15-850: Advanced Algorithms **CMU, Spring 2017**
Lecture #1: Deterministic MSTs January 25, 2017
Lecturer: Anupam Gupta Scribe: Anshu Bansal, C.J. Argue

1 Minimum Spanning Trees: History

The minimum spanning tree problem is classic: given a weighted 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's algorithm was the first MST algorithm from 1926. [1] Kruskal [Kru56] gave his algorithm in '56. Jarnik [Jar30] gave his algorithm in 1930, and it was rediscovered by Prim ('57) and Dijkstra ('59). 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. In Lecture #2, we'll go over 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.*

[1] Borvka, Otakar (1926). "O jistm problmu minimlnm" [About a certain minimal problem]. Prce mor. prodovd. spol. v Brn III (in Czech and German). 3: 3758.

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, and let T be any spanning tree that contains e. Since T contains no cycle, there is some $e' \in C \backslash T$, and by choice of e we have $w(e') < w(e)$. Then $T' = (T - \{e'\}) \cup \{e\}$ 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. □

The algorithms we'll discuss today only use the cut rule.

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.

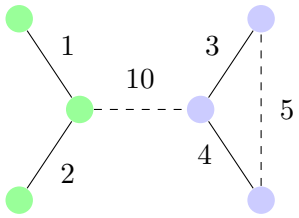

Figure 1.1: Dashed lines are not yet in the MST. Note that 5 will be analyzed next, but will not be added. 10 will be added. Colors designate connected components.

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.

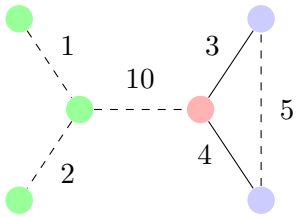

Figure 1.2: Dashed lines are not yet in the MST. We started at the red node, and the blue nodes are also part of T right now.

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 $elem$ 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`$(v, w(uv))$.[2]

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 `insert`s, and n `extractmin`s, with the `decreasekey`s 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.

[2] 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.

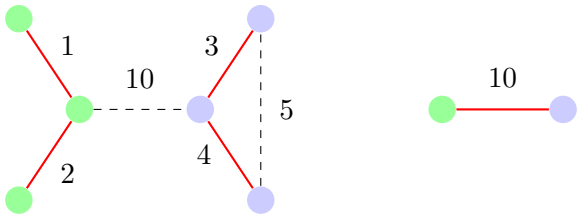

Figure 1.3: The red edges will be chosen and contracted in a single step, yielding the graph on the right, which we recurse on. Colors designate components.

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, last page): 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.

References

[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

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

[KKT95] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321, 1995. 1

[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. Pettite 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

The Ackermann Function $A(m,n)$ is a very rapidly growing function Formally, $A(m,n) : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ is defined recursively

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

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!

The α function is the inverse of $A(n,n)$, which 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.

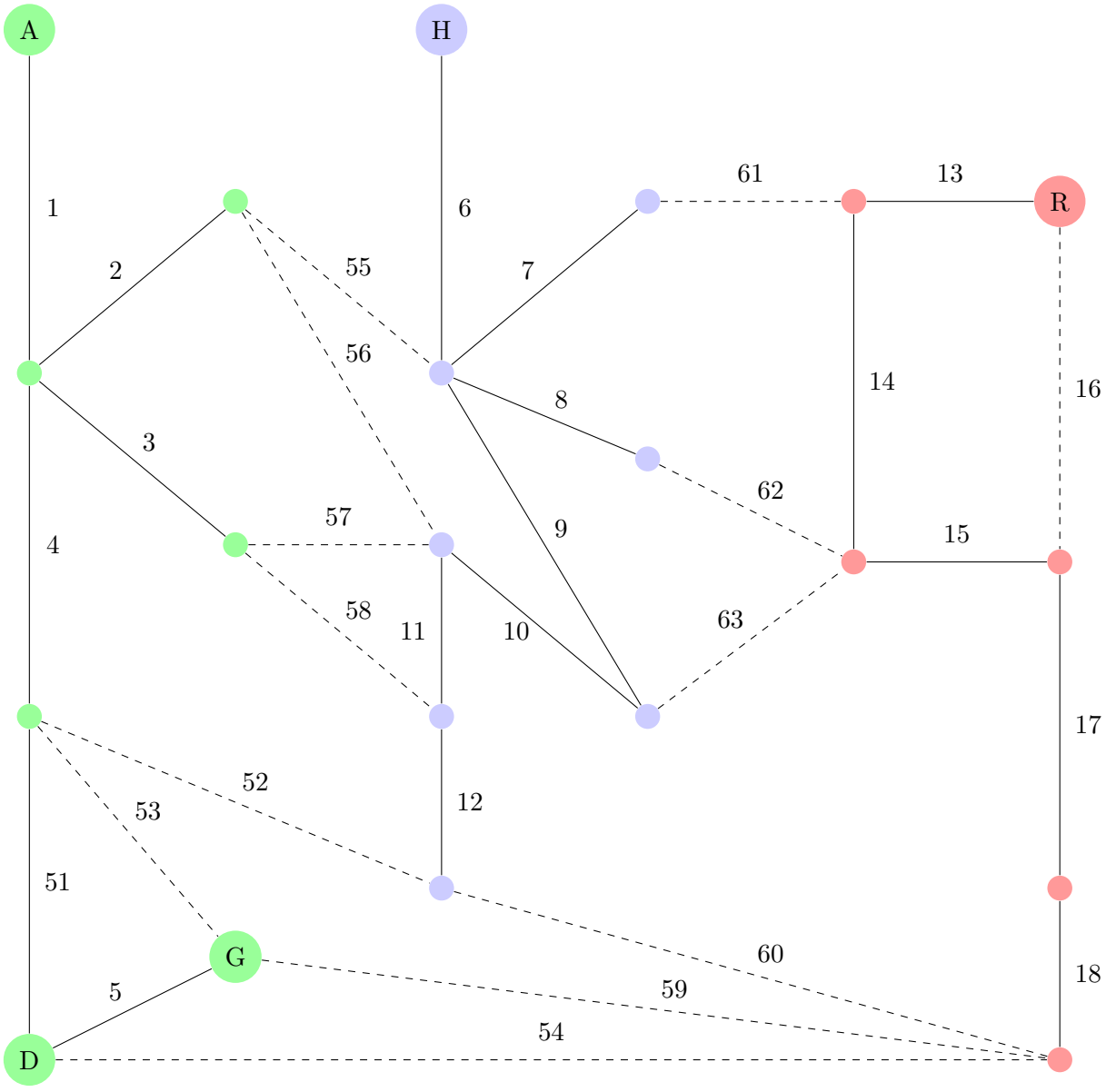

Figure 1.4: We set $K = 6$. We begin Prim's algorithm at vertices A, H, R, and D (in that order). Note that although D begins as its own component, it stops when it joins with tree A. Dashed edges are not chosen in this step (though may be chosen in the next recursive call), colors denote trees.