node, and the blue nodes are also part of T right now.

We'll use a *priority queue* data structure which keeps track of the lightest edge connecting T to each vertex not yet in T. This priority queue will be equipped with three operations:

- `insert`$(elem, key)$ inserts the given $(element, key)$ pair into the queue,
- `decreasekey`$(elem, newkey)$ changes the key of the element *elem* from its current key to $\min(originalkey, newkey)$, and
- `extractmin`$()$ removes the element with the minimum key from the priority queue, and returns the $(elem, key)$ pair.

To implement Prim's algorithm, initially we insert each vertex in $V \setminus \{r\}$ into the priority queue with key ∞, and the root r with key 0. At each step, we'll use `extractmin` to find the vertex u with smallest key, add to the tree, and then for each neighbor of u, say v, we do `decreasekey`$(v, w(uv))$.[1]

Note that by using the standard *binary heap* data structre we can get $O(\log n)$ worst-case time for each operation above. Overall we do m `decreasekey` operations, n `inserts`, and n `extractmins`, with the `decreasekeys` supplying the dominating $O(m \lg n)$ term.

2.3 Borůvka's Algorithm

Unlike Kruskal's and Prim's algorithms, Borůvka's algorithm adds many edges in parallel, and can be implemented without any non-trivial data structures. We simply pick the lightest edge out of each vertex; if edge weights are distinct, this is guaranteed to form a forest.

We now contract each tree in this forest, and then recurse on the resulting graph, keeping track of which edges we chose at each step. Each contraction round takes $O(m)$ work (we will work out the details of this in HW #1), and we're guaranteed to shrink away at least half of the nodes (as each node at least pairs up with one other node, maybe many more). So we have at most $\lceil \lg_2 n \rceil$ rounds of computation, leaving us with $O(m \lg n)$ total work. An example of this is below, in Figure 1.3.

[1]We can optimize slightly by only insering a vertex into the priority queue when it has an edge to the current tree T — while this does not seem particularly useful right now, this will be crucial in the Fredman-Tarjan proof.

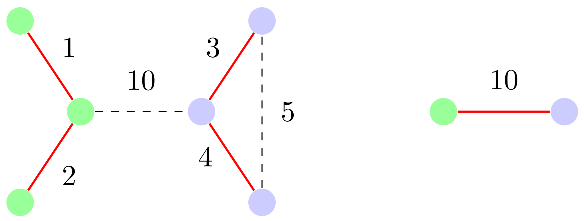

Figure 1.3: The red edges will be chosen and contracted in a single step, yielding the graph on the right, which we recurse on. Colors designate components.

2.4 A Slight Improvement on Prim's

We can actually easily improve the performance of Prim's algorithm by using a more sophisticated data structure, namely by using *Fibonacci heaps* instead of binary heaps to implement the priority queue. Fibonacci heaps (due to Frredman and Tarjan) implement the `insert` and `decreasekey` operations in constant amortized time, and `extractmin` in amortized $O(\lg H)$ time, where H is the maximum number of elements in the heap during the execution. Since we do n `extractmin`s, and $O(m+n)$ other of the other two operations, and the maximum size of the heap is $H \leq n$, this gives us a total cost of $O(m + n \lg n)$.

Note that this is linear on graphs with $m = \Omega(n \log n)$ edges; however, we'd like to get linear-time on all graphs. So the difficult cases are the graphs with $m = o(n \log n)$ edges.

3 Fredman and Tarjan's $O(m \log^* n)$-time Algorithm

Fredman and Tarjan's algorithm builds on Prim's algorithm: the crucial observation uses the following crucial facts.

> The amortized cost of `extractmin` operations in Fibonacci heaps is $O(\log H)$, where H is the maximum size of the heap. And in Prim's algorithm, the size of the heap is just the number of nodes that are adjacent to the current tree. So if the current tree always has a "small boundary", the `extractmin` cost will be low.

How can we maintain the boundary to be small? Once the boundary exceeds a certain size, stop growing the Prim tree, and begin Prim's algorithm anew from a different vertex. Do this until all vertices lie in some tree; then contract these trees (much like Borůvka), and recurse on the smaller graph.

Formally, each round of the algorithm works like this (an example can be seen in Figure 1.4, on the last page of these notes): It depends on a *threshold value* K, to be defined later. Initially all vertices are unmarked.

