1
Minimum Spanning Trees

We start our exploration of algorithms this semester with a classic problem in algorithmic graph theory: given a graph with edge weights, finding a spanning tree of minimum total weight. Why this problem?

1. It is a rich problem with lots of structure, which is easy to discover, and which we can use to get good algorithms.

2. These algorithms allows us to showcase the interplay between data structures and algorithms—while we will have too much time to spend exploring data structures in this course, they can be crucial in getting improved running times.

3. And finally, some of it is for nostalgia: algorithms for this problem have been known for almost a hundred years now!

1.1 Minimum Spanning Trees: History

In minimum spanning tree problem, the input is an undirected connected graph $G = (V, E)$ with n nodes and m edges, where the edges have weights $w(e) \in \mathbb{R}$. The goal is to find a spanning tree of the graph with the minimum total edge-weight. If the graph G is disconnected, we get a *spanning forest* 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:

A spanning tree/forest is defined to be an acyclic subgraph T that is inclusion-wise maximal, i.e., adding any edge in $G \setminus T$ would create a cycle.

- Otakar Borůvka gave the first known MST algorithm in 1926; it was subsequently rediscovered by Gustave Choquet (1938), Georges Sollin (1965), and several others. Vojtěch Jarník gave his algorithm in 1930, and it was independently discovered by Robert Prim ('57) and Edsger Dijkstra ('59), among others. Joseph Kruskal

Otakar Borůvka (1926)

Vojtěch Jarník (1930)

J.B. Kruskal, Jr. (1956)

gave his algorithm in '56; this was rediscovered by Loberman and Weinberger in 1957. All these can easily be implemented in $O(m \log n)$ time; we will discuss these in this lecture.

Both Prim and Kruskal refer to Borůvka's paper, but say it is "unnecesarily elaborate". However, while Borůvka's paper *is* written in a complicated fashion, but his essential ideas are very clean.

- In 1975, Andy Yao achieved a runtime of $O(m \log \log n)$. His algorithm builds on Borůvka's algorithm (which he attributes to Sollin), and uses as a subroutine the linear-time algorithm for median-finding, which had only recently been invented in 1974. We will work through Yao's algorithm in HW#1.

Andrew Chi-Chih Yao (1975)

- In 1984, Michael Fredman and Bob Tarjan gave an $O(m \log^* n)$ time algorithm, based on their Fibonacci heaps data structure. Here $\log^*$ is the *iterated logarithm* function, and denotes the number of times we must take logarithms before the argument becomes smaller than 1. The actual runtime is a bit more nuanced, which we will not bother with today.

  This result was soon improved by Gabow, Galil, Spencer, and Tarjan ('86) to get an $O(m \log \log^* n)$ runtime—note the logarithm *applied to* the iterated logarithm.

Michael L. Fredman and Robert E. Tarjan (1987)

Gabow, Galil, Spencer, and Tarjan (1986)

- In 1995, David Karger, Phil Klein and Bob Tarjan finally got the holy grail of $O(m)$ time! ...but it was a randomized algorithm, so the search for a deterministic linear-time algorithm continued.

Karger, Klein, and Tarjan (1995)

- In 1997, Bernard Chazelle gave an $O(m\alpha(n))$-time deterministic algorithm. Here $\alpha(n)$ is the inverse Ackermann function (defined in §1.6). This function grows extremely slowly, even slower than the iterated logarithm function. However, it still goes to infinity as $n \to \infty$, so we still don't have a deterministic linear-time MST algorithm.

Chazelle (1997)

- In 1998, Seth Pettie and Vijaya Ramachandran gave an *optimal* algorithm for computing minimum spanning trees—however, we don't know its runtime! More formally, they show that if there exists an algorithm which uses $MST^*(m,n)$ comparisons to find MSTs on all graphs with m edges and n nodes, the Pettie-Ramachandran algorithm will run in time $O(MST^*(m,n))$.)

Pettie and Ramachandran (1998)

This was part of Seth's Ph.D. thesis, and Vijaya was his advisor.

In this chapter, we'll go through the three classics: Jarnik/Prim's, Kruskal's, and Borůvka's algorithms. Then we will discuss Fredman and Tarjan's algorithm, and finally present Karger, Klein, and Tarjan's randomized algorithm. This will lead us to discuss another intriguing question: ***how do we verify whether a given tree is an MST?***

1.1.1 *Two Assumptions*

For the rest of this chapter, assume that the edge weights are *distinct*. This does not change things in any essential way, but it simplifies some of the statements, because distinct edge weights imply that the MST is unique. (Exercise: prove this!) Also assume the graph is simple, and hence $m \leq \binom{n}{2}$; you can delete all self-loops and remove all-but-the-lightest from any collection of parallel edges, all by preprocessing the graph in linear time.

