Graph Theory, Part II: Graph Algorithms

Clint Givens

March 21, 2016

Abstract

What's the cheapest way to build a complete electrical grid connecting a large number of cities? What's the quickest way to drive from Seattle to Miami? To answer these types of questions, it helps to know a bit about graph algorithms. An algorithm is just a reliable process used to compute some desired information from input data–for instance, there is an algorithm which takes a pair of positive integers as input, and computes their greatest common divisor as the output. A graph algorithm is, quite naturally, one which takes a graph as the input data.

In fact, we've already encountered our first graph algorithm in "Introduction to Graph Theory" – it was a process for finding Eulerian paths or cycles in a graph. In this session, we'll be looking especially at weighted graphs, which have numbers (weights) attached to each edge, representing things like distance, cost, or travel time. We'll discover two algorithms – Kruskal's and Prim's – which help us design an efficient electrical grid, as well as one – Dijkstra's – which helps us plan the quickest possible road trip.

1 Electrical Grids and MSTs: Prim's Algorithm

The Queen of Primland is looking to modernize her kingdom's electrical grid. This will involve building new transmission lines between various major cities. Now, building these new lines is costly, so the Queen would like to minimize the total length of the new transmission lines that must be built. Also, it turns out that it is much less expensive to build the new lines along existing roadways. But—and this is the good news—once the lines are built, it is quite cheap to send electrical power between any two locations connected to the grid, no matter what distance the electricity has to travel along the transmission lines.

The Queen has drawn up a map showing each of the major cities which must be connected to the grid. Also shown are the roadways which can be used, together with their lengths:

[INSERT PRIMLAND GRAPH HERE]

So the Queen's question becomes: Given the map above, how can we build an electrical grid along the roads, in such a way that every city is connected to the grid, and we minimize the total length of new transmission lines built?

You've probably guessed that we will be viewing the Queen's map as a graph, where the cities are the vertices and the roadways are the edges. This graph has some additional structure not seen in the previous Graphs chapter—namely, each edge e has a positive real number $w(e)$ associated with it. These numbers are called *edge-weights*, or just *weights*, and a graph which includes them is a *weighted graph*.[1] The *weight of a graph*, $w(G)$, is just the sum of all its edge-weights.

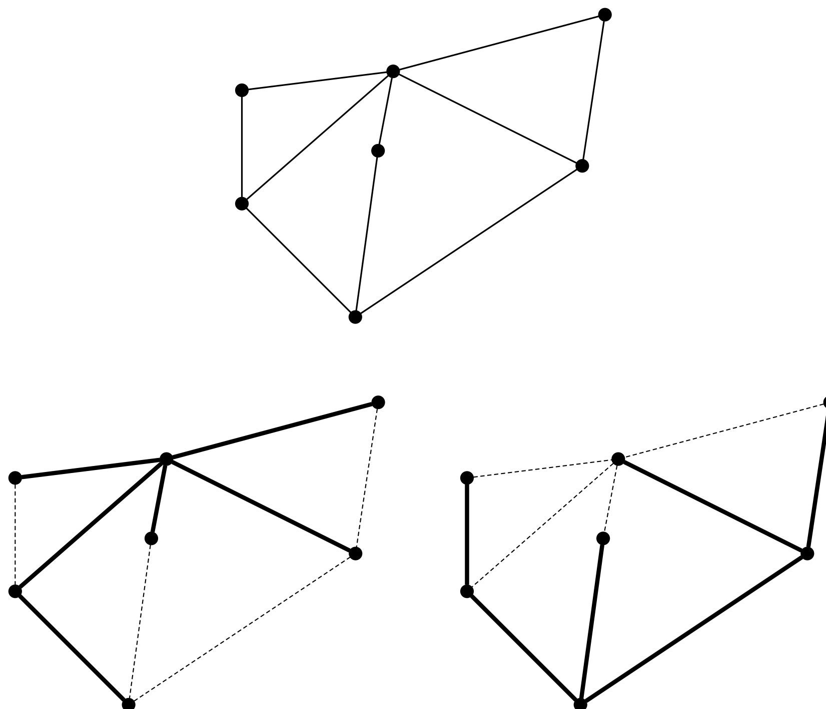
We are about to develop some tools which will allow us to effectively answer the Queen's question, but before we go on, you may wish to think about it a little on your own. What is the best grid you can make? What strategies would be useful in constructing an efficient grid?

We recall from the "Introduction to Graph Theory" chapter that a *tree* is a connected, acyclic graph. (Acyclic means having no cycles.) Also recall the theorem about trees which was proved in that chapter: A tree with n vertices will have exactly $n - 1$ edges.

[1] In some applications, it can be useful to allow edge-weights to be negative, but here we will restrict our attention to cases where the weights are positive.

Given any connected graph G (not necessarily a tree) with vertex set $\mathcal{V}$, we can try to find a a subgraph T of G which is a tree, and whose vertex set is also $\mathcal{V}$. In other words, we would like to discard some number of *edges* in order to get rid of all cycles in G, but we need to keep all the *vertices* and also make sure the resulting graph stays connected! Such a tree is called a *spanning tree* for the graph G, and there may be many of them.

Example 1. Below is a graph G along with two different spanning trees for G. As expected, each one has 6 edges, one fewer than the number of vertices in G. There are many other possible spanning trees as well—you might try sketching one or two others if you're so inclined.



This brings us to our first proof of the chapter. A word of caution to the reader: The proofs in this chapter (there are three of them) require close and careful reading in order to understand. If you're not used to reading mathematical proofs, and/or if this is your first time through the chapter, you might choose to skim them rather than try to follow every detail—and that's okay! Just be sure to come back when you're ready to give them a closer look.

Theorem 1. *Every connected graph G has at least one spanning tree.*

Proof. If G is a tree, then G is already its own spanning tree and we are done! Otherwise G is a connected graph which is *not* a tree, and so it must contain at least one cycle. Choose any cycle C, and delete any single edge $e = \{u, v\}$ that appears on C.

