

17 Minimum Spanning Trees

In this chapter we will define spanning trees and look at the two most well known algorithms for computing the minimum-weight spanning tree in a weighted graph.

17.1 Spanning Trees

Recall that we define a tree as a graph which is connected and acyclic with $m = n - 1$. We proved in Theorem 65 that having any two of these three properties implies the third, so to show that a graph is a tree we can show that it has whichever two of these properties are most convenient to prove.

Recall also that a graph G' is a subgraph of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$, i.e. G' can be constructed from G by selectively removing (possibly 0) vertices and/or edges. We say that G' is a **spanning** subgraph of G if G' is a subgraph with $V(G') = V(G)$, i.e. G' is created with only edge deletions. A spanning tree is a spanning subgraph which is a tree.

Theorem 76. *A graph G has a spanning tree if and only if G is connected.*

Proof. First we prove that if G is connected then it has a spanning tree. We can find a spanning tree of a connected graph in $O(m + n)$ time by running BFS (or DFS) from any arbitrary vertex and adding to our tree the edges that we use to visit new vertices. Since G is connected, BFS will visit all vertices, and thus the shortest paths tree will be a spanning tree.

Now let G be a graph with a spanning tree. We need to show that G is connected. Let T be a spanning tree for G . To show that G is connected we need to show that for every pair of vertices in G there is a path in G between those vertices. Let $u, v \in V(G)$ be arbitrary. Because T is a spanning subgraph of G , $V(T) = V(G)$, so u and v are also vertices of T . Because T is a tree, T is connected, so there exists a $u \rightarrow v$ path P in T . Because T is a subgraph of G , all of the vertices and edges of P are also present in G , so P is a $u \rightarrow v$ path in G , and thus u and v are connected. Since u and v were arbitrary vertices, this means that G is connected. $\square$

Note that unlike the shortest path trees we've discussed over the previous two chapters, a spanning tree is not a rooted tree (we could arbitrarily pick a vertex to treat as the root, but there is no distinguished choice for this vertex). Unlike our SSSP algorithms, an MST algorithm does not take in a source vertex s . Also, while shortest path algorithms are most commonly defined for directed graphs (since shortest paths in undirected graphs can be seen as a special case of shortest paths in directed graphs), spanning trees are defined only for undirected graphs.

A graph can have many spanning trees, exponentially many even if it is very dense. For example, a complete graph (one with an edge between every pair of vertices) on n vertices has n^{n-2} distinct spanning trees (this is known as Cayley's formula, and was first proven in 1860).

Out of the many possible spanning trees, we would often like to find the best one. To compare different spanning trees, we define the weight of a tree as the sum of the weights of its edges:

$$w(T) := \sum_{e \in E(T)} w(e).$$

We define a **minimum spanning tree** for G as a spanning tree with minimum weight, among all spanning trees for G .

Throughout this chapter we will focus on weighted graphs with *distinct* edge weights. The assumption that all edge weights are distinct makes several of our proofs and algorithms shorter and simpler. While this assumption may seem unrealistic, in actuality it is generally easy to convert an MST algorithm that works only for graphs with distinct edge weights into one that works with arbitrary integer weights. We can simply choose tiny perturbation values $0 < \epsilon_1 < \epsilon_2 < \dots < \epsilon_{m-1} < \epsilon_m$ such that $\sum_{i=1}^m \epsilon_i < 1$ and then add ϵ_i to the weight of the i 'th edge. For example, if our graph had edge weights of 5, 3, 5, 7, 7, 7 and 12, we might change these to 5.01, 3.02, 5.03, 7.04, 7.05, 7.06, and 12.07.

These ϵ_i values give us a “tiebreaker” between identical edges, ensuring that all edge-weights in the graph are distinct, but since the total weight of all the ϵ_i 's is less than 1 the weight of the MST increases by less than 1, and thus the extra weight cannot cause any tree that wasn't tied for the lightest in the original graph to become an MST in the new graph.

17.2 Properties of MSTs

We begin our study of minimum spanning trees by proving some properties that will be helpful in designing and analyzing MST algorithms. The proofs in this section also illustrate some general techniques that are useful for proofs involving spanning trees.