1. Pick an arbitrary unmarked vertex and start Prim's algorithm from it, creating a tree T. Keep track of the lightest edge from T to each vertex in the neighborhood of T, that is $N(T) := \{v \in V - T : \exists u \in T \text{ s.t. } uv \in E\}$. Note that $N(T)$ may contain vertices that are marked.

2. If at any time $|N(T)| \geq K$, or if T has just added an edge to some vertex that was previously marked, stop and mark all vertices in the current T, and go to step 1.

3. Terminate when each node belongs to some tree.

Let's first note that the runtime of this routine (i.e., one round of the algorithm) is $O(m + n \lg K)$. Each instance of step 1 decreases the number of connected components by 1 so there are at most n instances, and each instance requires a `findmin` on a heap of size at most K, which takes $O(\log K)$ times. At this point, we've successfully identified a forest, where each edge is part of the final MST.

We claim that throughout this routine, every marked vertex u is in a component C such that $\sum_{v \in C} d_v \geq K$. If u became marked because the neighborhood of its component had size at least K, then this is true. Otherwise, u became marked because it entered a component C of marked vertices. Since the vertices of C were marked, $\sum_{v \in C} d_v \geq K$ before u joined, and this sum only increased when u (or other vertices) joined. Thus, if $C_1, \ldots, C_l$ are the components at the end of this routine, we have

$$2m = \sum_v d_v = \sum_{i=1}^{l} \sum_{v \in C_i} d_v \geq \sum_{i=1}^{l} K \geq Kl$$

Thus $l \leq \frac{2m}{K}$, i.e. this routine produced at most $\frac{2m}{K}$ trees.

For the entire algorithm, say we start round i with n_i nodes and $m_i \leq m$ edges; we set K_i accordingly. Now we run the above routine, and contract the trees formed to get a smaller graph with n_{i+1} nodes, $m_{i+1} \leq m_i \leq m$ edges, and then start round $i+1$ etc. Recall that the time to implement the contraction step at the end of round i is $O(m_i + n_i)$, and hence is dominated the routine above which takes $O(m_i + n_i \lg K_i)$ time.

How should we set the thresholds K_i? One clean way is to set

$$K_i := 2^{\frac{2m}{n_i}}$$

which ensures that

$$O(m_i + n_i \lg K_i) = O\left(m_i + n_i \cdot \frac{2m}{n_i}\right) = O(m).$$

Hence, we do linear work in each round. And all that remains is seeing how many levels we have. Recall that n_{i+1} is the number of trees we have in round i. Since we have at most $2m/K_i$ trees, we have $n_{i+1} \leq \frac{2m}{K_i}$. Rewriting, this gives

$$K_i \leq \frac{2m}{n_{i+1}} = \lg K_{i+1} \implies K_{i+1} \geq 2^{K_i}. \tag{1.1}$$

Hence the threshold value exponentiates in each step. (It increases "tetrationally".) Hence after $\log^* n$ rounds, the value of K would be at least n, and hence the above routine would just do Prim's algorithm, and end with a single tree. This means we have at most $\log^* n$ rounds, and hence a total of $O(m \log^* n)$ work.

4 A Linear-Time Randomized Algorithm

We now present the Karger-Klain-Tarjan randomized MST algorithm [KKT95], that runs in $O(m+n)$ expected time.

4.1 Heavy & light edges

The crucial definition for this algorithm is that of *heavy* and *light* edges with respect to some tree T. Take any tree T in the graph[2]. For example, in Figure 1.5 we pick the tree with black edges.

[2]Here when we talk about trees, in fact we mean trees or forests. And MST refers to the minimum spanning forest if the graph is not connected.

The tree T might not be an MST, or even not be connected (i.e. it may be a forest). Now look at any edge e in the graph; just comparing it to the tree edges (and to none of the other edges in G) may allow us to infer that e may not be in the MST for G. E.g., the blue edge on the right is heavier than any other edge in the fundamental cycle it forms with the tree. By the cycle rule this edge cannot be in the MST of the graph. What about the red edge on the left? This edge is not the heaviest edge in the cycle, so we cannot discard it. (In this example, it happens to be in the MST of the graph.) Therefore we can discard some edges which will never appear in the MST based on any given tree of a graph. This brings us to a definition.