We claim that deleting an edge in this way cannot cause G to become disconnected. If this is true, then we can simply repeat the process of deleting an edge along a cycle until there are no more cycles remaining. The resulting graph T will have no cycles but remains connected and includes all of G's vertices. Thus T is a spanning tree for G.

It may seem obvious that deleting an edge along a cycle C won't cause a connected graph to become disconnected. Indeed, if we consider any two vertices which were previously connected by a path using the deleted edge, then we might hope a revised path P' can simply "go around cycle C the other way" and avoid the missing edge.

This intuition is a good one, but to make the proof rigorous there is a subtlety that should be addressed. According to the definition of a *path* in a graph G, all vertices should be distinct. If our original path P contained other vertices from C besides the endpoints of the deleted edge, then "going around C the other way" might very well cause us to retrace our steps, which is not allowed!

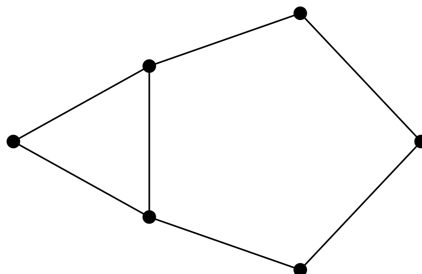
And so in the interest of demonstrating complete rigor, we will show how to patch this little gap. Let w_{entry} denote the first vertex along P which is part of the cycle C, and let w_{exit} be the last vertex along P which is part of C. Since we know P includes at least two distinct vertices from C (namely, u and v themselves), we therefore know that w_{entry} and w_{exit} are different from each other. (They might or might not be simply different labels for u and v themselves.)

Now we can construct a valid path $\widetilde{P}$ as follows: Follow path P from its initial vertex until you reach w_{entry}. Then go along cycle C toward vertex w_{exit} (in whichever direction does not require the missing edge!). Then, from w_{exit} continue on path P to its final vertex.

Since w_{entry} and w_{exit} are the first and last vertices on path P which intersect with cycle C, and since $\widetilde{P}$ only uses other vertices from C to travel between those two, we can be sure that we are not retracing steps at any point along $\widetilde{P}$. This completes our argument that removing edge e from G does not cause it to become disconnected, and hence our proof that every connected graph has an MST. □

Exercise 1. How many spanning trees does a cycle on n vertices have?

Exercise 2. Shown below is a graph which consists of a 3-cycle and a 5-cycle that share a single edge.

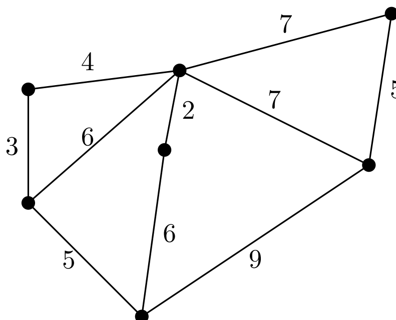


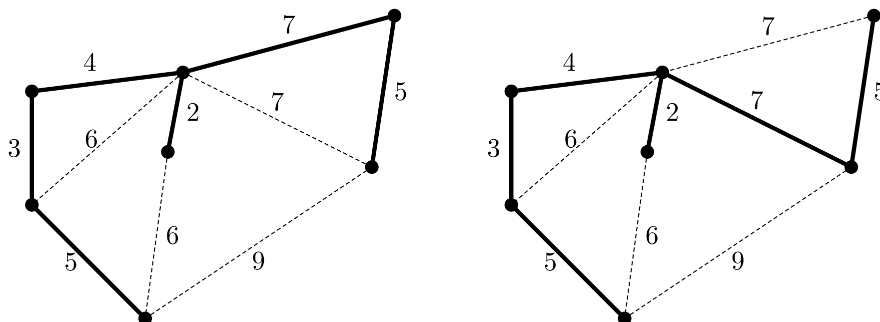
How many distinct spanning trees does this graph have?

Exercise 3. Generalize the previous exercise to a graph G which consists of an s-cycle and a t-cycle sharing a single edge, for any integers $s, t \geq 3$.

Having established that every connected, unweighted graph has at least one spanning tree (and perhaps many of them), we are ready to point out that every connected, *weighted* graph has at least one *minimum spanning tree (MST)*: a spanning tree which has the lowest possible sum for its edge-weights.

Example 2. We've reproduced the graph G from Example 1, but this time we added some edge-weights roughly corresponding to the length of the edges. Also shown are two different MSTs. You may wish to convince yourself that these are the only two minimum spanning trees for G, though there are many more spanning trees which are not MSTs.





A minimum-weight spanning tree is exactly what the Queen needs for her map, and we are just about ready to introduce our first *algorithm*, which solves the minimum-weight spanning tree problem. You might have a notion of an algorithm as "something a computer does," which is often true. But more generally, an algorithm means a reliable, step-by-step procedure used to compute desired information from input data, regardless of whether the steps are carried out by a computer, a human, a mechanical device, or a well-trained lemur. There is an algorithm you learned in grade school which takes a pair of positive integers and computes their sum. There is another algorithm which computes their product, and yet another which computes their greatest common divisor. (In fact, each of these problems has *multiple* valid algorithms—there's more than one way to skin a cat.)

A graph algorithm is one which takes a graph as the input data. The first algorithm we will examine is known as *Prim's Algorithm*, named after American mathematician Robert C. Prim, who published it in 1957. (Unbeknownst to Prim at the time, the algorithm was actually first discovered in 1930 by Czech mathematician Vojtěch Jarník, and was independently published in 1959 by Edsger W. Dijkstra, whose Shortest Path Algorithm we will examine later in the chapter.)