1.1.2 *The Cut and Cycle Rules*

Most of these algorithms rely on two rules: the ***cut rule*** (known in Bob Tarjan's monograph as the blue rule) and the ***cycle rule*** (or the red rule). Recall that a *cut* in the graph is a partition of the vertices into two non-empty sets $(S, \bar{S} = V \setminus S)$, and an edge *crosses* this cut if its two endpoints lie in different sets.

Tarjan (1983)

Theorem 1.1 (Cut Rule)**.** *For any cut of the graph, 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 $e = \{u, v\}$ be the minimum-weight edge that crosses the cut defined by $(S, \bar{S})$ (W.l.o.g., $u \in S, v \notin S$), and let T be a spanning tree not containing e. Then $T \cup \{e\}$ contains a unique cycle C. Since C crosses the cut $(S, \bar{S})$ once (namely at e), it must also cross at another edge e'. But $w(e') > w(e)$, so $T' = (T - \{e'\}) \cup \{e\}$ is a lower-weight tree than T, so T is not the MST. Since T was an arbitrary spanning tree not containing e, the MST must contain e. □

Theorem 1.2 (Cycle Rule)**.** *For any cycle in G, 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 the choice of e we have $w(e') < w(e)$, so T' is a lower-weight spanning tree than T, a contradiction. □

To find a minimum spanning tree, we repeated apply whichever of these rules we like. E.g., we choose some cut, use the cut rule to designate the lightest edge in it as belonging to the MST by coloring it blue (hence the name). Or we choose a cycle which contains no

This edge e cannot have previously been colored red—this follows from the above lemmas. Or more directly, any cycle crosses any cut an even number of times, so a cycle containing e also contains another edge f in the cut, which is heavier.

red edge, use the cycle rule to mark the heaviest edge as not being in the MST, and color it red. (Again, this edge cannot already be blue for similar reasons.) And if either of the rules is not applicable, we are done. Indeed, if we cannot apply the blue rule, the blue edges cross every cut, and hence form a spanning tree, which must be the MST. Similarly, once the non-red edges do not contain a cycle, they form a spanning tree, which must be the MST. All known algorithms differ only in their choice of cut/cycle, and how they find these fast. Indeed, all the deterministic algorithms we discuss today will just use the cut rule, whereas the randomized algorithm will use the cycle rule as well.

1.2 *The Classical Algorithms*

1.2.1 *Kruskal's Algorithm*

For Kruskal's Algorithm, first sort all the edges such that $w(e_1) < w(e_2) < \cdots < w(e_m)$. This takes $O(m \log m) = O(m \log n)$ time. Start with all edges being uncolored, and iterate through the edges in the sorted order, coloring an edge blue if and only if it connects two vertices which are not currently in the same blue component. Figure 1.1 gives an example of how edges are added.

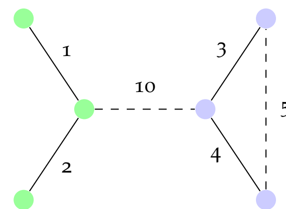

Figure 1.1: Dashed lines are not yet in the MST. Note that 5 will be analyzed next, but will not be added. 10 will be added. Colors designate connected components.

To keep track of which vertex is in which component, use a ***disjoint set union-find*** data structure. This data structure 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 data structure which allows us to do m operations in $O(m\,\alpha(m))$ ***amortized*** time, where $\alpha(\cdot)$ is the inverse Ackermann function mentioned above. Note that the naïve implementation of Kruskal's algorithm spends $O(m \log m) = O(m \log n)$ time to sort the edges, and then performs n `makeset`s, m `find`s, and $n - 1$ `union` operations, the total runtime is $O(m \log n + m\,\alpha(m))$, which is dominated by the $O(m \log n)$ term.

1.2.2 *The Jarnik/Prim Algorithm*

For the Jarnik/Prim algorithm, first take an arbitrary root vertex r to start our MST T. At each iteration, take the cheapest edge connecting

of our current tree T of blue edges to some vertex not yet in T, and color it blue—thereby adding this edge to T and increasing its size by one. Figure 1.2 below shows an example of how we edges are added.

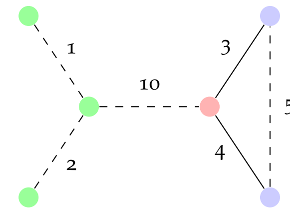

Figure 1.2: Dashed lines are not yet in the MST. We started at the red node, and the blue nodes are also part of T right now.

We'll use a ***priority queue*** data structure which keeps track of the lightest edge connecting T to each vertex not yet in T. A priority queue data structure is equipped with (at least) 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.

Note that by using the standard ***binary heap*** data structure we can get $O(\log n)$ worst-case time for each priority queue operation above.