Often in these proofs we want to discuss a graph created by adding or removing a single edge from another graph. To do this more easily we define the following notation:

$$G + e := (V(G), E(G) \cup \{e\}), \quad G - e := (V(G), E(G) \setminus \{e\}).$$

That is, $G + e$ is the graph created by adding edge e to G and $G - e$ is the graph created by removing edge e from G .

We begin by proving that our assumption of distinct edge weights in the graph guarantees that we can only have one minimum spanning tree. This means that for the rest of the chapter we can talk about “the” minimum spanning tree in a graph, rather than needing to consider the possibility that multiple trees tie for the minimum weight.

Theorem 77. *If G is a connected weighted graph with distinct edge weights, then G has a unique minimum spanning tree.*

Proof. Assume for sake of contradiction that there exist two distinct minimum spanning trees T_1 and T_2 for G . Since T_1 and T_2 are both spanning trees, $V(T_1) = V(T_2) = V(G)$ and $|E(T_1)| = |E(T_2)| = n - 1$. Thus, each tree must have at least one edge the other does not. Let e be the lightest (minimum weight) edge which is in one tree but not the other. Without loss of generality, assume $e \in E(T_1)$.

Consider the graph G' obtained by adding e to T_2 . Since T_2 is connected, T_2 already contains a path between e 's endpoints, so adding e to T_2 creates a cycle. Call this cycle C . Since T_1 is acyclic, but contains e , there must be some other edge of C which is in T_2 but not T_1 . Let f be such an edge. Since all edge weights in G are distinct and we chose e as the lightest edge which is in one tree but not the other, $w(e) < w(f)$.

Now consider the graph $T' := T_2 - f + e$. We note that T' has $n - 1$ edges (since we added and removed one edge from a tree). Let $v_1, v_2, \dots, v_k$ be the vertices of cycle C in order with $e = (v_1, v_k)$ and $f = (v_i, v_{i+1})$ (for some $1 < i < k$). We note that removing f from T_2 splits T_2 into two connected components, one containing $v_1, \dots, v_i$ (among other vertices) and the other containing $v_{i+1}, \dots, v_k$ (among other vertices). Adding e to $T_2 - f$ reconnects these two components, so T' is a connected graph with $n - 1$ edges on vertex set V and thus is a spanning tree for G . Furthermore $w(T') = w(T_2) - w(f) + w(e) < w(T_2)$ (since $w(e) < w(f)$). Thus, there is a spanning tree with smaller weight than T_2 , contradicting our assumption that T_2 was a minimum spanning tree. Therefore, G has only one MST. $\square$

A **cut** in a graph $G = (V, E)$ is a partition of the vertex set V into two non-empty sets $S, V \setminus S$. We say that an edge e **crosses** the cut, if e has one endpoint in S and one endpoint in $V \setminus S$.

The following theorem is known as the “light edge property”, and is the core result behind the MST algorithm we will see in the next section. The proof uses the same “swapping edges in a cycle” technique that we introduced in the proof of Theorem 77.

Theorem 78. *Let G be a connected weighted graph with distinct edge weights and $S, V \setminus S$ be a cut in G . Let e be the minimum weight edge crossing this cut. Then the minimum spanning tree for G contains e .*

Proof. As in the previous proof, we will use contradiction. Let T be a minimum spanning tree for G and suppose for contradiction that $e \notin T$. Then adding e to T creates a cycle C . Call the vertices of C $v_1, v_2, \dots, v_k$, where v_1 is the endpoint of e in S and v_k is the endpoint of e in $V \setminus S$. Then, there must be at least one place where the path $v_1, v_2, \dots, v_k$ transitions from S to $V \setminus S$ (since it starts in S and ends in $V \setminus S$). Let $1 \leq i < k$ be an integer such that $v_i \in S$ and $v_{i+1} \notin S$. Call the edge $\{v_i, v_{i+1}\}$ f . Then f crosses the cut $S, V \setminus S$. Since e is the lightest edge crossing this cut, $w(f) > w(e)$.

Consider the graph $T' := T - f + e$. Note that T' still has $n - 1$ edges. Furthermore, T' is still connected, by the same argument we used in the proof of Theorem 77 (removing f splits T into two components, one of which contains $v_1, \dots, v_i$ and the other of which contains $v_{i+1}, \dots, v_k$. Adding e reconnects these components). Thus, T' is a spanning tree for G with $w(T') = w(T) - w(f) + w(e) < w(T)$, contradicting our assumption that T was a minimum spanning tree. $\square$

17.3 The Prim-Jarník Algorithm

The light edge property implies that we can construct an MST by repeatedly taking cuts in G and choosing the lightest edge crossing them to be part of our tree. This is what the first algorithm we will consider does.