The idea behind the algorithm is that we will start at some vertex, and begin growing a tree from that vertex, continually adding the minimal edge which extends the tree (and does not cause a cycle). Here is a more formal description:

Algorithm 1: Prim's Algorithm

Input: A connected, weighted graph $G = (\mathcal{V}, \mathcal{E})$

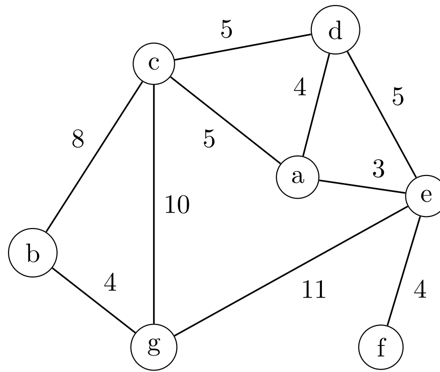
Output: A set of edges $\mathcal{E}_{\text{tree}} \subseteq \mathcal{E}$ which form a minimum spanning tree for G

1. Start with an empty vertex-list, $\mathcal{V}_{\text{tree}}$, and an empty edge-list, $\mathcal{E}_{\text{tree}}$.
2. Pick an arbitrary vertex v_0, and add it to $\mathcal{V}_{\text{tree}}$.
3. Among all edges which have exactly one endpoint in $\mathcal{V}_{\text{tree}}$, find the edge with smallest weight (if more than one works, choose any one of them). Add it to $\mathcal{E}_{\text{tree}}$ and add its other endpoint to $\mathcal{V}_{\text{tree}}$.
4. Repeat step 3 until there are no longer any edges of G with exactly one endpoint in $\mathcal{V}_{\text{tree}}$.
5. Output the list $\mathcal{E}_{\text{tree}}$.

That's it! Seems simple enough, right? Let's take a moment and see how the algorithm works on a relatively small graph.

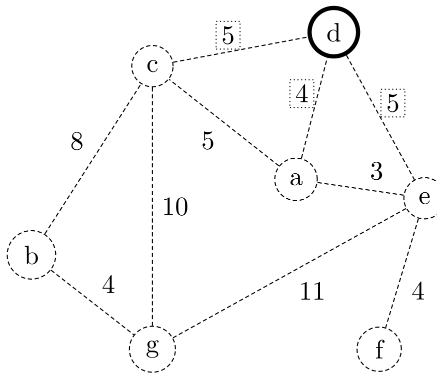
Example 3. We won't write out the lists $\mathcal{V}_{\text{tree}}$ and $\mathcal{E}_{\text{tree}}$ explicitly in each of the steps below, but rest assured that it would be quite simple to extract that information from the graphs shown below if anyone really wanted to (!).

Here is the graph we will use for our example. It has seven vertices, so the final output should be an MST which will have six edges.



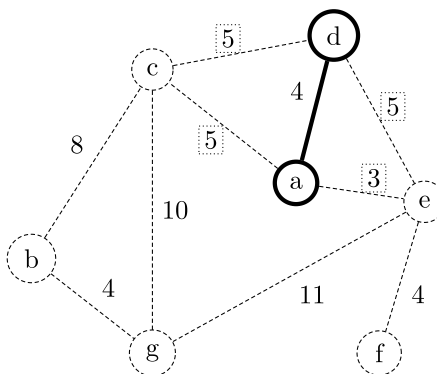
1. *Start with an empty vertex-list, $\mathcal{V}_{\text{tree}}$, and an empty edge-list, $\mathcal{E}_{\text{tree}}$.*
2. *Pick an arbitrary vertex v_0 and add it to $\mathcal{V}_{\text{tree}}$.*

   We arbitrarily pick $v_0 = d$. We will put a box around the edge-weights of those edges which are adjacent to the growing subtree, so it's easier to see which edge should be chosen in the next step.

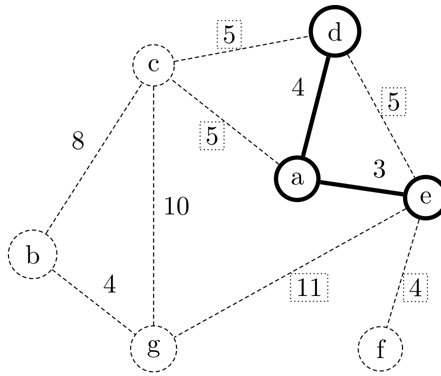


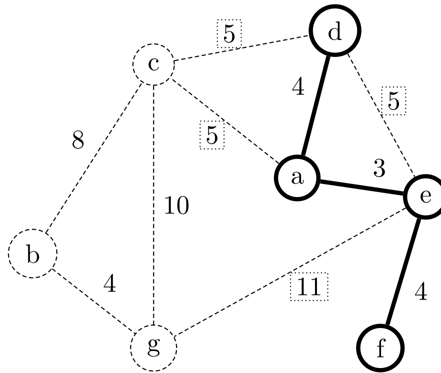
3. *Among all edges which have exactly one endpoint in $\mathcal{V}_{\text{tree}}$, find the edge with the smallest weight (if more than one works, choose any one of them). Add it to $\mathcal{E}_{\text{tree}}$ and add its other endpoint to $\mathcal{V}_{\text{tree}}$.*
4. *Repeat step 3 until there are no longer any edges of G with exactly one endpoint in $\mathcal{V}_{\text{tree}}$.*

   Of the edges adjacent to d, the smallest is ad with a weight of 4. $\mathcal{V}_{\text{tree}} = \{a, d\}$, $\mathcal{E}_{\text{tree}} = \{ad\}$.

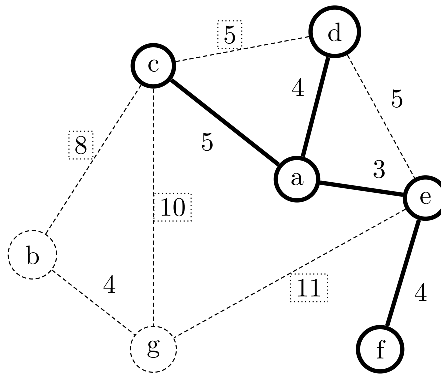


In the previous step, the edges with exactly one vertex in $\mathcal{V}_{\text{tree}}$ were cd, ac, de, ae. The smallest edge-weight is 3 so we add edge ae to the growing subtree, followed by edge ef with a weight of 4.