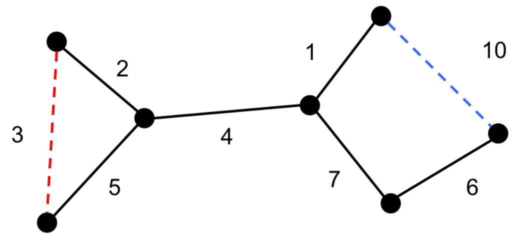

Figure 1.5: Example of heavy and light edges

Definition 1.1. Let T be a forest that is a subgraph of a graph G and $e \in E(G)$. If e creates a cycle when added to T and e is the heaviest edge in the cycle, then we say edge e is *T-heavy*. Otherwise, we say edge e is *T-light*.

Every edge is either T-heavy or T-light. Notice that if an edge e does not belong to a cycle, then e is T-light. If e is in the tree T, e is also T-light. In Figure 1.5, the red edge is T-light, the blue edge is T-heavy, and all the black edges are T-light.

Fact 1.2. *Edge e is T-light $\iff$ $e \in MST(T \cup \{e\})$.*

Fact 1.3. *If T is a MST of G then edge $e \in E(G)$ is T-light $\iff$ $e \in T$.*

Therefore our idea is that in order to pick a good tree/forest T, we find all the T-heavy edges and get rid of them. Hopefully the number of edges remaining is small. The MST must be in the remaining edges. To make this idea work, we want to find some tree T such that there are a lot of T-heavy edges.

Fact 1.4. *For any tree T, the T-light edges contain the MST of the underlying graph G. In other words, any T-heavy edge is also heavy with respect to the MST of the entire graph.*

Will add short proof here.

It is easy to classify a single edge e as being heavy or light in linear time, but the following remarkable theorem is also true:

Theorem 1.5 (MST Verification)**.** *Given a tree $T \subseteq G$, we can output all the T-light edges in $E(G)$ in time $O(|V| + |E|)$.*

For now we assume this theorem; in Appendix B we give some of the ideas in its proof. (This was skipped in class, and is for your enjoyment.)

5 The Karger-Klain-Tarjan Algorithm

The idea behind this algorithm is simple and elegant: we randomly choose half of the edges and find the MST on this "half-of-a-graph". We hope this tree T will have a lot of T-heavy edges, which we can discard; then we can recursively find the MST on the remaining graph. Since both the recursive calls are on smaller graphs, hopefully the runtime will be linear. The formal algorithm is given below as Algorithm 1.

Algorithm 1 KKT(G)

```
1: Run 3 rounds of Borůvka's Algorithm on $G$ to get a graph $G' = (V', E')$ with $n' \leq n/8$ vertices
   and $m' \leq m$ edges.
2: $E_1 \leftarrow$ random sample of $E'$, where each edge is picked independently w.p. 1/2.
3: $T_1 \leftarrow$ KKT($G_1 = (V', E_1)$).
4: $E_2 \leftarrow$ all the $T_1$-light edges in $E'$.
5: $T_2 \leftarrow$ KKT($G_2 = (V', E_2)$).
6: Return $T_2$ (combine with the Borůvka edges chosen in Step 1).
```

Theorem 1.6. *The KKT algorithm returns MST(G).*

Proof. The key idea is Fact 1.4, that discarding heavy edges of any tree in a graph will not change the MST. Formally, we proceed by induction. The base case is trivial, because we will find the MST in Step 1. Otherwise in Step 3 we find a MST T_1 of graph G_1, and in E_2 we discard all the T_1-heavy edges from E' (not only from E_1!!) which cannot possibly be in the MST of G' due to the cycle rule. Or in other words, $\text{MST}(G') \subseteq E_2$. Therefore what Step 5 returns is the MST of G_2, which is also the MST of G'. Combining it with the edges we chose in Step 1, we get the MST of graph G. $\square$

Now we need to deal with the running time. Since pick each edge with probability 1/2, we get:

Claim 1.7. $\mathbf{E}[\#E_1] = \frac{1}{2}m'$.

Claim 1.8. $\mathbf{E}[\#E_2] \leq 2n'$. *This means graph G_2 will hopefully be very sparse.*

How should we analyze $\mathbf{E}[\#E_2]$? In Step 2 of the KKG algorithm, we first flip a coin on each edge, then take all the edges where we got heads and finally in Step 3 we use KKG algorithm on this new graph to get the MST of G_1. Need to make the proof more succinct.

