Chapter 18

Minimum Spanning Trees I: A Fast Deterministic Algorithm

We now begin the next part of the book wherein we study graph algorithms, starting with the problem of finding a *minimum spanning tree (MST)*.

18.1 Minimum Spanning Trees

Minimum spanning trees (MSTs) are a standard topic in most undergraduate algorithms textbooks. Our goal in this and the next chapter is to go beyond these standard treatments of this topic and cover more advanced—albeit classical at this point—algorithms for MSTs. But let us start with a quick review of basics of MSTs first.

Let $G = (V, E)$ be an undirected *connected* graph with *positive* weights $w : E \to \mathbb{R}^+$ on its edges. A **spanning tree** of a connected graph G is any spanning subgraph of G which is a tree (a tree itself is a connected subgraph with no cycles). With a slight abuse of notation, for any tree T, we write $w(T) := \sum_{e \in T} w(e)$ to denote the sum of the weights of edges in T.

Problem 18.1. In the **minimum spanning tree (MST)** problem, we are given an undirected graph $G = (V, E)$ and positive edge-weights $w : E \to \mathbb{R}^+$; the goal is to find a spanning tree T of G with minimum weight $w(T)$, namely, an MST of G.

Throughout this book, we will be making the following assumption.

Assumption 18.2. We can assume without loss of generality that the edge weights $w(e)$ for $e \in E$ are *distinct*. This can be achieved for instance by using a consistent tie-breaking rule based on the IDs of the edges.

A standard consequence of Assumption 18.2 is the *uniqueness* of the MST.

Proposition 18.3. *In any weighted graph $G = (V, E, w)$ satisfying Assumption 18.2, the MST of G is unique.*

Proof. Proof by contradiction: suppose there are two different MSTs T_1 and T_2 for G. Let e_1 be any edge in T_1 not in T_2. Suppose we add e_1 to T_2. This will create a cycle C in $T_2 \cup \{e_1\}$. We consider two cases:

- *Case 1:* There is at least one edge $e_2 \neq e_1$ in C with $w(e_2) > w(e_1)$: Remove e_2 from T_2 and instead add e_1. Let $T := T_2 \cup \{e_1\} \setminus \{e_2\}$. This is also a spanning tree of G but now $w(T) < w(T_2)$ as we removed a "heavy" edge and brought in a "lighter" one. This is a contradiction with T_2 being an MST so this case cannot happen.
- *Case 2:* Every edge in the cycle C has weight lower than e_1. Remove e_1 from T_1 to create two connected components S and $V \setminus S$. Because C was a cycle with e_1, there is still a path P in C from the endpoints of e_1 without using e_1 and P has at least one edge, say, f, between S and $V \setminus S$. Let $T := T_1 \setminus \{e_1\} \cup \{f\}$. This is also a spanning tree of G but now $w(T) < w(T_1)$, a contradiction with T_1 being an MST, so this cannot happen either.

Given the edge-weights are distinct, the only possible cases are the above and we showed neither can happen, hence proving the statement by contradiction. □

Consequently, throughout the book, unless explicitly stated otherwise, we assume the edge-weights are distinct and thus, by Proposition 18.3, **MST of the input graph is unique**. In particular, we can refer to it as *the* MST of G (instead of a/some MST of G) which makes the analysis (mostly its exposition) easier.

18.1.1 The Basic Rules of MSTs

There are two general *rules* that are used quite frequently in the design of essentially every MST algorithm. You have most likely seen these rules before but perhaps not in this exact formulation. You may want to refer to Figure 18.1 for a simple illustration of these rules.

Cut Rule: *In any graph $G = (V, E)$, the minimum weight edge e in* **any cut** *S always belongs to the MST of G. This rule determines which edges should be in the MST.*

Proof. Suppose, by contradiction, that $S \subseteq V$ is a cut in G such that its minimum weight edge e is not part of the MST T. Add e to T which creates a cycle C. Cycle C should have another edge f crossing the cut S. Consider $T' := T \cup \{e\} \setminus \{f\}$, which is still a spanning tree of G but has $w(T') < w(T)$ as $w(e) < w(f)$ since e is the minimum weight edge of the cut S. This is a contradiction with T being the MST of G. □