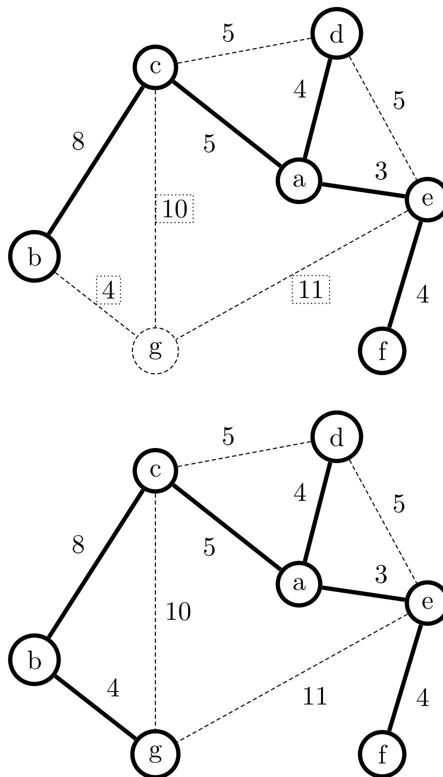




Now there are three adjacent edges with the minimal weight of 5: cd, ac, de. Adding de is not an option, as that would create a cycle. (We will go ahead and unbox it.) But cd and ac are both viable options, so we will choose arbitrarily between them—let's go with ac.



Since edge cd would give us a cycle, we will choose bc instead, with a weight of 8, followed by bg with a weight of 4.



5. *Output the list* $\mathcal{E}_{\text{tree}}$.

   Since all vertices are now part of $\mathcal{V}_{\text{tree}}$, there are no longer any edges that have exactly one vertex in $\mathcal{V}_{\text{tree}}$, so we output the minimum spanning tree $\mathcal{E}_{\text{tree}} = \{ac, ad, ae, bc, bg, ef\}$. **We did it!** The weight of our MST is $4 + 8 + 5 + 4 + 3 + 4 = 28$.

Remark. It is worth noting that there are actually two MSTs for this graph—we would have obtained the other one as our output if we happened to choose cd instead of ac as the fourth edge of the tree.

We will try out the algorithm on the Queen's graph soon, but first we should pause to ask: Does it really work? Sure, it did fine on the small graph above, but we could have gotten that one just by looking at it. What about a bigger, messier, more complicated graph? What if it had a thousand vertices—or a million? In order to be sure, we need to analyze the algorithm closely and *prove* that it works as advertised.

Theorem 2. *Given any connected, weighted graph G, Prim's Algorithm outputs a minimum spanning tree of G.*

Proof. The crucial step in the algorithm is step 3, which repeatedly adds one new edge and vertex to the collection. Our proof comes in three parts.

Part 1. Show the output is a *tree.* Consider the subgraph $T_i \subseteq G$ which consists of all edges in $\mathcal{E}_{\text{tree}}$ (and their endpoints) after i repetitions of step 3. Note that $T_0 = \{v_0\}$ is a tree, and that each time we add an edge, exactly one of its vertices was already in T_i. This ensures that T_{i+1} remains connected, and also that adding the new edge will not create a cycle. It follows that T_i is a tree, for all i. Since T_i is always a tree, it is particularly a tree when the algorithm terminates, and this shows that the output of Prim's Algorithm, call it T, is a tree.

Part 2. Show the output is a *spanning* tree. Speaking of termination, note that step 4 instructs us to stop repeating step 3 as soon as there are no edges in G with exactly one endpoint in $\mathcal{V}_{\text{tree}}$. We know the resulting output T is a tree, but can we be sure it is it a *spanning* tree for G—that is, can we be sure it includes all vertices of G? We leave it as an exercise for the reader to show that it does (see below).

Part 3. Show the output is a *minimum* spanning tree. Finally we must show that our output T has the minimum possible weight among all spanning trees of G. Let $\widetilde{T}$ be *any* MST for the graph G. Of course if $T = \widetilde{T}$ then we're done already. If not, though, then we claim we can adjust $\widetilde{T}$ slightly, by removing one edge and replacing it with an edge from T, and the resulting graph will still be an MST for G. By repeating this process on the newly created MST (as many times as necessary), we can transform the original MST $\widetilde{T}$ into our output T, and at every step of the transformation the graph will be an MST—including the last step, which is T itself!

So if we can show how to swap out an edge of $\widetilde{T}$ and replace it with one from T such that the result is still an MST, then we will be all done with the proof. To see how we can do this, think about the very first edge added to T which does *not* appear in $\widetilde{T}$. Call this edge $e = \{u, v\}$, and suppose it was added in step i of the algorithm (with u already part of the tree and v added in step i). It follows that the subtree T_{i-1} is shared in common by T and $\widetilde{T}$. (Of course it's possible that $T_{i-1} = T_0 = \{v_0\}$.)

Since the spanning tree $\widetilde{T}$ does not contain edge e, it must contain some other path from u to v which doesn't use e. Think about the first edge along that path which is not part of the subtree T_{i-1}, say $\widetilde{e} = \{\widetilde{u}, \widetilde{v}\}$, where $\widetilde{u} \in T_{i-1}$ and $\widetilde{v} \notin T_{i-1}$. It must be the case that $w(e) \leq w(\widetilde{e})$, because if not then Prim's Algorithm would have chosen $\widetilde{e}$ in step i, rather than e. On the other hand, we cannot have a strict inequality $w(e) < w(\widetilde{e})$, because in that case we could remove edge $\widetilde{e}$ from the path in $\widetilde{T}$, and replace it with edge e. The resulting graph would still be connected and acyclic, so it would be a new spanning tree with strictly smaller edge-weight—a contradiction since $\widetilde{T}$ was assumed to be an MST already. The only remaining possibility is that $w(e) = w(\widetilde{e})$. In this case, we can still remove edge $\widetilde{e}$ and replace it with e in $\widetilde{T}$, and the result will be a different minimum spanning tree.

But notice this is exactly what we needed to show—namely, that there is some edge we can remove from $\widetilde{T}$ and replace with an edge from T, so that the resulting graph is still an MST. This completes the proof of correctness for Prim's Algorithm. □

Exercise 4. Complete Part 2 in the proof of Prim's Algorithm by showing that once there are no more edges in G with exactly one endpoint in $\mathcal{V}_{\text{tree}}$, $\mathcal{V}_{\text{tree}}$ must contain ALL vertices of G.