The algorithm was first described by Vojtěch Jarník in 1929, although it is most commonly known today as “Prim’s algorithm” after Robert Prim, who independently rediscovered it in 1957. As a compromise, we will refer to it as the Prim-Jarník algorithm.

```

PRIM-JARNÍK( $G$ )
 $v \leftarrow$  an arbitrary vertex in  $V$ 
 $T \leftarrow (\{v\}, \emptyset)$ 
While  $V(T) \neq V(G)$ :
     $e \leftarrow$  the lightest edge crossing the cut  $V(T), V(G) \setminus V(T)$ 
    Add  $e$  to  $E(T)$ 
    Add  $e$ ’s endpoint from  $V(G) \setminus V(T)$  to  $V(T)$ 
Return  $T$ 

```

Theorem 79. *Let G be a connected weighted graph with distinct rational edge weights. Then PRIM-JARNÍK(G) computes the minimum spanning tree of G .*

Proof. We note that every edge we add to T is the lightest edge crossing some cut in G . Thus, by Theorem 78, every edge we add to T is part of the MST. Since every edge we add includes exactly one vertex which isn’t part of our current tree, we know that each iteration of the while loop finds a new edge from the MST to add. Furthermore since we start with one vertex in T and each iteration adds only one vertex, the loop runs for $n - 1$ iterations, so by the time we return T it contains all $n - 1$ of the edges from the MST and thus T is the MST for G . $\square$

Theorem 80. PRIM-JARNÍK can be implemented to run in time $O(m + n \log n)$.

Proof. As we showed in the proof of Theorem 79, the while loop in PRIM-JARNÍK runs $n - 1$ times. The work outside of the loop can be done in constant time. The tricky part is analyzing the work inside of the loop, in particular, how long does it take to find the lightest edge crossing $V(T), V(G) \setminus V(T)$? This depends on how we represent T .

One simple approach would be to use a length n boolean array to keep track of whether or not each vertex is in T , and a list of the edges currently in T . We can find the lightest edge across $V(T), V(G) \setminus V(T)$ by iterating through all the edges of G , and keep track of the lightest one among those which have one endpoint true in our boolean array and the other false. This takes $O(m)$ time. Thus, each iteration of the loop takes $O(m)$ time and so the total runtime of the algorithm with this approach is $O(mn)$.

We can do better, however. Instead of iterating through all of the edges every iteration of the while loop, we can use a priority queue to store the edge between $V(T)$ and $V(G) \setminus V(T)$. To get the lightest edge from $V(T)$ to $V(G) \setminus V(T)$, we call EXTRACTMIN on our priority queue until we get an edge crossing $V(T), V(G) \setminus V(T)$. When we add a new vertex to T , we add its edges to vertices in $V(G) \setminus V(T)$ to the priority queue. Each edge is inserted into the priority queue at most once and extracted at most once, so we make $O(m)$ calls to EXTRACTMIN, each of which takes $O(\log n)$ time (whether we use a binary or Fibonacci heap). Thus, the total runtime of our algorithm becomes $O(m \log n)$.

We can do even better with a slightly different perspective. To avoid needing $O(m)$ extractions from our priority queue, instead of storing edges in our queue we can store *vertices*. We let the

priority queue store each vertex in $V(G) \setminus V(T)$ and the key for a vertex v is the weight of the lightest edge from $V(T)$ to v . We can use EXTRACTMIN on this priority queue to get the next vertex v we should add to T . Then we can find the lightest edge e from v to the current vertices of T , add v and e to T , and then we call DECREASEKEY to update the keys of v 's neighbors based on their edges from v . With this approach we make only $O(n)$ total calls to EXTRACTMIN along with $O(m)$ calls to DECREASEKEY. Thus, with a Fibonacci heap, the total runtime of the algorithm becomes $O(m + n \log n)$. □