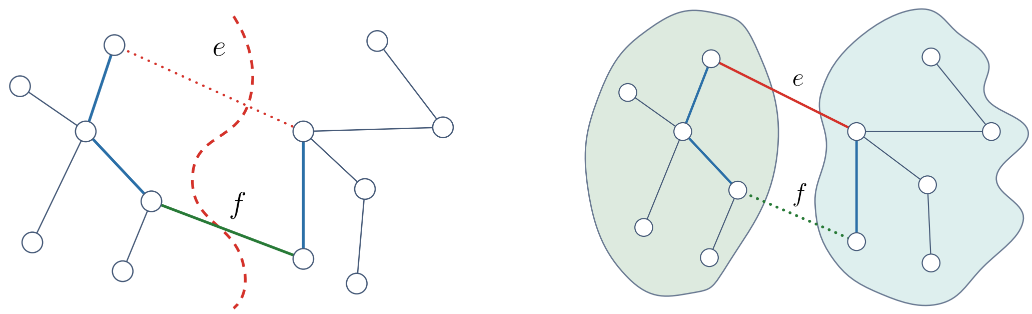
Cycle Rule: *In any graph $G = (V, E)$, the maximum weight edge e in* **any cycle** *C never belongs to the MST of G. This rule determines which edges should not be in the MST.*

Proof. Suppose, by contradiction, that C is a cycle in G such that its maximum weight edge e is part of the MST T. Remove e from T to get two connected components, and find an edge $f \neq e$ in C that connects back these components. Consider $T' := T \setminus \{e\} \cup \{f\}$, which is still a spanning tree of G. We have $w(T') < w(T)$ as $w(f) < w(e)$ since e is the maximum weight edge of cycle C. This contradicts T being the MST. □

18.2 Refresher on the Classical Algorithms for MSTs

This section reviews classical algorithms for MSTs—*Kruskal's, Prim's,* and *Boruvka's* algorithms—that you most likely have seen previously (at least some subset of them[1]). A familiar reader can safely skip this section and continue to the next one.

[1]Sadly, Boruvka's algorithm seems to be missing from many standard textbooks on algorithms even though it is arguably simpler than both the other two alternatives.



(a) Here, e is the minimum weight edge of the cut shown by red dashed line and thus should replace f in the MST. The cycle C is the thick (blue, green) edges and the dashed line for e.

(b) Here, e is the max-weight edge of the cycle with thick (blue, green, red) edges. Removing e from the MST creates two connected components that should be connected by the edge f instead.

Figure 18.1: An illustration of the proofs of **cut rule** and **cycle rule**.

18.2.1 Kruskal's Algorithm

To state Kruskal's algorithm, we first need to review the *union-find* data structure.

Definition 18.4. A **union-find** is a data structure supporting these operations on input set $\{1, 2, \ldots, n\}$:

- `initialize`: starts by creating n *disjoint* sets $S_1, \ldots, S_n$ where $S_i := \{i\}$;
- `union`(i, j): Given index of two sets i, j, replace S_i and S_j with $S_i \cup S_j$ and the index of the new set with $\min\{i, j\}$;
- `find`(e): given an element $e \in [n]$, returns the index of the set S_j where $e \in S_j$.

There are different ways of implementing a union-find data structure and this problem was studied extensively in 60's, 70's, and 80's and at this point is extremely well-understood. We will not go over these implementations in this text. For our purpose, we can use the simplest (non-trivial) implementation of this data structure that implements `initialize` in $O(n)$ time and `union` and `find` in $O(\log n)$ time.

Kruskal's algorithm works by iterating over the edges in the order of their weights and includeing every edge that does not create a cycle; this is done by maintaining the connected components in a union-find data structure and only picking an edge if it connects two different components and merge those components.

Algorithm 18.5.

(i) Create a union-find data structure D on n vertices in V by running `initialize`. Let $T = \emptyset$. Sort the edges of G in increasing order of their weights.

(ii) For $i = 1$ to m in the sorted ordering of edges:

- (a) Let $e_i = (u_i, v_i)$. Let $a = D.\texttt{find}(u_i)$ and $b = D.\texttt{find}(v_i)$.
- (b) If $a = b$ continue to the next edge; otherwise, $T \leftarrow T \cup \{e_i\}$ and run $D.\texttt{union}(a, b)$.