To implement the Jarnik/Prim algorithm, we initially insert each vertex in $V \setminus \{r\}$ into the priority queue with key ∞, and the root r with key 0. The key of an node v denotes the weight of the least-weight edge from a node in T to v; it is zero if $v \in T$, and ∞ if there are no edges yet from nodes in T to v. At each step, use `extractmin` to find the vertex u with smallest key, and add u to the tree using this edge. Then for each neighbor of u, say v, do `decreasekey`$(v, w(\{u, v\}))$. Overall we do m `decreasekey` operations, n `inserts`, and n `extractmins`, with the `decreasekeys` supplying the dominating $O(m \log n)$ term.

We can optimize slightly by inserting a vertex into the priority queue only when it has an edge to the current tree T. This does not seem particularly useful right now, but will be crucial in the Fredman-Tarjan proof.

1.2.3 Borůvka's Algorithm

Unlike Kruskal's and Jarnik/Prim's algorithms, Borůvka's algorithm adds many edges in parallel, and can be implemented without any non-trivial data structures. In a "round", simply take the lightest edge out of each vertex and color it blue; these edges are guaranteed to form a forest if edge-weights are distinct. (Exercise: why?)

Now contract the blue edges and recurse on the resulting graph. At the end, when the resulting graph is a single vertex, uncontract all the edges to get the MST. Each round can be implemented in $O(m)$ work: we will work out the details of this in HW #1. Moreover, we're guaranteed to shrink away at least half of the nodes (as each node at least pairs up with one other node), and maybe many more if we are lucky. So we have at most $\lceil \log_2 n \rceil$ rounds of computation, leaving us with $O(m \log n)$ total work.

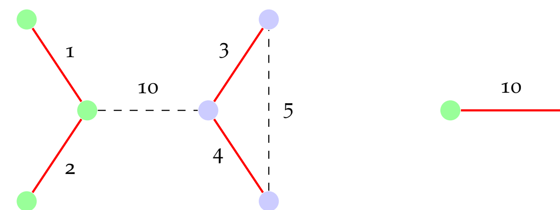

Figure 1.3: The red edges will be chosen and contracted in a single step, yielding the graph on the right, which we recurse on. Colors designate components.

1.2.4 A Slight Improvement on Jarnik/Prim

We can actually easily improve the performance of Jarnik/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 (invented by Fredman and Tarjan) implement the `insert` and `decreasekey` operations in constant amortized time, and `extractmin` in amortized $O(\log H)$ time, where H is the maximum number of elements in the heap during the execution. Since we do n `extractmins`, and $O(m+n)$ of the other two operations, and the maximum size of the heap is at most n, this gives us a total cost of $O(m + n \log n)$.

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

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

Fredman and Tarjan's algorithm builds on Jarnik/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. Moreover, in Jarnik/Prim's algorithm, the size of the heap is just the number of nodes that are adjacent to the current tree T. So if the current tree always has a "small boundary", the `extractmin` cost will be low.

How can we maintain the boundary to be smaller than some threshold K? Simple: Once the boundary exceeds K, stop growing the Prim tree, and begin Jarnik/Prim's algorithm anew from a different vertex. Do this until we have a forest where all vertices lie in some tree; then contract these trees (much like Borůvka), and recurse on the smaller graph. Before we formally define the algorithm, here's an example.

Formally, in each round of the algorithm, all vertices start as unmarked.

1. Pick an arbitrary unmarked vertex and start Jarnik/Prim's algorithm from it, creating a tree T. Keep track of the lightest edge from T to each vertex in the neighborhood $N(T)$ of T, where $N(T) := \{v \in V - T \mid \exists u \in T \text{ s.t. } \{u, v\} \in E\}$. 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.