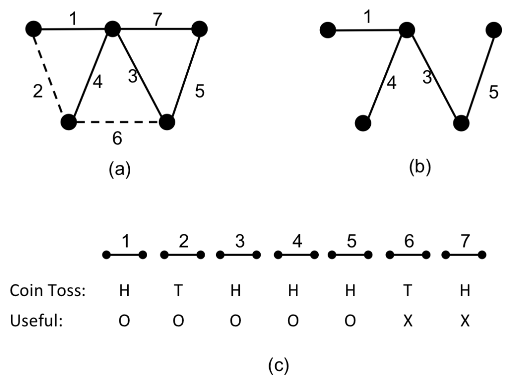

Figure 1.6: Illustration of another order of coin tossing

Let's think of this process in another order. The purpose of Step 3 is to calculate the MST of the new graph. It does not matter if we use the KKT algorithm or some other method – we will still get the same MST. So suppose we use Kruskal's algorithm instead. Then instead of first flipping coins for all edges then calculating the MST, we can do these two things simultaneously: We flip the coins for the edges in an increasing order by weight, and then only consider adding the edge to the graph as in Kruskal's algorithm if we get heads (see Algorithm 2).

Algorithm 2 ModifiedKruskals($G = (V, E)$)

```
1: Sort $E$ in increasing order
2: for all $e \in E$ do
3:     $p \leftarrow 0$ w.p. $\frac{1}{2}$, 1 otherwise
4:     if $p = 0$ then
5:         if $e$ does not form a cycle then
6:             Add $e$ to MST
7:         end if
8:     end if
9: end for
```

We define a coin toss is *useful*, if Kruskal's algorithm will add the edge to the MST if we get heads, and define a coin toss is *useless* if not. For example, Figure 1.6(a) gives a graph with coin toss result. The solid edges are the edges where the coin toss resulted in heads, and the dashed edges are the edges which got tails. Figure 1.6(b) is the MST on all the edges where the coin toss was heads. Now let's check the usefulness of all these coin flips, as in Figure 1.6(c). In the beginning the MST is an empty set. Then the coin flip of the first edge is useful, since it should be added into the MST. The result is heads, so we add this edge into MST. The coin flip of the second edge is useful, since it should be added into the MST. However the coin toss result is tails, so sadly we can not add this edge into MST. The coin flip of the third, forth, fifth edges are all useful, and the toss results are all heads, so we add all of them into the MST. The coin toss of sixth edge is useless, because we will not add this edge into MST regardless of the coin toss since it will create a cycle, and so is the seventh edge.

Claim 1.9. *$T_1 = MST(G_1)$. $e \in E'$ is T_1-light if and only if the coin flip of e is useful.*

Proof. If the coin flip of e is useful, then right before flipping the coin, we will not create a cycle by adding e to the current MST, which means either $e \in T_1$, or there is no cycle in $T_1 \cup \{e\}$, or edge e is not the heaviest edge in the cycle of $T_1 \cup \{e\}$. In any of these cases, edge e is T_1-light. On the other hand, if the coin flip of e is useless, then right before we flip the coin there would be a cycle if we add e into the MST, therefore edge e is T_1-heavy. $\square$

Now we can prove Claim 1.8.

Proof of Claim 1.8.

$$\mathbf{E}[\#E_2] = \mathbf{E}[\#\text{useful coin flips}] \leq \frac{n'-1}{1/2} \leq 2n'$$

The first inequality holds since by each useful coin flip we may add an edge into the MST of G_1 with probability $1/2$, and the final $T_1 = \mathrm{MST}(G_1)$ has at most $n'-1$ edges. This is then equivalent to flipping an unbiased coin until we get $n-1$ heads. Therefore the expectation of the number of useful coin flips is at most $2(n'-1) \leq 2n'$. $\square$