(iii) Return T.

Proof of Correctness: The fact that Algorithm 18.5 returns a spanning tree is immediate: any edge is ignored only if it creates a cycle (as it is inside a single connected component) and thus non-ignored edges should maintain the connectivity of the input graph. To see that T is the MST of G we can apply the **cycle rule**: any edge e ignored by the algorithm has both its endpoints inside a single connected component and thus forms a cycle inside that component; given the sortedness of the edges, this ignored edge is the maximum-weight edge of this cycle, and thus by the **cycle rule** cannot be part of the MST. So, any ignored edge cannot be part of the MST and thus the non-ignored edges T should form the MST.

Runtime Analysis: The first line of the algorithm takes $O(m \log m) = O(m \log n)$ time (as $n - 1 \leqslant m \leqslant \binom{n}{2}$ and so $\log n = \Theta(\log m)$) to sort all the edges (using, say, merge sort). Initializing the union-find takes $O(n)$ time and each of its operations takes $O(\log n)$ time, so each iteration of the for-loop also takes $O(\log n)$ time, making the total runtime of the algorithm $O(m \log n)$. Finally, it is worth pointing out that we can implement for-loop *much faster* and almost (but not quietly) in linear time using more sophisticated implementations of the union-find data structure; however that will not reduce the runtime due to the sorting step.

18.2.2 Prim's Algorithm

We now consider another textbook algorithm for MSTs, the *Prim's Algorithm*. To do so, we need to recall the definition of *priority queue* data structures.

Definition 18.6. A **priority queue** is a data structure Q supporting the following operations on a universe $\mathcal{U}$ of elements:

- `insert`(e, v): Given an element $e \in \mathcal{U}$ and a value $v \in \mathbb{R}$, inserts e with value v in Q;
- `decrease-key`(e, v): Given an element $e \in \mathcal{U}$ and a value $v \in \mathbb{R}$, changes the value of e to *minimum* of v and its old value;
- `extract-min`: Removes the element e with the smallest value v from Q and returns (e, v).

The Kruskal's algorithm is about maintaining *multiple* connected component of the forest F simultaneously and updating them (merge them together in the algorithm). On the other hand, Prim's algorithm only contains *one non-trivial* connected component (all other components are singleton vertices) which we "grow" until it contains the entire graph. The strategy for growing the component is to pick the edge with minimum weight that goes out of this component.

Algorithm 18.7.

(i) Pick an *arbitrary* vertex s of the graph. Let T be a subgraph of G consisting of only the vertex s for now and Q be a priority queue with keys being vertices in V and key-values being an edge $\mathit{edge}(v)$ and its weight. For every vertex $v \in N(s)$ set $\mathit{edge}(v) = (s, v)$ and run Q.`insert`$(v, w(\mathit{edge}(v)))$.

(ii) For $n - 1$ steps:

(a) Run Q.`extract-min` to get a vertex v with smallest value and let $\mathit{edge}(v)$ be the edge of v with weight equal to the value of v in Q. Add v and $\mathit{edge}(v)$ to T.

(b) For any vertex $u \in N(v)$, if $u \notin T$ update Q.`decrease-key`$(u, w(uv))$ (if u is not in Q, we instead run Q.`insert`$(u, w(uv))$).

(iii) Return T as the MST of G.

Proof of Correctness: There are two steps in proving the correctness of Prim's algorithm: (1) it outputs a spanning tree, and (2) the weight of this spanning tree is minimum, namely, it is an MST. The proof of first part is simply as follows: the graph T created by the algorithm consists of $n-1$ edges (given the $n-1$ steps of the for-loop), and does not have a cycle as every edge added is incident on a brand-new vertex v, which is a leaf-node of T (has degree 1) at the moment it is added and thus cannot close a cycle. Finally, in all $n-1$ steps a new edge will be added to T as otherwise the set of vertices included in T will have no outgoing edges, implying that G was not connected. Thus T is a spanning tree.