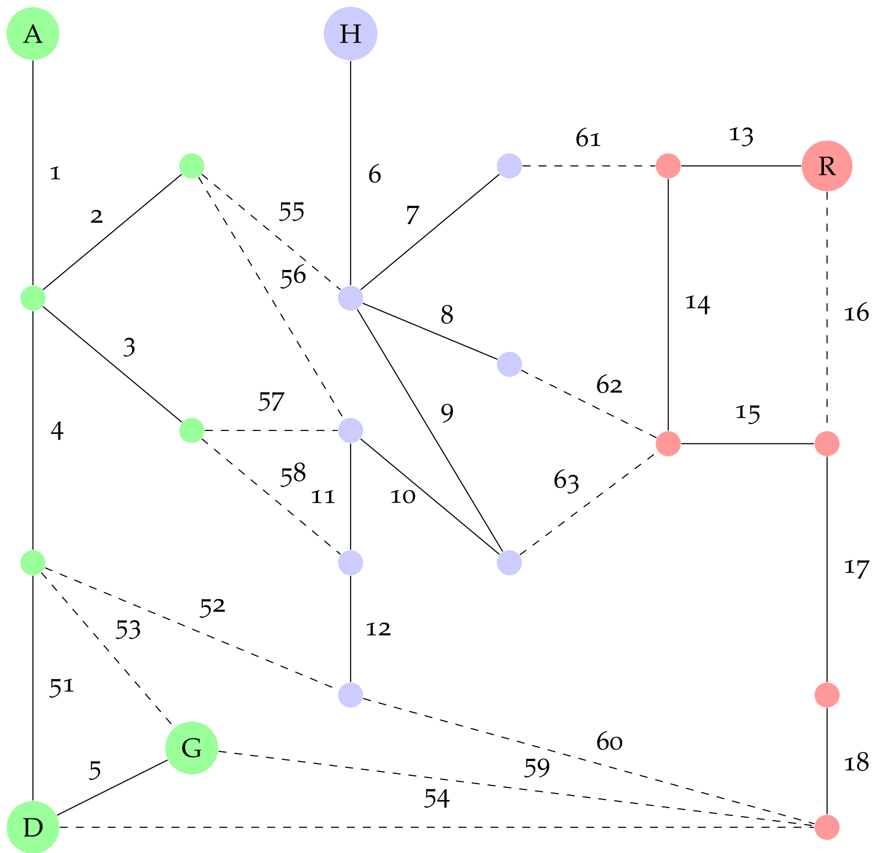

Figure 1.4: We begin at vertices A, H, R, and D (in that order) with $K = 6$. Although D begins as its own component, it stops when it joins with tree A. Dashed edges are not chosen in this step (though they may be chosen in the next recursive call), and colors denote trees.

3. Terminate when each node belongs to some tree.

Let's first note that the runtime of one round of the algorithm is $O(m + n \log K)$. Each edge is considered at most twice, once from each endpoint, giving us the $O(m)$ term. Each time we grow the current tree in step 1, the number of connected components decreases by 1, so there are at most n such steps. Each step calls `findmin` on a heap of size at most K, which takes $O(\log K)$ times. Hence, at the end of this round, we've successfully identified a forest, each edge of which is part of the final MST, in $O(m + n \log K)$ time.

Let d_v be the degree of the vertex v in the graph we consider in this round. We claim that every marked vertex u belongs to a component C such that $\sum_{v \in C} d_v \geq K$. Indeed, 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 (and 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.

The choice of K will change over the course of the algorithm. How should we set the thresholds K_i? Say we start round i with n_i nodes and $m_i \leq m$ edges. One clean way is to set

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

which ensures that

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

In turn, this means the number of trees, and hence the number of nodes n_{i+1} in the next round, is at most $\frac{2m_i}{K_i} \leq \frac{2m}{K_i}$. The number of edges is $m_{i+1} \leq m_i \leq m$. Rewriting, this gives

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

Hence the threshold value exponentiates in each step. Hence after $\log^* n$ rounds, the value of K would be at least n, and we would just run Jarnik/Prim's algorithm to completion, ending with a single tree. This means we have at most $\log^* n$ rounds, and a total of $O(m \log^* n)$ work.

The threshold increases "tetrationally".

In retrospect, I don't know whether to consider the Fredman-Tarjan algorithm as being trivial (once we have Fibonacci heaps) or

being devilishly clever. I think it is the latter (and that is the beauty of the best algorithms). Indeed, there's a lovely idea—of keeping the neighborhoods small at the beginning when there's a lot of work to do, but allow them to grow quickly, as the graph collapses. It is quite non-obvious at the start, and obvious in hindsight. And once you see it, you cannot un-see it!

1.4 *A Linear-Time Randomized Algorithm*

Another algorithm that is extremely clever but almost obvious in hindsight is the the Karger-Klein-Tarjan randomized MST algorithm, which runs in $O(m+n)$ expected time. The new idea here is to compute a "rough approximation" to the MST, use that to throw away many edges using the *cycle rule*, and then recurse on the rest of the graph.

Karger, Klein, and Tarjan (1995)

A version of this algorithm was proposed by Karger in 1992, but he only obtained an $O(m + n \log n)$ runtime. The enhancement to linear time was given by Klein and Tarjan at the STOC 1994 conference; the combined paper is cited above.

1.4.1 *Heavy & light edges*

The crucial definition is that of edges being *heavy* and *light* with respect to some forest F.

Definition 1.3. Let F be a forest that is a subgraph of G. An edge $e \in E(G)$ is ***F-heavy*** if e creates a cycle when added to F, and moreover it is the heaviest edge in this cycle. Otherwise, we say edge e is ***F-light***.

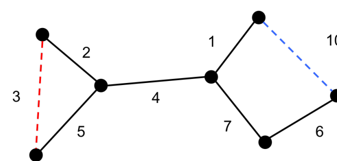

Figure 1.5: Fix this figure, make it interesting. Every edge in F is F-light, as are the edges on the left, and also those going between the components. The edge on the right is F-heavy.

The next facts follow from the definition:

Fact 1.4. Edge e is F-light $\iff$ $e \in \mathrm{MST}(F \cup \{e\})$.

Fact 1.5 (Completeness). If T is an MST of G then edge $e \in E(G)$ is T-light if and only if $e \in T$.