Exercise 5. The following algorithm is exactly the same as Prim's Algorithm except for the underlined words. Explain why it does NOT work, and give a specific example of a graph for which it fails to output a minimum spanning tree.

Algorithm: Flawed Version of Prim's Algorithm

Input: A connected, weighted graph $G = (\mathcal{V}, \mathcal{E})$

Output: A set of edges $\mathcal{E}_{\text{tree}} \subseteq \mathcal{E}$ which form a minimum spanning tree for G (?)

1. Start with an empty vertex-list, $\mathcal{V}_{\text{tree}}$, and an empty edge-list, $\mathcal{E}_{\text{tree}}$.
2. Pick an arbitrary vertex v_0, and add it to $\mathcal{V}_{\text{tree}}$.
3. Among all edges which have <u>at least one</u> endpoint in $\mathcal{V}_{\text{tree}}$, find the edge with smallest weight (if more than one works, choose any one of them). Add it to $\mathcal{E}_{\text{tree}}$ and add its other endpoint to $\mathcal{V}_{\text{tree}}$.
4. Repeat step 3 until there are no longer any edges of G with exactly one endpoint in $\mathcal{V}_{\text{tree}}$.
5. Output the list $\mathcal{E}_{\text{tree}}$.

All right, we've seen the algorithm in action on a small graph and now we've proven that it always works. No more stalling! It's time to apply Prim's Algorithm to the Queen's map and find the optimal electrical network. Why don't you do the honors?

Exercise 6. Apply Prim's Algorithm to the Queen's map and find the optimal electrical network. What is the total distance of the smallest possible network for the Kingdom of Primland?

2 Another Way to Skin the Cat: Kruskal's Algorithm

Prim's Algorithm works well, but it's not the only game in town. In this section, we'll see an alternative way to compute the minimum spanning tree: *Kruskal's Algorithm*, named for Joseph Kruskal, the American mathematician who first published the method in 1956.

Prim's Algorithm starts with any old vertex and grows a subtree outward from that starting point. Kruskal's Algorithm, by contrast, does not grow a tree outward from a single starting point, but rather pieces it together haphazardly from edges strewn throughout the graph. Really, the idea is simple: We repeatedly select the smallest unchosen edge which doesn't create any cycles. And this simple idea works! Here is a more formal description of the algorithm.

Algorithm 2: Kruskal's Algorithm

Input: A connected, weighted graph $G = (\mathcal{V}, \mathcal{E})$

Output: A set of edges $\mathcal{E}_{\text{tree}} \subseteq \mathcal{E}$ which form a minimum spanning tree for G

1 Start with an empty vertex-list, $\mathcal{V}_{\text{tree}}$, and an empty edge-list, $\mathcal{E}_{\text{tree}}$.

2 Among all edges not already in $\mathcal{E}_{\text{tree}}$, find the edge with smallest weight such that adding it to $\mathcal{E}_{\text{tree}}$ would not create a cycle (if more than one edge satisfies this condition, choose any one of them). Add it to $\mathcal{E}_{\text{tree}}$ and add its endpoints to $\mathcal{V}_{\text{tree}}$.

3 Repeat step 2 until it is no longer possible to add an edge to $\mathcal{E}_{\text{tree}}$ without creating a cycle (or until all edges of G have been added to $\mathcal{E}_{\text{tree}}$).

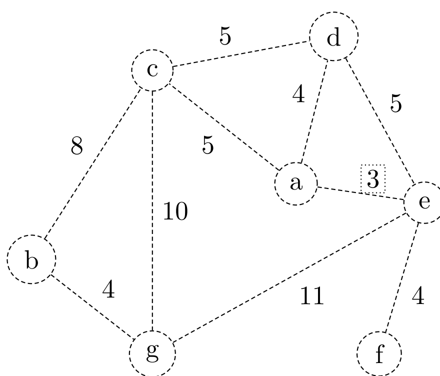
4 Output the list $\mathcal{E}_{\text{tree}}$.

We're about to see this method in action, but first a quick definition. As you know, a connected graph which contains no cycles is called a tree. In general, an acyclic graph might be disconnected, in which case it would consist of multiple components, each one a tree. So mathematicians decided it made perfect sense to refer to a general acyclic graph as, you guessed it, a *forest*! (Note that in graph theory, even a single tree counts as a forest, which is both mathematically convenient and linguistically questionable.)

Example 4. All right, let's take this thing out for a ride. For comparison's sake, we'll try it on the same graph we used as an example for Prim's Algorithm.

1. *Start with an empty vertex-list, $\mathcal{V}_{\text{tree}}$, and an empty edge-list, $\mathcal{E}_{\text{tree}}$.*

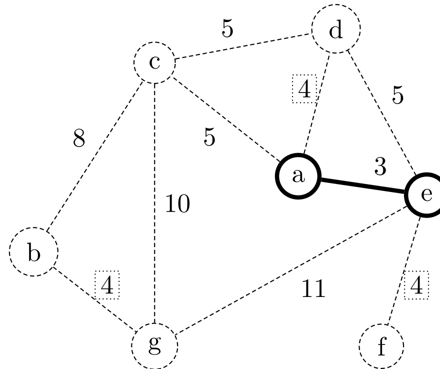
   Since Kruskal's Algorithm adds edges from smallest to largest, we will place a box around any edges which are coming up next in order of increasing weight (starting here with the very smallest edge).



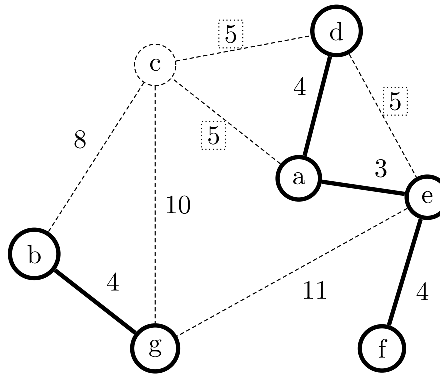
2. *Among all edges not already in $\mathcal{E}_{\text{tree}}$, find the edge with smallest weight such that adding it to $\mathcal{E}_{\text{tree}}$ would not create a cycle (if more than one edge satisfies this condition, choose any one of them). Add it to $\mathcal{E}_{\text{tree}}$ and add its endpoints to $\mathcal{V}_{\text{tree}}$.*

3. *Repeat step 2 until it is no longer possible to add an edge to $\mathcal{E}_{\text{tree}}$ without creating a cycle (or until all edges of G have been added to $\mathcal{E}_{\text{tree}}$).*

   The smallest edge is ae with a weight of 3; we add it to the forest first.