Theorem 1.10. *$KKT(G = (V, E))$ can return the MST in time $O(m + n)$.*

Proof. Let T_G be the expected running time on graph G, and

$$T_{m,n} := \max_{G=(V,E),|V|=n,|E|=m} \{T_G\}$$

In the KKT algorithm, Step 1, 2, 4 and 6 will be done in linear time, so we assume the time cost of Step 1, 2, 4 and 6 is at most cm. Step 3 will spend time T_{G_1}, and Step 5 will spend time T_{G_2}. Then we have

$$T_G \leq cm + T_{G_1} + T_{G_2} \leq cm + T_{m_1,n'} + T_{m_2,n'}$$

Here we assume that $T_{m,n} \leq c(2m+n)$, then

$$\begin{aligned} T_G &= cm + \mathbf{E}[c(2m_1+n')] + \mathbf{E}[c(2m_2+n')] \\ &\leq c(m+m'+6n') \\ &\leq c(2m+n) \end{aligned}$$

The first inequality holds because $\mathbf{E}[m_1] \leq \frac{1}{2}m'$ and $\mathbf{E}[m_2] \leq 2n'$. The second inequality holds because $n' \leq n/8$ and $m' \leq m$. Indeed, we shrunk the graph using Borůvka's algorithm in the first step to ensure $n' \leq 8n$, just to give us some "breathing room". ☐

Acknowledgments

These lecture notes were scribed by C.J. Argue and Xinyu Wu, based on previous scribe notes of Anshu Bansal and Yu Zhao.

References

[Bor26] O. Borůvka. O jistém problému minimálním [About a certain minimal problem]. *Práce Moravské Přírodovědecké Společnosti (in Czech)*, (3):37–58, 1926. 1

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the Association for Computing Machinery*, 47(6):1028–1047, 2000. 1

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596, 1987. 1

[GGST86] H.N. Gabow, Z. Galil, T. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Cornbirratotim*, 6:109–122, 1986. 1

[Hag10] Torben Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *Graph-theoretic concepts in computer science*, volume 5911 of *Lecture Notes in Comput. Sci.*, pages 178–189. Springer, Berlin, 2010. B

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. B.1

[Jar30] V. Jarník. O jistém problému minimálním [About a certain minimal problem]. *Práce Moravské Přírodovědecké Společnosti (in Czech)*, (6):57–63, 1930. 1

[Kin97] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997. B

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. Assoc. Comput. Mach.*, 42(2):321–328, 1995. 1, 4

[Kom85] János Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985. B

[Kru56] J.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. 1

[PR02] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the Association for Computing Machinery*, 49(1):16–34, 2002. 1

[Yao75] A. Yao. An $O(|E| \log\log|V|)$ algorithm for finding minimum spanning trees. *Inf. Process. Lett.*, (4):21–23, 1975. 1

A The Ackermann Function

In 1928, Wilhelm Ackermann defined a fast-growing function that is totally computable but not primitive recursive. Today, we use the term *Ackermann function* $A(m,n)$ to refer to one of many variants that are rapidly-growing and have similar propeties. It seems to arise often in algorithm analysis, so let's briefly discuss it here.

For our purposes, it will be cleanest to define $A(m,n) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ recursively as

$$A(m,n) = \begin{cases} 2n & : \quad m = 1 \\ 2 & : \quad m \geq 1, xn = 1 \\ A(m-1, A(m, n-1)) & : \quad m \geq 2, n \geq 2 \end{cases}$$

Here are the values of $A(m,n)$ for $m, n \leq 4$:

	1	2	3	4	...	n
1	2	4	6	8	...	$2n$
2	2	4	8	16	...	2^n
3	2	2^2	2^{2^2}	$2^{2^{2^2}}$	...	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}$
4	2	4	65536	!!!	...	huge!

We can define the α function to be a functional inverse of the diagonal $A(n,n)$; by construction, $\alpha(\cdot)$ grows extremely slowly. For example, $\alpha(m) \leq 4$ for all $m \leq 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ where the tower has height 65536.

B MST Verification

Now we come back to the implementation of the MST verification blackbox. Here we only consider only trees (not forests), since we can run this algorithm on each connected component separately. Here we refine Theorem 1.5 as follows.

Theorem 1.11 (MST Verification). *Given $T = (V, E)$ where $|V| = n$ and m pairs of vertices (u_i, v_i), we can find the heaviest edge on the path in T from u_i to v_i for all i in $O(m+n)$ time.*

Dixon, Rauch and Tarjan first showed that MST verification was achievable in $O(m+n)$ time. Komlos [Kom85] shows how to do it with $O(m+n)$ comparisons. King [Kin97] showed how to achieve linear time on a RAM machine, and Hagerup [Hag10] presents a simpler linear time algorithm. Here we will present the algorithm of Komlos.