Fact 1.6 (Soundness). For any forest F, the F-light edges contain the MST of the underlying graph G. In other words, any F-heavy edge is also heavy with respect to the MST of the entire graph.

This suggests a clear strategy: pick a forest F from the current edges, and discard all the F-heavy edges. Hopefully the number of edges remaining is small. By Fact 1.6 these edges contain the MST of G, so repeat the process on them. To make this idea work, we want a forest F with many F-heavy edges. The catch is that a forest has many heavy edges if it has small weight, if there are many off-forest edges forming cycles where they are the heaviest edges. Indeed, one such forest in the MST T^* of G: Fact 1.5 shows there are $m - (n-1)$ many T^*-heavy edges, the maximum possible. How do we find some similarly good tree/forest, but in linear time?

A second issue is to classify edges as light/heavy, given a forest F. It is easy to classify a single edge e in linear time, but the following remarkable theorem is also true:

Theorem 1.7 (MST Verification)**.** *Given a forest $F \subseteq G$, we can output the set of all F-light edges in G in time $O(m+n)$.*

This MST verification algorithm itself uses several interesting ideas; we discuss some of them in Section 1.5. But for now, let us use it to give the randomized linear-time MST algorithm.

1.4.2 *The Randomized MST Algorithm*

The idea is simple and elegant: randomly choose half of the edges and find the minimum-weight spanning forest F on this "half-of-a-graph". This forest F should have many F-heavy edges; we discard these and recursively find the MST on the remaining graph. Since both the recursive calls are on smaller graphs, hopefully the runtime will be linear.

The random subgraph may not be connected, so the maximum spanning forest is obtained by finding the MST for each of its connected components.

The actual algorithm below has just one extra step: we first run a few rounds of Borůvka's algorithm to force a reduction in the number of vertices, and then do the steps above.

Algorithm 1: KKT(G)

1.1 Run 3 rounds of Borůvka's Algorithm on G, contracting the chosen edges to get a graph $G' = (V', E')$ with $n' \leq n/8$ vertices and $m' \leq m$ edges.
1.2 If G' has a single vertex, return any chosen edges.
1.3 $E_1 \leftarrow$ random sample of E', each edge picked indep. w.p. $1/2$.
1.4 $F_1 \leftarrow$ KKT($G_1 = (V', E_1)$).
1.5 $E_2 \leftarrow$ all the F_1-light edges in E'.
1.6 $F_2 \leftarrow$ KKT($G_2 = (V', E_2)$).
1.7 **return** F_2 (combined with Borůvka edges chosen in Step 1).

Theorem 1.8. *The KKT algorithm returns MST(G).*

Proof. This follows from Fact 1.6, that discarding heavy edges of any forest F in a graph does not change the MST. Indeed, the MST on G_2 is the same as the MST on G', since the discarded F_1-heavy edges cannot be in $MST(G')$ because of Fact 1.6. Adding back the edges picked by Borůvka's algorithm in Step 1 gives the MST on G, by the cut rule. ☐

Now we need to bound the running time. The following two claims formalize the intuition that we recurse on "smaller" subgraphs:

Claim 1.9. $\mathbb{E}[\#E_1] = \frac{1}{2}m'$.

Claim 1.10. $\mathbb{E}[\#E_2] \leq 2n'$.

The first claim is easy to prove, using linearity of expectations, and that each edge is picked with probability 1/2. The proof of Claim 1.10

is also short, but before we prove it, let us complete the proof of the linear running time.

Theorem 1.11. *The KKT algorithm, run on a graph with m edges and n vertices, terminates in expected time $O(m+n)$.*

Proof. Let $\mathbf{T}_G$ be the ***expected*** running time on graph G, and

$$\mathbf{T}_{m,n} := \max_{G=(V,E),|V|=n,|E|=m} \{T_G\}.$$

In the KKT algorithm, Step 1, 2, 3, 5, and 7 can each be done in linear time: indeed, the only non-trivial part is Step 4, for which we use Theorem 1.7. Let the total time for these steps be at most $c(m+n)$. Steps 4 and 6 require time T_{G_1} and T_{G_2} respectively. Then we have

$$\mathbf{T}_G \le c(m+n) + \mathbb{E}[\mathbf{T}_{G_1} + \mathbf{T}_{G_2}] \le cm + \mathbb{E}[\mathbf{T}_{m_1,n'} + \mathbf{T}_{m_2,n'}],$$

where $m_1 = \#E_1$ and $m_2 = \#E_2$ are both random variables. Inductively assume that $\mathbf{T}_{m,n} \le 2c(m+n)$, then

$$\begin{aligned}
\mathbf{T}_G &\le c(m+n) + \mathbb{E}[2c(m_1+n')] + \mathbb{E}[2c(m_2+n')] \\
&\le c(m+n) + c(m'+2n') + 2c(2n'+n') = c(m+m'+n+8n') \\
&\le 2c(m+n).
\end{aligned}$$

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

Now we prove Claim 1.10. Recall that we randomly subsample the edges of G' to get G_1, compute its maximum spanning forest F_1, and now we want to bound the expected number of edges in G' that are F_1-light. The key to the proof is to do all these steps together, deferring the random decisions to when we really need them. This makes it apparent which edges are light, making them easy to count.