To prove T is an MST we apply the **cut rule**. Let e be an edge added to T by the algorithm. Let C be the connected component of T before adding the edge e. Since all vertices in C have been marked, all edges incident on them have been added to Q. This means that all edges between T and $N(T)$, the set of neighbors of T, have been included in Q (by way of running Q.`decrease-key`). Since e has the minimum weight inside Q, this means that it has the minimum weight among all edges between T and $N(T)$. Thus, e is the minimum weight edge of the cut T and hence, by the **cut rule** e belongs to the MST. This proves all $n-1$ edges inserted to T belong to the MST, hence T is the MST of the input graph.

Runtime Analysis: The algorithm requires $O(n)$ `insert` operations, $O(m)$ `decrease-key`, and $O(n)$ `extract-min` operations. Thus, if we implement the priority queue via a *min-heap* (a.k.a. a *binary heap*) that runs in each operation in $O(\log |Q|)$ time, we get that the total runtime of the algorithm is $O(m \log n)$.

18.2.3 Boruvka's Algorithm

Finally, we get to Boruvka's algorithm. It works by having each vertex be its own connected component and then proceeds in *rounds*. In each round, it picks the minimum weight edge going out of each component, adds them to the tree, and merges those components to create larger connected components.

Algorithm 18.8 (Boruvka's Algorithm).

(*i*) Start with a set $S(v) = \{v\}$ for each vertex $v \in V$ and an empty subgraph $T = \emptyset$.

(*ii*) In each **round**:

(a) For each set S, find the minimum weight edge e_S going out of S.

(b) Add every edge $e_S = (u, v)$ to the subgraph T and merge the sets of $S(u)$ and $S(v)$ into a single set S.

(*iii*) Repeat the rounds until all vertices are merged into a single set and return T.

Proof of Correctness: As before, we first argue that T is a tree. Given that the maintained sets in the algorithm are connected components of T, the subgraph T is connected. Any edge added by the algorithm in a round also connects two different connected components. The only concern is that what if the edges added in the same round form a cycle? But, this cannot happen because the weight of the picked edges in any path should be strictly decreasing (as in, if u picks an edge to v, but v picks an edge to w instead, then $w(u, v) > w(v, w)$) and thus the edges cannot form a cycle. As such, T is indeed a tree.

To argue T is the MST of G, we again apply the **cut rule**: any edge inserted by the algorithm is the minimum weight edge going out of its connected component (hence the cut) and thus by the **cut rule** belongs to the MST.

Runtime Analysis: Each round of the algorithm takes $O(m)$ time as it simply involves going over all edges of the graph twice (once from each endpoint). The number of rounds is also $O(\log n)$ as we argue next. In each round, each connected component gets merged with at least another one, and thus the number of components decreases by, at least, a factor of two. Hence, after at most $\log n$ rounds, only one component remain and the algorithm terminates. Hence, the algorithm takes $O(m \log n)$ time in total.

This concludes our summary of the background on classical algorithms for MST.

18.3 Fredman-Tarjan Algorithm in $O(m \log^*(n))$ Time

We now go over one of the earliest "advanced" algorithms for MSTs—named the *Fredman-Tarjan* algorithm—presented in [FT87] as part of—and one key application of—their introduction of *Fibonacci heaps*. The Fredman-Tarjan algorithm runs in $O(m \log^*(n))$ time, where $\log^*(n)$ is the number of times we need to take $\log(\cdot)$ until the number reaches one[2]. The algorithm is quite simple and even more so elegant and beautiful.

Let us start by defining Fibonacci heaps that implement priority queues (Definition 18.6) faster than a typical min-heap.

Proposition 18.9 ([FT87])**.** *There is a data structure H, called a* **Fibonacci heap***, that implements a priority queue (Definition 18.6) such that* `insert` *and* `decrease-key` *operations take $O(1)$ and* `extract-min` *operation takes $O(\log |H|)$* amortized *time where $|H|$ denotes the maximum number of elements in the heap during the entire execution.*

The construction of Fibonacci heaps is not particularly simple and we will not go over that in this book. Instead, we show how to use this data structure to get faster MST algorithms.