B.1 Balance the tree

Suppose $T = (V, E)$ is a tree on n vertices, and we run Boruvka's algorithm on T. (Let $V_1 = V$ be the original vertices at the beginning of round 1, V_i be the vertices in round i, and say there are L rounds so that $|V_{L+1}| = 1$). We build a tree T' as follows: the vertices are the union of all the

V_i. There is an edge from $v \in V_i$ to $w \in V_{i+1}$ if the vertex v belongs to a component in round i and that is contracted to form $w \in V_{i+1}$; the weight of this edge (v, w) is the min-weight edge out of v in round i. (Note that all vertices in V are now leaves in T'.) Figure 1.7 shows an example of balancing a tree.

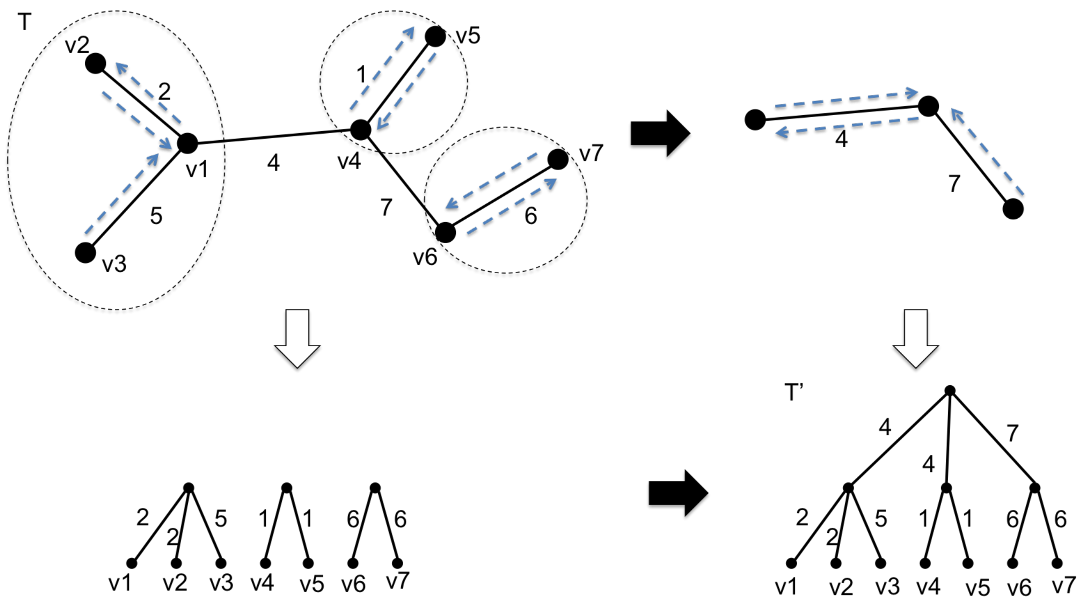

Figure 1.7: Illustration of balancing a tree

Fact 1.12. *For nodes u, v in a tree T, let $\texttt{maxwt}_T(u, v)$ be the maximum weight of an edge on the (unique) path between u, v in the tree T. For all $u, v \in V$, we have*

$$\texttt{maxwt}_T(u, v) = \texttt{maxwt}_{T'}(u, v)$$

This is a problem in Homework 1. In Figure 1.7, we have $\texttt{maxwt}_T(v_1, v_7)$ is 7 which is the weight of edge (v_4, v_6). We can check that $\texttt{maxwt}_{T'}(v_1, v_7)$ is also 7.

Fact 1.13. *T' has at most $\log_2 n$ layers since each vertex has at least 2 children and all leaves of T' are at the bottom (the same) level.*

Here we can make a balanced tree T' based on tree T with some good properties. Then we can just query the heaviest edges of vertex pairs in tree T' instead of T. Another trick is that we can assume that all queries are ancestor-descent queries if we can find the least common ancestor quickly.

Theorem 1.14 (Harel-Tarjan)**.** *Given a tree T, we can preprocess in $O(n)$ time, and answer all LCA queries in $O(1)$ time.*

This was shown by Harel and Tarjan in [HT84].