This idea to defer looking at the random choices of the algorithm is often called the ***principle of deferred decisions***.

Proof of Claim 1.10. For the sake of the proof, we can use any correct algorithm to compute F_1, so let us use Kruskal's algorithm. Moreover, let's run a *lazy* version as follows: first sort all the edges in E', and not just those in $E_1 \subseteq E'$, and consider then in increasing order of weights. Now if the currently considered edge e_i connects two different trees in the current blue forest, call e_i *useful* and flip an independent unbiased coin: if the coin comes up "heads", color e_i blue and add it to F_1, else color e_i red. The crucial observation is that this process produces a forest from the same distribution as first choosing G_1 and then computing F_1 by running Kruskal's algorithm on it.

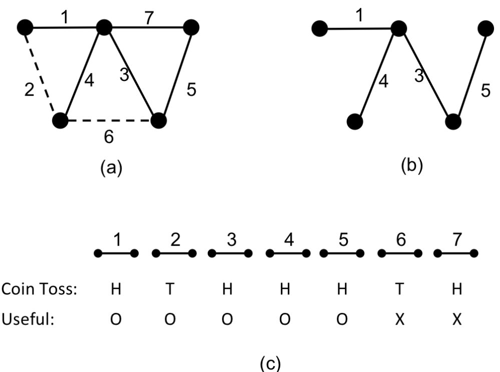

Figure 1.6: Illustration of another order of coin tossing

Now, let us consider the lazy process again: which edges are F_1-light? We claim that these are precisely the useful edges. Indeed, any non-useful edge e_j forms a cycle with the previously chosen blue edges in F_1, and it is the heaviest edge on that cycle. Hence e_j does not belong to $MST(F_1 \cup \{e_j\})$, so it is F_1-heavy by Fact 1.4. And a useful edge e_i would belong to $MST(F_1 \cup \{e_i\})$, since running Kruskal's algorithm on $F_1 \cup \{e_i\}$ would see that e_i connects two different blue components and hence would pick it.

Finally, how many useful edges are there, in expectation? Let's abstract away the details: we're running a process that periodically asks us to flip an independent unbiased coin. Since each time we see a heads, we add an edge to the forest, so we definitely stop when we see $n' - 1$ heads. (We may stop earlier, in case the process runs out of edges, but then we can pad the random sequence to flip some more coins.) Since the coins are independent and unbiased, the expected number of flips until we see $n' - 1$ heads is exactly $2(n' - 1)$. This proves Claim 1.10. □

That's it. The algorithm and proof are both short and slick and beautiful: this result is a real gem. I think it's an algorithm from ***The Book***. The one slight annoyance with the algorithm is the relative complexity of the MST verification algorithm, which we use to find the F_1-light edges in linear time. Nonetheless, these verification algorithms also contain many nice ideas, which we now discuss.

Paul Erdős claimed that God has "The Book" which contains the most elegant proof of each mathematical theorem.

The current verification algorithms are deterministic; can we use randomness to simplify these as well?

1.5 *Optional: MST Verification*

We now come back to the implementation of the MST verification procedure. Here we only consider only trees (not forests), since we can run this algorithm separately on each tree in the forest and incur only a linear extra cost. Let us refine Theorem 1.7 as follows.

Theorem 1.12 (MST Verification). *Given a tree $T = (V, E)$ where $|V| = n$, and m pairs of vertices (y_i, z_i) in T, we can find the heaviest edge on the unique y_i-to-z_i path in T for all i, in $O(m + n)$ time.*

Since the edge $\{y_i, z_i\}$ is T-heavy precisely if it is heavier than the heaviest edge on the corresponding tree path, this also proves Theorem 1.7. Observe that the query pairs are given up-front: there is an inverse-Ackermann-type lower bound for the problem where the queries arrive online.

Pettie (2006)

How do we get such a linear-time algorithm? *A priori*, it is not easy to even show a ***query-complexity upper bound***: that there exists a procedure that performs *a linear number of edge-weight comparisons* to solve the MST verification problem. This problem was solved

by János Komlós. His result was subsequently made algorithmic ("how do you find (in linear time) which linear number of queries to make?") by Brendan Dixon, Monika Rauch (now Monika Henzinger) and Bob Tarjan. This algorithm was futher simplified by Valerie King [1], and by Thomas Hagerup [2]. We will just discuss Komlós's query-complexity bound.

Komlos (1985)

[1]

[2]

1.5.1 A Simpler Case

To start developing the algortihm, it helps to consider special cases: e.g., what if the tree is a complete binary tree? Let's assume something slightly less restrictive than a complete binary tree: suppose tree T is rooted at some node r, all internal nodes have at least 2 children, and all its leaves are at the same level. Moreover, all queries $\{y_i, z_i\}$ are for pairs where y_i is a leaf and z_i its ancestor.