   Now three different edges have a weight of 4: bg, ad, and ef. We will go ahead and add all of them at once in the next figure (collapsing three steps of the algorithm into one step of explanation), because doing so does not create any cycles.

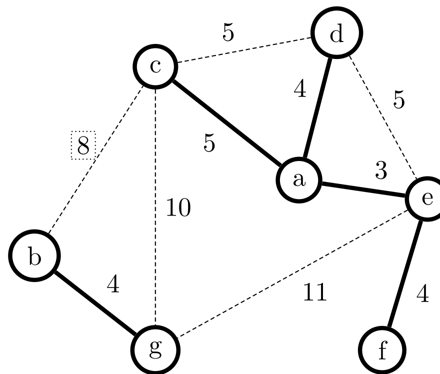


   Again there are three edges to consider: ac, ad, and cd, each of which has a weight of 5. But this time we cannot simply add all of them to the forest at once, because cycles will arise. So we will try adding them one at a time (just like the algorithm does).

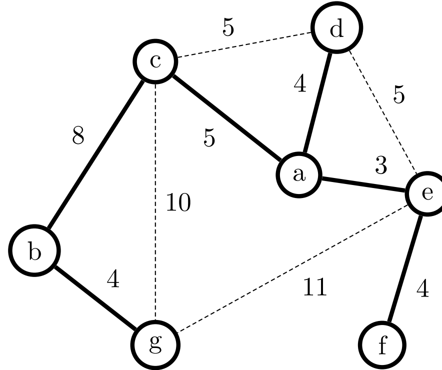
   First look at ac. Adding it would not create a cycle, so we go ahead and do so.

   Next look at cd. Having already added ac, including cd as well would create a cycle, so we skip it.

   de also gives a cycle, so we skip it too.



The next edge to add is bc, with a weight of 8. There are then four edges not yet added, but including any one of them would create a cycle. We are done growing our forest!



4. *Output the list* $\mathcal{E}_{\text{tree}}$.

   The MST generated by Kruskal's Algorithm is $\mathcal{E}_{\text{tree}} = \{ac, ad, ae, bc, bg, ef\}$, which happens to be the same one we got using Prim's Algorithm. We could have gotten the other MST for G if we had added edge cd first, instead of ac, when considering the edges of weight 5.

Having gone through a smallish example in detail, we are ready to take the next step. Just as with Prim's Algorithm, we are going to prove that the algorithm does really work—not just on the smallish graph we just tested, but on any possible (connected) graph G, no matter how massive, complicated, or weird.

Theorem 3. *Given any connected, weighted graph G, Kruskal's Algorithm outputs a minimum spanning tree of G.*

Proof. At each step in Kruskal's Algorithm, the growing subgraph will always be a forest, because (according to step 3) we will never add an edge which creates a cycle. Again our proof comes in three parts.

Part 1. Show the output is a *tree*. Consider the forest F_i which results from repeating step 3 exactly i times. We need to verify that that when the algorithm is complete, the resulting output T will be not just a forest, but in fact a *single* tree. Suppose that the forest F_i (for some particular i) consists of two or more distinct trees, $T_1, T_2, \ldots, T_k$ for $k \geq 2$. We claim that in this situation, there exists at least one edge which has not yet been chosen by the algorithm, and that will not create a cycle if it is added to F_i.

So consider the two distinct trees T_1 and T_2 in F_i. Let's also pick two vertices $v_1 \in T_1$ and $v_2 \in T_2$. We know the original graph G is connected, which means there is some path in G between v_1 and v_2. Our idea is to show that one of the edges along this path has not yet been chosen as part of F and will not create a cycle. Notice that v_1 is the first vertex on the path and is in T_1, while v_2 is the last vertex on the path and is not in T_1. Thus there has to be some vertex on the path which is the very last one to be in T_1 (it could be v_1 itself, or it might be some later vertex). Let $e = \{u, v\}$ be the next edge in the path, so that $u \in T_1$ and $v \notin T_1$. This is the edge we've been looking for.

First, $e = \{u, v\}$ cannot be part of the growing forest F_i already. It's clearly not part of T_1, because its endpoint v is not in T_1. And it's also not part of any other tree in F (otherwise u would be in both T_1 and that other tree, but two disconnected trees cannot share a vertex). So e has not yet been added to F_i.

Second, adding e will not create a cycle. To see why, think about vertex v which is not contained in T_1. If v is not currently in any subtree in F_i, then adding e just extends the tree T_1 by one edge and vertex. On the other hand, if v *is* part of another subtree in F_i, then adding e could create a cycle only if there is already a path in F_i between u and v. But this would mean that T_1 and the other tree were actually connected to each other, whereas our assumption was that they are distinct subtrees in F_1. Hence, adding edge e will not create a cycle.

Since there is at least one edge, namely e, which is not yet in F_i and will not create a cycle, Kruskal's Algorithm will repeat step 3 at least one more time (though it wouldn't have to choose edge e itself,

necessarily!). Since this outcome depended only on the assumption that there were at least two distinct trees in the growing forest F_i, it follows that when the algorithm does terminate, its output T must be a forest which actually consists of only a single tree.

Part 2. Show the output is a *spanning* tree. Speaking of termination, note that step 3 says to terminate when we can no longer add any edge without creating a cycle. This phrasing was useful because it immediately guaranteed the output had no cycles, but we also need to know that the output contains all of the vertices of G. In other words: We've shown that the output is a single tree, but is it a *spanning* tree for G? Just as we did for Prim's Algorithm, we leave it as an exercise for the reader to show that the output T is a spanning tree (see below).