17.4 Kruskal's Algorithm

The other algorithm that we will present for computing an MST is also a very natural greedy algorithm. We start with a subgraph consisting of the empty graph on $V(G)$ and then repeatedly add to our subgraph the cheapest edge that connects two disconnected components of our subgraph until the whole subgraph is connected. This algorithm was developed by Joseph Kruskal in 1956.

```

KRUSKAL'S( $G$ )
   $A \leftarrow$  an array of  $G$ 's edges, sorted in order of increasing weight
   $T \leftarrow (V(G), \emptyset)$ 
  For  $i$  from 1 to  $m$ :
    If adding  $A[i]$  to  $T$  would not create a cycle:
      Add  $A[i]$  to  $T$ 
  Return  $T$ 

```

Theorem 81. *Let G be a connected weighted graph with distinct rational edge weights. Then KRUSKAL'S(G) computes the minimum spanning tree of G .*

Proof. Let $e = (u, v)$ be an edge that KRUSKAL'S adds to T . Let U be the component of T containing u at the start of the iteration of the for loop where e is added to T . We note adding any edge which crosses the cut $U, V \setminus U$ to T would connect two different connected components and therefore would not create a cycle. But since U is a connected component (meaning that it is a maximal set of connected vertices), none of these edges have been added to T yet. Since we are considering the edges in increasing order of weight (because A is sorted) and e is the first edge crossing this cut that we've considered, e must be the lightest edge crossing $U, V \setminus U$. Therefore by Theorem 78, every edge that KRUSKAL'S adds to T is part of the MST.

Furthermore, we note that KRUSKAL'S will add all edges of the MST to T , because an MST is acyclic, so each of these edges can be added without creating a cycle. Therefore, when the algorithm returns T , T is the MST for G . □

Theorem 82. *KRUSKAL'S can be implemented to run in time $O(m \log n)$.*

Proof. Sorting the edges by weight takes $O(m \log m)$ time (note that since $m \leq n^2$ this is $O(m \log n)$). The for loop runs m times. This time it's not as clear how to track whether adding an edge to T would create a cycle. T can have many components and we need some way to keep track of which is which.

We can keep an array labeling each vertex with a component number (initially each vertex has its own component). This allows us to check whether an edge would create a cycle in constant time (we just check whether its two endpoints have different component labels). When we add an edge, connecting two components, we relabel all of the vertices in one of the two merged components with the label of the other component. Unfortunately, since a component can have $\Omega(n)$ vertices, this can take $\Theta(n)$ time. Thus, with this approach our runtime is $\Theta(m \log n + n^2)$.

A cleverer approach is to have a "component" object consisting of a pair of integers where the first one is an identifier for the component and the second one is the size of the component. Each vertex maintains a pointer to its current component (initially we have n components, with identifiers ranging from 1 to n , each with a size of 1).

To check if two vertices are in the same component, we simply check if the identifiers for their components are the same. Since our components now keep track of their size, we can make merging more efficient. In particular, when we merge two components, we can change the pointers for all of the vertices from the *smaller* of the two merged components to point to the component object for the *larger* of the two components, and update this component's size to include the added vertices.

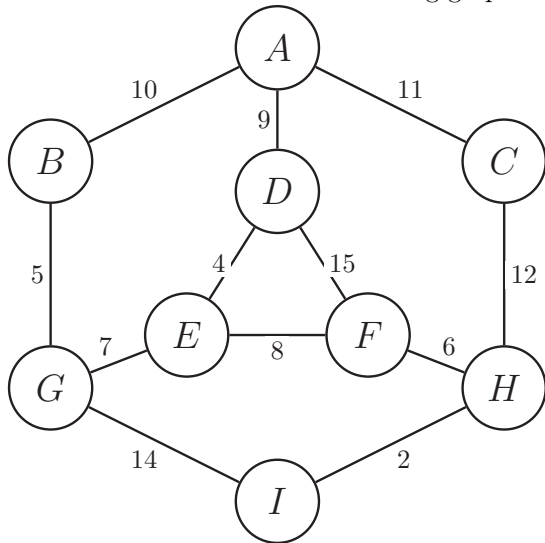
With this approach, whenever we need to update the component for a given vertex v , v 's new component must be at least twice as big as its old component (since updating v means that v 's component was the smaller one in the merge). We can double the size of v 's component only $O(\log n)$ times before it includes all n vertices. Thus, with this approach, the total runtime for the for loop becomes $O(m + n \log n)$.