A node v is an ***ancestor*** of u if v lies on the unique path from u to the root; then u is a ***descendent*** of v.

Now for an edge (u, v) of the tree, where v is the parent and u the child, consider all queries starting within subtree T_u and ending at vertex v or higher. Say these queries go from some leaves inside T_v up to $w_1, w_2, \ldots, w_k$, where w_1 is closest to the root. Define the "query string"

$$Q_e := (w_1, w_2, \ldots, w_k).$$

We want to calculate the "answer string"

$$A_e := (a_1, a_2, \cdots, a_k),$$

where a_i is the largest weight among the edges between w_i and u.

Now given the answer string $A_{(b,a)}$, we can get the answer string for a child edge. In the example, say the query string for edge (c, b) is $Q_{(c,b)} = (w_1, w_4, b)$. We have lost some queries that were in $Q_{(b,a)}$, (e.g., for w_3) but we now have a query ending at b. To get $A_{(b,a)}$ we can drop the lost queries, add in the entry for b, and also take the component-wise maximum with the weight of (c, b) itself. E.g., if (c, b) has weight t, then

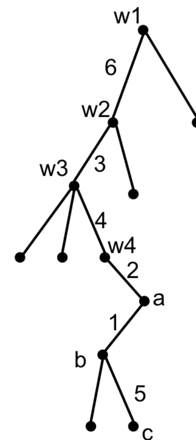

Figure 1.7: Query string $Q_{(b,a)} = (w_1, w_3, w_4)$ means there are three queries starting from vertices in T_b and ending at w_1, w_3, w_4. The answer string is $A_{(b,a)} = (a_1, a_3, a_4) = (6, 4, 4)$.

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

Naïvely this would require us to compare the weight $w_{(c,b)}$ with all the entries in the answer string, incurring $|A_{e'}|$ comparisons. The crucial observation is this: since the nodes in the query string are sorted from top to bottom, the answers must be non-increasing: i.e., $a_1 \geq a_2 \geq \cdots \geq a_k$. Therefore we can do binary search to reduce the number of comparisons between edge-weights. Indeed, given the answer string for some edge e, we can compute answers $A_{e'}$ for a child edge e' using at most $\lceil \log(|A_{e'}| + 1) \rceil$ comparisons. This will be enough to prove the result.

Claim 1.13. The total number of comparisons for all queries is at most

$$\sum_e \log(|Q_e| + 1) \le O(n + n \log \frac{m+n}{n}) = O(m+n).$$

Proof. Let the number of edges at height i be n_i, where height 1 corresponds to edges incident to the leaves.

$$\begin{aligned}
\sum_{e \in \text{height } i} \log_2(1 + |Q_e|) &= n_i \operatorname{avg}_{e \in \text{height } i}(\log_2(1 + |Q_e|)) \\
&\le n_i \log_2 \left(1 + \operatorname{avg}_{e \in \text{height } i}(|Q_e|)\right) \\
&\le n_i \log_2 \left(1 + \frac{m}{n_i}\right) \\
&= n_i \left(\log_2 \frac{m+n}{4n} + \log_2 \frac{4n}{n_i}\right).
\end{aligned}$$

The first inequality uses concavity of the function $\log_2(1+x)$, and Jensen's inequality. The second holds because each of the m queries can only appear on at most one edge, so the average "load" is at most m/n_i. Summing the first term over all heights gives $n \log_2 \frac{m+n}{4n} = O(m)$.

Jensen's inequality says that for any convex function f and any random variable X, $\mathbb{E}[f(X)] \ge f(\mathbb{E}[X])$. Concavity requires flipping the sign, of course.

To bound the second term (summed over all heights), recall that each node has at least two children, so the number of edges at least doubles each time the height decreases. Hence, $n_i \le n/2^{i-1}$, and

$$\sum_{i \ge 1} n_i \log_2 \frac{4n}{n_i} \le \sum_{i \ge 1} \frac{n}{2^{i-1}} \log_2 \frac{4n}{n/2^{i-1}} = n \cdot \sum_{i \ge 1} \frac{O(i)}{2^i} = O(n).$$

$$\begin{aligned}
S &= \sum_{i \ge 0} \frac{i}{2^i} \\
2S &= \sum_{i \ge 0} \frac{i}{2^{i-1}} = \sum_{i \ge 0} \frac{i+1}{2^i} \\
\implies 2S - S &= \sum_{i \ge 0} \frac{(i+1) - i}{2^i} = \sum_{i \ge 0} \frac{1}{2^i} = 2.
\end{aligned}$$

The inequality above uses that $x \log(4n/x)$ is increasing for $x \le n$. □

Converting this into an algorithm that runs in $O(m+n)$ time requires quite a bit more work. The essential idea is to store each query string $Q_{(u,v)}$ as a bit vector of length $\log_2 n$, indicating which nodes on the path from v to the root belong to it $Q_{(u,v)}$. Now the answers $A_{(u,v)}$ can be stored by encoding the locations of the successive maxima. And answers for a child edge can be computed from that of the parent edge using some tricky bit operations (e.g., by precomputing solutions on bit-strings of length, say $(\log_2 n)/3$, of which there are only $n^{1/3} \times n^{1/3} = n^{2/3}$). If you are interested, check out these lecture slides by Uri Zwick.