B.2 Get the answer

Now we have reduced our question to how to answer the ancestor-descent queries efficiently. For each edge $e = (u, v)$ where v is the parent of u, we will look at all queries starting in subtree T_u and ending above vertex v. Say those queries go to $w_1, w_2, \ldots, w_k$. Then the "query string" is

$Q_e = (w_1, w_2, \ldots, w_k)$. Then we need to calculate the "answer string" $A_e = (a_1, a_2, \cdots, a_k)$ where a_i is the maximal weight among the edges between w_i and u.

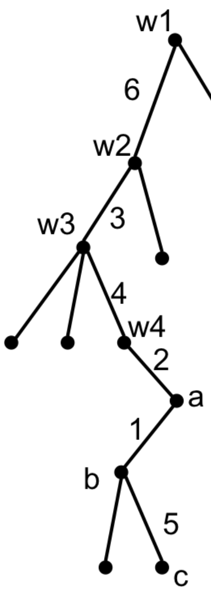

Figure 1.8: Illustration of queries and answers

Figure 1.8 gives us an example. Suppose $Q_{(b,a)} = (w_1, w_3, w_4)$, which means there are three queries starting from some vertices in the subtree of b and ending to w_1, w_3, w_4. Then we get the answer

$$A_{(b,a)} = (a_1, a_3, a_4) = (6, 4, 4)$$

since the maximal weight of an edge on the path from w_1 to b is the weight of edge (w_1, w_2), and the maximal weight of an edge on the path from either w_3, w_4 to b is from the weight of edge (w_3, w_4).

Given the answer of $A_{(b,a)}$, how should we find the answer of $A_{(c,b)}$ efficiently? Say $Q_{(c,b)} = (w_1, w_4, b)$. Comparing with $Q_{(b,a)}$, we may lose some queries since they are from some other children of vertex b, and we may have some other queries ending at b which will not contain in $Q_{(b,a)}$. The easiest way is to update every query. Suppose the weight of edge (c, b) is t. Then we can calculate

$$A_{(c,b)} = (\max\{a_1, t\}, \max\{a_4, t\}, t) = (\max\{6, 5\}, \max\{4, 5\}, 5)$$

The trick is that if in $Q_e = (w_1, w_2, \ldots, w_k)$, $w_1, w_2, \ldots, w_k$ is sorted from the top to the bottom, then the answers $A_e = (a_1, a_2, \cdots, a_k)$ should be non-increasing, $a_1 \geq a_2 \geq \cdots \geq a_k$. Therefore we can do binary search to reduce the number of comparisons.

Claim 1.15. *Given the question string Q_e for $e = (u, v)$ where v is the parent of u, the answer string A_e for e , we can compute answers $A_{e'}$ for $e' = (w, u)$ where w is a child of u, within time $\lceil \log(|A_e| + 1) \rceil$.*

Theorem 1.16. *The total number of comparisons for all queries*

$$t \leq \sum_e \log\left(|Q_e| + 1\right) \leq O(m + n)$$

Proof. Assume the number of edges in level i is n_i. Here the level is counted from the bottom to the top, the edges connected to leaves are at level 0.

$$\begin{aligned}
\sum_{e \in \text{level } i} \log_2(1+|Q_e|) &= n_i \operatorname*{avg}_{e \in \text{level } i}(\log_2(1+|Q_e|)) \\
&\le n_i \log_2\left(1 + \operatorname*{avg}_{e \in \text{level } i}(|Q_e|)\right) \\
&= n_i \log_2\left(1 + \frac{\sum_{e \in \text{level } i}|Q_e|}{n_i}\right) \\
&\le n_i \log_2\left(1 + \frac{m}{n_i}\right) \\
&= n_i\left(\log_2 \frac{m+n}{n} + \log_2 \frac{n}{n_i}\right)
\end{aligned}$$

The first inequality holds by Jensen's inequality and convexity of the function $\log_2(1+x)$. The second inequality holds since the number of all queries is m and each query will only appear on at most one edge on any particular level. Therefore we have

$$\begin{aligned}
t &= \sum_e \log_2(1+|Q_e|) \\
&= \sum_i \sum_{e \in \text{level } i} \log_2(1+|Q_e|) \\
&\le \sum_i n_i\left(\log_2 \frac{m+n}{n} + \log_2 \frac{n}{n_i}\right) \\
&= n\log_2 \frac{m+n}{n} + \sum_i n_i \log_2 \frac{n}{n_i} \\
&\le n\log_2 \frac{m+n}{n} + cn \\
&= O(n \log \frac{m+n}{n} + n) = O(m+n)
\end{aligned}$$