Since $m \in \Omega(n)$ (because G is connected), $m \log n \in \Omega(m + n \log n)$, so our runtime is dominated by the time to sort the edges and our overall runtime is $O(m \log n)$. □

If our edge list is already sorted or if it can be sorted in linear time (e.g. because the edge weights are all relatively small integers and can be sorted with radix sort) then KRUSKAL'S with the implementation we just discussed runs in $O(m + n \log n)$ time the same as PRIM-JARNÍK. Kruskal's algorithm can actually be made to run even faster, by using a specialized "union-find" data structure, but the details are beyond the scope of this course.

17.5 Comprehension Questions

Questions 1-3 are based on the following graph:



1. Suppose that we are using PRIM-JARNÍK to find an MST in the graph and so far the only edge we have added to our subgraph is H, I. Which edge will we add next?
2. Suppose that we are using KRUSKAL'S to find an MST in the graph and so far the only edge we have added to our subgraph is H, I. Which edge will we add next?
3. Highlight all the edges in the MST for this graph.
4. True or false? Every connected graph with integer edge weights has a unique minimum spanning tree.
5. What will the runtime of KRUSKAL'S be if instead of using an array of component numbers/pointers, we check whether adding an edge (u, v) would create a cycle by running DEPTH-FIRSTSEARCH(T, u) and seeing if we visit v ?

6. What would the runtime of PRIM-JARNÍK be if instead of using a Fibonacci heap, we used a binary heap (where EXTRACTMIN and DECREASEKEY both take $\Theta(\log n)$ time)?

17.6 Exercises

- In this chapter we proved that if all of the edges in a graph have distinct rational weights then PRIM-JARNÍK(G) and KRUSKAL'S(G) find minimum spanning trees. In this problem we show that this assumption was not necessary.
 - Prove that if G is a graph with rational edge weights then PRIM-JARNÍK(G) finds one of the minimum spanning trees of G .
 - Prove that if G is a graph with rational edge weights then KRUSKAL'S(G) finds one of the minimum spanning trees of G .
- Suppose that you are given a weighted graph (G, w) and a minimum spanning tree T for G , along with an edge $e \in E(G)$ and a new weight x for e . Give an $O(m)$ algorithm for finding a new MST for G after $w(e)$ is updated to x . [Hint: You may want to case on whether or not $e \in T$ and/or whether $w(e)$ was increased or decreased]
- Let (G, w) be a (undirected) connected graph where every edge has a distinct edge weight.
 - Prove that for every cycle in G , the *maximum* weight edge in the cycle is *not* part of the minimum spanning tree.
 - Show that the following algorithm correctly finds an MST in G . Analyze the algorithm's runtime.

```

CYCLEREMOVALMST( $G, w$ )
   $C \leftarrow$  a cycle in  $G$  (or Null if  $G$  is acyclic)
  While  $C \neq \text{Null}$ :
     $e \leftarrow$  the maximum weight edge in  $C$ 
     $G \leftarrow G \setminus e$ 
     $C \leftarrow$  a cycle in  $G$  (or Null if  $G$  is acyclic)
  Return  $G$ 

```

- You are visiting a magnificent garden, consisting of a collection of flower beds connected by winding trails. You have acquired a map of the garden, listing all of the flower beds and all of the trails between them (each trail connects exactly two distinct flower beds). Your map also lists, for each trail, the number of minutes it takes to walk along it while fully appreciating the natural surroundings (you may assume that this time does not depend on which direction you are walking on the trail, nor on whether or not you have already walked this same trail).

In order to maximize your enjoyment, you want to visit all of the flowerbeds in the garden, while fully appreciating the natural surroundings on every trail you take (you may start and end at whichever flower beds you wish). We define a *stroll* as a walk through the garden which does this. Since there may be many different strolls, you're interested in finding the fastest one. This turns out to be a difficult problem, however (we'll see why later in the course). You aren't in too much of a hurry, so you're willing to settle for a stroll whose length is at most twice that of the best possible stroll. Give a polynomial-time algorithm which finds such a stroll. [Hint: Use a minimum spanning tree]
- Suppose that you are given a weighted graph (G, w) where all of the edges have integer weights between 1 and 411 (inclusive). Give an algorithm to find a minimum spanning tree for G in $O(m + n)$ time.