1.5.2 *Solving the General Case*

Finally, we reduce a general instance of MST verification to the special instances considered in §1.5.2. First we reduce to a "branching" tree with the special properties we asked for, then we alter the queries to become leaf-ancestor queries.

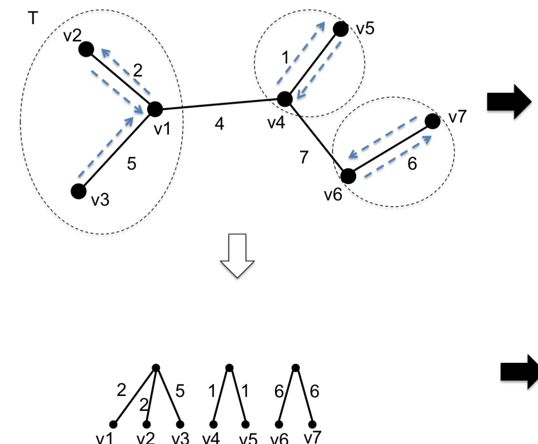

Figure 1.8: Illustration of balancing a tree. We have $\texttt{maxwt}_T(v_1, v_7)$ is 7 which is the weight of edge (v_4, v_6). We can

To achieve this reduction, run Borůvka's algorithm on the tree T. After the ith round of edge selection and contraction, let V_i be the remaining vertices, so that $V_0 = V$ is the original set of nodes. Define a new tree T' whose vertex set V' is the disjoint union $V_0 \uplus V_1 \uplus \cdots$. A node $u \in V_i$ has an edge in T' to $v \in V_{i+1}$ if the component containing u was contracted into the new vertex v; the weight of this edge in T' is the weight of the minimum-weight edge chosen by u in this round. Moreover, if r is the single vertex corresponding to the entire tree T at the end of the run of Borůvka's algorithm, then root tree T' at r.

Exercise 1.14. Show that each node in T' has at least two children, and all leaves belong to the same level. There are n leaves (corresponding to the nodes in T), and at most $2n - 1$ nodes in T'. Also show how to construct T' in linear time.

Exercise 1.15. For nodes u, v in a tree T, let $\mathtt{maxwt}_T(u, v)$ be the maximum weight of an edge on the (unique) path between u, v in the tree T. Show that all $u, v \in V$, $\mathtt{maxwt}_T(u, v) = \mathtt{maxwt}_{T'}(u, v)$.

This exercise means arbitrary queries (y_i, z_i) in the original tree T can be reduced to leaf-leaf queries in T'. To make these leaf-ancestor queries, we simply find the least-common ancestor $\ell_i := \text{lca}(y_i, z_i)$ for each pair, and replace the original query by the maximum of two queries $(y_i, \ell_i), (z_i, \ell_i)$. To show that we can find the least-common ancestors in linear time, we defer to a theorem of David Harel and Bob Tarjan:

Harel and Tarjan (1984)

Theorem 1.16. *Given a tree T, we can preprocess it in $O(n)$ time, so that all subsequent least-common ancestor queries for T can be answered in $O(1)$ time.*

Interestingly, this algorithm also proceeds by solving the least-common ancestor problem for complete balanced binary trees, and then extending the solution to general trees. For a survey of algorithms for this problem, see the paper of Alstrup et al.

Alstrup et al. (2004)

This completes Komlós' proof that the MST verification problem can be solved using $O(m + n)$ comparisons. An outstanding open problem is to get a really simple linear-time algorithm for this problem. (An algorithm that runs in time $O(m\alpha(n))$ can be given using the disjoint set union-find data structure.)

1.6 The Ackermann Function

Wilhelm Ackermann defined a fast-growing function that is totally computable but not primitive recursive. Today, we use the term

Ackermann (1928)

A similar function was defined by Gabriel Sudan, a Romanian mathematician, in 1927.

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 illustrative purposes, it is 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	4	2^{2^2}	$2^{2^{2^2}}$	...	$2^{2^{\cdot^{\cdot^{2}}}}$
4	2	4	65536	!!!	...	huge!

We can define the ***inverse Ackermann function*** $\alpha(\cdot)$ 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 $m \leq 2^{2^{\cdot^{\cdot^{2}}}}$ where the tower has height 65536.

1.7 Matroids

To come here. See HW1 as well.