Part 3. Show the output is a *minimum* spanning tree. So the output T is a spanning tree, but we still must show that it is a minimum spanning tree. We will show that at every stage of its construction, the growing forest F_i is a subgraph of some MST $\widetilde{T}_i \subseteq G$. In particular, this will mean that the output final output T is a subtree of some MST $\widetilde{T}$. Since T and $\widetilde{T}$ have the same vertex set, $\mathcal{V}$, this can only happen if $T = \widetilde{T}$, which proves that T is an MST, as desired.

For our base case, the forest F_1 contains just a single edge e_1 which is of minimum weight in G. We know that F_1 is a subgraph of some MST for G, because we could have named one of e_1's endpoints as v_0 in Prim's Algorithm, and then continued Prim's Algorithm by choosing e_0 as the first edge!

Now consider F_i for some particular i, and assume that F_i is currently a subgraph of some minimum spanning tree $\widetilde{T}_i$. We need to show that, after adding the next edge e, the resulting forest F_{i+1} is still a subgraph of some MST (not necessarily $\widetilde{T}_i$). Now the next edge, e, is a minimal edge such that $e \notin F_i$, and adding e does not create any cycle in F_i. Of course if $e \in \widetilde{T}_i$ then $F + e$ is still a subgraph of $\widetilde{T}_i$, as required. Otherwise $e \notin \widetilde{T}_i$. In this case, consider the graph $\widetilde{T}_i + e$. Since $\widetilde{T}_i$ is a spanning tree for G, e's endpoints are already in $\widetilde{T}_i$ and adding e will create a cycle in $\widetilde{T}_i + e$.

Now this cycle in $\widetilde{T}_i + e$ has at least one edge, say $\widetilde{e}$, which was not already in F_i (otherwise adding e to F_i would cause the cycle to appear, and e would never have been selected in the first place). Since F_i is a subgraph of the acyclic graph $\widetilde{T}_i$, adding $\widetilde{e}$ to F_i (instead of e) would not create a cycle in F. It follows that $w(e) \le w(\widetilde{e})$, or else the algorithm would have selected $\widetilde{e}$ as the next edge instead of e. But we cannot have a strict inequality $w(e) < w(\widetilde{e})$, because if that were the case we could replace $\widetilde{e}$ with e in $\widetilde{T}_i$, and reduce its edge-weight—contradicting the fact that $\widetilde{T}_i$ is a minimum spanning tree! This forces $w(e) = w(\widetilde{e})$. In this case, we can replace $\widetilde{e}$ with e in $\widetilde{T}_i$ without changing the total edge-weight, and get a different MST—call it $\widetilde{T}_{i+1}$. This name is justified because $\widetilde{T}_{i+1}$ contains the forest F_i as well as the new edge e, and hence $F_{i+1} = F_i + e$ is contained in the MST $\widetilde{T}_{i+1}$.

Since each F_i is contained in some MST, in particular the final output spanning tree T is also contained in some MST, which is exactly what we needed to show. □

Exercise 7. Complete Part 2 of the proof by explaining why the single tree T which is output by the algorithm does indeed contain every vertex of the original graph G.

Exercise 8. True or false? "If e is a unique minimal edge in a connected graph—one such that $w(e) < w(f)$ for all edges $f \neq e$—then e must appear in *every* minimum spanning tree."

If true, explain why. If false, provide a counterexample.

Exercise 9. True or false? "If e is a unique maximal edge in a connected graph—one such that $w(e) > w(f)$ for all edges $f \neq e$—then e cannot appear in *any* minimum spanning tree."

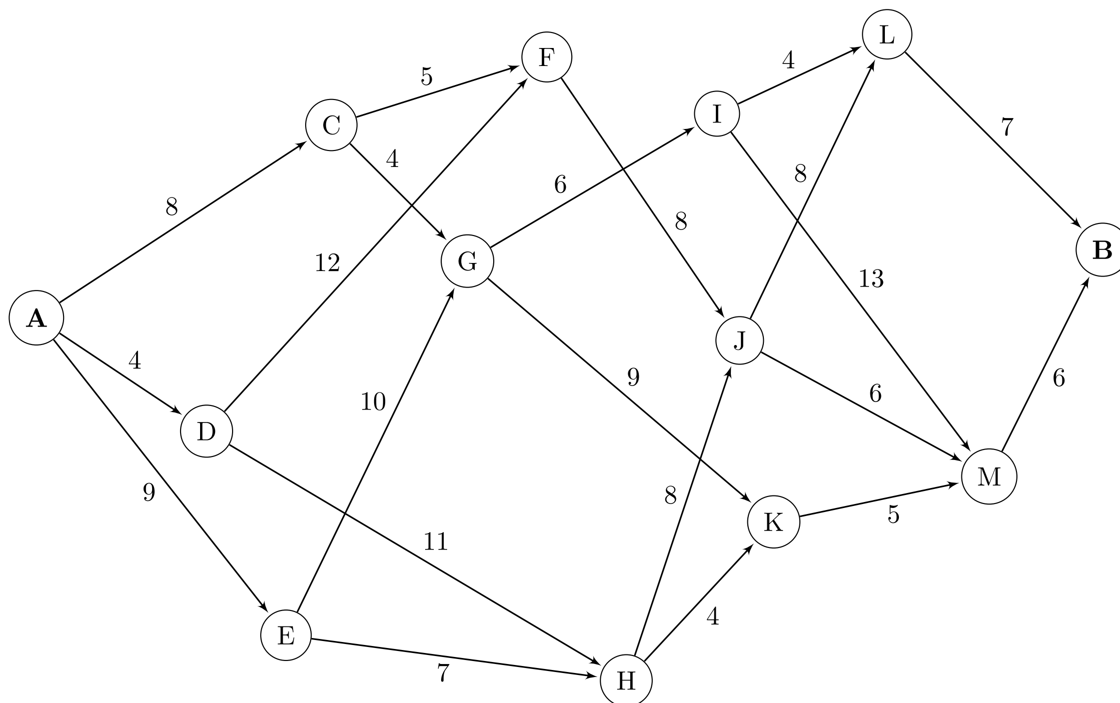
If true, explain why. If false, provide a counterexample.