18.3.1 Warm-Up: Improving Prim's Runtime via Fibonacci Heaps

Using Fibonacci heaps in Prim's algorithm in place of its min-heap immediately leads to improving its runtime. As stated, Prim's algorithm involves $O(n)$ `insert` operations, one for each vertex of the graph, $O(m)$ `decrease-key` operations, one each time we visit an edge of the graph, and $O(n)$ `extract-min` operation, one each time we mark a new vertex. Thus, using Fibonacci heaps, by Proposition 18.9, the runtime of Prim's algorithm actually reduces to $O(m + n \log n)$ time.

As long as the input graph is even slightly dense, i.e., $m = \Omega(n \log n)$ edges, Prim's algorithm equipped with Fibonacci heaps runs in linear-time in the input size which is optimal. But, when m is $\Theta(n)$ itself, then this algorithm does not provide any improvement over the runtime of the original Prim's algorithm.

18.3.2 The Fredman-Tarjan Algorithm

We next go over Fredman-Tarjan algorithm that improves this runtime for all values of m and prove the following theorem.

Theorem 18.10 ([FT87])**.** *There is a deterministic algorithm for the minimum spanning tree problem that runs in $O(m \log^*(n))$ time.*

The key observation behind the Fredman-Tarjan algorithm is that the runtime of Prim's algorithm with a Fibonacci heap depends on the size of the heap, which itself is equal to the

[2]This is an extremely slow growing function and even if we take n to be the estimated number of atoms in the entire universe, $\log^*(n)$ will be at most 6 still.

frontier of the graph search, namely, vertices waiting in the priority queue to be examined next. So, instead of growing the frontier to become "too much to handle", we can limit its size and instead run Prim's algorithm for another starting vertex and keep doing this – one then needs to recombine these partial trees recursively.

We now state the formal algorithm.

Algorithm 18.11.

- The algorithm runs in **rounds**. In round i, it works with a graph $G_i = (V_i, E_i)$ with n_i vertices and m_i edges which is obtained from G by *contracting* some vertices chosen in previous rounds; we take $G_1 = G$ at the beginning. In round i, it also uses a *threshold* t_i on the size of the Fibonacci heaps it uses which we set to be $t_i := 2^{2m/n_i}$ (note that the numerator in the exponent is m independent of the round but the denominator depends on the round).

- In each round i, the algorithm does as follows:

  1. Start with every vertex $v \in V_i$ being unmarked.
  2. Pick an arbitrary unmarked vertex and run Prim's algorithm from it to get a tree T_v. Use Fibonacci heaps on the neighborhood of T_v, defined as
  $$N(T_v) := \{u \in V_i \setminus T_v \mid \exists z \in T_v \text{ such that } u \in N(z)\};$$
  in each step, use `extract-min` on the heap to take the vertex with minimum edge-weight from $N(T_v)$ and expand T_v from that vertex (note that vertices in $N(T_v)$ can be marked already but we still include them in $N(T_v)$).
  3. If at the end of each step of Prim's algorithm in the previous line, $|N(T_v)| \geqslant t_i$, or that the next vertex to explore is already a marked vertex (i.e., T_v just got an edge to an already marked vertex), then terminate this branch of Prim's algorithm, mark all vertices in T_v, and go to Line (2). Continue as long as there is any unmarked vertex left.
  4. Let $C_1, C_2, C_3, \ldots$ be the connected components created by different T_v's picked in the above steps. Create G_{i+1} for the next round by contracting these components in G_i into a single vertex. Then go to the next round and continue this until the whole graph is contracted to a single vertex and we have a single spanning tree (obtained by combining the trees computed in different steps).

Proof of Correctness. The proof of correctness of the algorithm follows from the correctness of Prim's and Boruvka's algorithms. We first have that Algorithm 18.11 always finds a spanning tree of the input graph the same way Boruvka's algorithm always finds a spanning tree. We then have that each edge e added to the tree T is a minimum weight edge of some cut and thus we can apply the **cut rule** to argue this edge belongs to the MST. The cut itself is found by considering the connected component of G on the set of vertices in T (and possibly by un-contracting the vertices in rounds > 1), the same exact way as in Prim's algorithm.