□

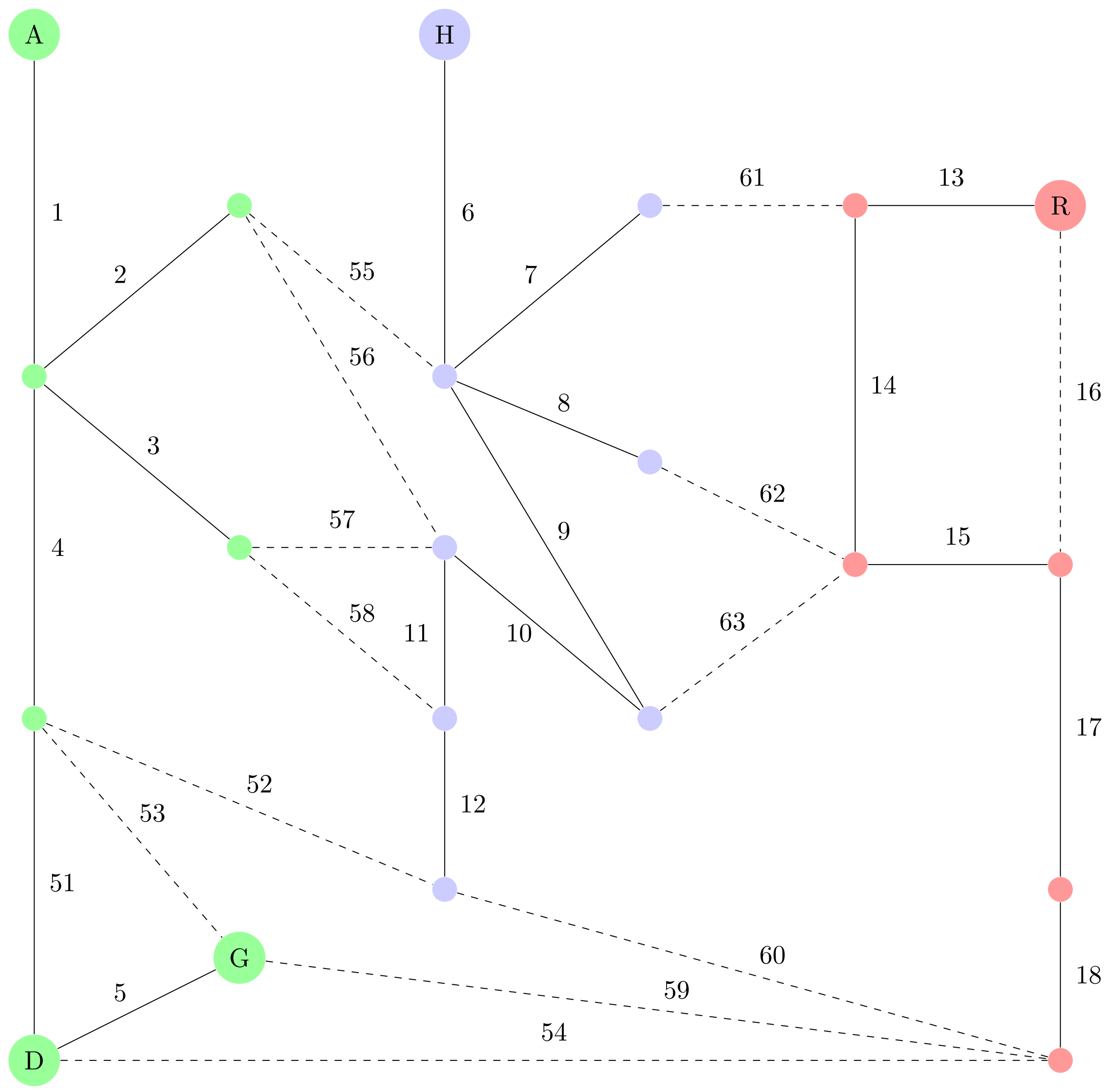

Figure 1.4: We set $K = 6$. We begin Prim's algorithm at vertices A, H, R, and D (in that order). Note that although D begins as its own component, it stops when it joins with tree A. Dashed edges are not chosen in this step (though may be chosen in the next recursive call), colors denote trees.