Exercise 10. Apply Kruskal's Algorithm to the Queen's map and find the optimal electrical network (again!). Did you get the same MST as you did with Prim's Algorithm?

3 Dijkstra's Algorithm: Finding the Shortest Path

It's time to shift gears away from minimum spanning trees and to a new problem: finding the shortest path.

Here is a map of the City of Townsville. Townsville's city planner was an avant-garde type and designed a rather unique, convoluted network of roads—much to the dismay of the city's residents, who it must be said were more interested in the functionality than the artistry of their road system. To make matters worse, the roads are all one-way!



A map with one-way roads can be represented by a (weighted) *directed graph*, also known as a *digraph* for short. Digraphs differ from ordinary (undirected) graphs in the way their edge-sets are defined. For digraphs, the edge-set $\mathcal{E}$ is made up of ordered pairs of vertices from $\mathcal{V}$: (u, v) represents a *directed edge* which goes *from* u, *to* v.

Here is a simple digraph:

[INSERT SIMPLE DIGRAPH]

There are some important differences to keep in mind when working with digraphs. First, with digraphs there may be two distinct edges between a single pair of vertices—(u, v) and also (v, u). Having both directed edges between u and v is just like having a two-way street. (And for this reason, undirected graphs can actually be thought of as just a very special subcategory of digraphs.)

Second, when we talk about paths in a digraph, we mean *directed paths*: ones which don't go the wrong way down a one-way street. (We call a digraph *connected* if there are always two directed paths—one in each direction—between every pair of vertices.)

Dijkstra's Algorithm, named for Edsger W. Dijkstra who created it in the late 1950s, takes as input a digraph G with two special vertices—a *startpoint* A and a *goal* B. The output is a list of vertices giving the shortest directed path from A to B. (If there is no path from A to B, the algorithm will detect this and output "No path exists.")

The idea of Dijkstra's Algorithm is to label each vertex $v \neq A$ with a *tentative distance* from A, which we will write $\mathsf{dist}(v)$. The tentative distance represents the maximum distance that vertex *could* be from A, based on the algorithm's current state of knowledge. The algorithm then "visits" vertices in a well-chosen order, working its way outward from A and reducing the tentative distance labels as necessary. After a vertex is visited, its label becomes fixed and is guaranteed to reflect the correct distance from A.

At the same time, the algorithm is also keeping track of a *tentative parent* for each vertex. The tentative parent, which we will denote $\mathsf{parent}(v)$, is another vertex which represents the algorithm's best guess, given

its current knowledge, of which vertex comes just before v in a shortest path from A to v.

This algorithm's formal description is a little bit more involved than that for Prim's or Kruskal's Algorithm. As you read through it, try to get an overall sense for what the algorithm is doing. (As always, we'll go through a detailed example afterward.)

Algorithm 3: Dijkstra's Algorithm

Input: A weighted digraph $G = (\mathcal{V}, \mathcal{E})$ with two specially marked vertices: startpoint A and goal B

Output: A set of vertices which form a minimum-distance path from A to B, or the message "No path exists."

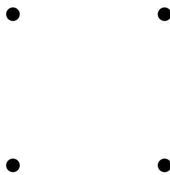
1. Assign a distance label $\mathsf{dist}(A) = 0$ to the startpoint A, and mark it as the *current-vertex* initially (i.e. the one currently being visited by the algorithm; we will use the symbol v_{cur}). Label all other $v \in \mathcal{V}$ with the tentative distance $\mathsf{dist}(v) = \infty$, as well as tentative parent $\mathsf{parent}(v) = \mathsf{None}$. All vertices are *unvisited* until the algorithm explicitly marks them *visited*.
2. Do the following for every unvisited neighbor v_{nbr} of the current-vertex v_{cur}: Compare the neighbor's tentative distance label, with the distance required to travel first to v_{cur} and then along the edge $e = (v_{\mathsf{cur}}, v_{\mathsf{nbr}})$. That is, compare the two values $\mathsf{dist}(v_{\mathsf{nbr}})$, and $\mathsf{dist}(v_{\mathsf{cur}}) + w(e)$. If $\mathsf{dist}(v_{\mathsf{cur}}) + w(e)$ is smaller, then update $\mathsf{dist}(v_{\mathsf{nbr}})$ to be that smaller value, and update its tentative parent to be the current-vertex: $\mathsf{parent}(v_{\mathsf{nbr}}) := v_{\mathsf{cur}}$.
3. Mark v_{cur} *visited*. If v_{cur} was the goal vertex, B, jump ahead to step 6.
4. If all unvisited vertices have $\mathsf{dist}(v) = \infty$, halt and output "No path exists."
5. Otherwise choose a new current-vertex by setting v_{cur} equal to an unvisited vertex v which has the smallest tentative distance, $\mathsf{dist}(v)$. Return to step 2.
6. If you are here then the goal vertex B was just marked *visited* in step 3. Now the length of the shortest path from A to B is given by $\mathsf{dist}(B)$. Moreover, the actual path itself can be obtained by tracing the chain of tentative parents from B back to A (and then reversing order to give a path from A to B).

Example 5. Consider the following weighted digraph G, with startpoint A and B as shown:

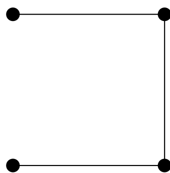
4 Challenge

Exercise 11. In discussing minimum spanning trees, we assumed that the graph edges available to us, and their lengths, were completely determined in advance (we had to build the electrical transmission lines along pre-existing roadways). What if we dropped this requirement, and allowed the creation of pathways anywhere?

For instance, consider the four corners of a square with edge length 1:



Certainly we could connect these four points together using 3 units of wire, for instance like so:



But if the wire can be laid anywhere, not just on straight lines between pairs of points, then we can join up all four points using less than 3 units of wire.

(a) Show how to connect the four points up using strictly less than 3 units of wire.

(b) Show how to connect the four points up using strictly less than 2.8 units of wire. (This one is tricky if you haven't seen it before.)