Runtime analysis. We now switch to the runtime which is the main part of the analysis for this new algorithm. Each round involves visiting each edge at most twice (from either side) and inserting/updating their weights into a Fibonacci heap; as well as removing each vertex from the heap at most once via the `extract-min` operation (after that, the vertex is marked one way or another and will not be inserted to the heap again). As size of each heap is capped at t_i

by the construction of the algorithm[3], the total runtime of the round i, by Proposition 18.9, is $O(m_i + n_i \cdot \log t_i)$ time. Given the choice of $t_i := 2^{2m/n_i}$, this total runtime is $O(m_i + m)$ time. Finally, as $m_i \leqslant m$, we have that Algorithm 18.11 takes $O(m)$ time in each round.

Consequently, to bound the total time of the algorithm, we need to bound the number of rounds before the algorithm contract the entire graph into a single component. This is done in the following lemma, that shows the thresholds are growing exponentially in each round.

Lemma 18.12. *For every round i of Algorithm 18.11, we have $t_{i+1} \geqslant 2^{t_i}$.*

Proof. We are going to bound n_{i+1} as a function of t_i and use the fact that $t_{i+1} = 2^{2m/n_{i+1}}$ to conclude the proof. By definition, n_{i+1} is equal to the number of connected components found at the end of the round. While we are truncating Prim's algorithm to make its heap small, we still do not expect to have "very small" components using the threshold we are using.

In particular, we claim that, for every component C in round i:

$$\sum_{v \in C} \deg_{G_i}(v) \geqslant t_i; \tag{18.1}$$

namely, sum of the degrees of vertices (in G_i) in a component C is at least equal the threshold. The proof of Eq (18.1) is as follows. For each sub-tree T created in the algorithm in round i and each vertex v extracted from the heap (i.e., marked) during its construction, we put at most $\deg_{G_i}(v)$ vertices in the heap corresponding to T (we may insert less if those vertices are already inserted). This means that size of the heap can only increase to sum of the degrees of extracted/marked vertices. Thus, if $|N(T)| \geqslant t_i$, we know that all vertices in T will be marked and so collectively they have contributed at least t_i to the sum of the degrees, hence proving Eq (18.1). On the other hand, if we terminate the search because we connected to another marked vertex, then, by the prior argument (or rather inductively) sum of the degrees in that component was already t_i and we are only adding these vertices to that component which can only increase the total degrees. This proves Eq (18.1).

Now, let $C_1, \ldots, C_{n_{i+1}}$ be the connected components found in this step (recall that n_{i+1} is equal to the number of these connected components). We have,

$$\begin{aligned}
2m_i &= \sum_{v \in V_i} \deg_{G_i}(v) && \text{(by the handshaking lemma)} \\
&= \sum_{j=1}^{n_{i+1}} \sum_{v \in C_j} \deg_{G_i}(v) \\
& && \text{(by partitioning } V_i \text{ into } C_1, \ldots, C_{n_{i+1}} \text{ as the algorithm continues until it marks all vertices)} \\
&\geqslant n_{i+1} \cdot t_i. && \text{(by Eq (18.1))}
\end{aligned}$$

Thus, we have that

$$\log t_{i+1} = \frac{2m}{n_{i+1}} \geqslant \frac{2m_i}{n_{i+1}} \geqslant t_i,$$

and hence $t_{i+1} \geqslant 2^{t_i}$, concluding the proof. □

By Lemma 18.12, in round i of the algorithm, the value of t_i is at least a *tower* of twos of length i (i.e., two to the power two to the power two ... for i times). This means that

[3]Technically speaking, size of the heap can grow a bit larger than this, by the degree of the final vertex added. But at that point, we are no longer running any `extract-min` operation on the heap. So, for every `extract-min` operation, size of the heap is indeed bounded by t_i.

after $O(\log^*(n))$ rounds, the value of t_i will be at least n and in that round, we are only running Prim's algorithm once and so the algorithm terminates surely in that round. Thus, we conclude Algorithm 18.11 takes $O(m \log^*(n))$ time.

This concludes the proof of Theorem 18.10.

Exercises

Exercise 18.1. Design a deterministic $O(m)$ time algorithm for finding MST of a given planar graph. A planar graph is a graph that can be drawn on a surface so that no two edges cross each other.

Hint: All you need to know about planar graphs for this problem are: (1) how many edges are in a planar graph, and, (2) what happens to a planar graph after contracting its edges.