CMSC 351: Spanning Trees

Justin Wyss-Gallifent

November 27, 2023

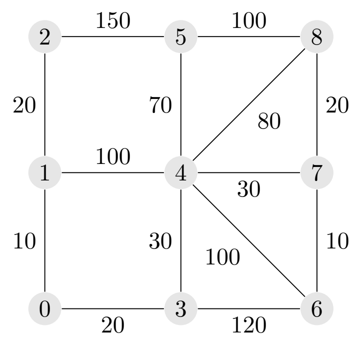
1 Introduction

We've discussed shortest paths and Dijkstra's Algorithm, which finds the minimal cost tree from a given starting vertex to all other vertices, but these involve choosing an initial vertex.

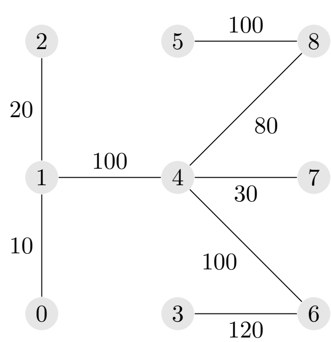
Suppose now we have a weighted graph G and we wish to find a subgraph of G which is not only a tree but which spans the original graph (includes all vertices), and also has minimal cost, where "minimal" means taken over all possible trees.

Consider the following graph:

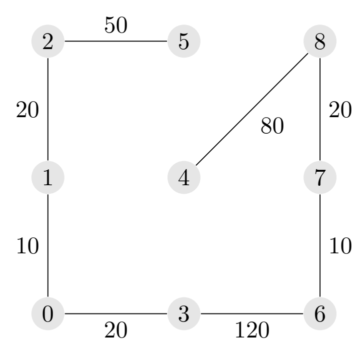
Example 1.1. Consider this weighted graph:



There are many ways to get a spanning tree. Here are two:



and



Observe that the spanning tree on the left has a total cost of $20+10+100+100+80+30+100+120 = 560$ while the spanning tree on the right has a total cost of $150+20+10+20+120+10+20+80 = 420$.

Clearly the one on the right costs less, but could we have done better?

Before proceeding further:

Theorem 1.0.1. Every tree with n vertices has $n - 1$ edges.

Proof. By structural induction. A single vertex is a tree with $n = 1$ vertices and $n - 1 = 0$ edges. A new tree is created by adding an edge to an existing vertex and a new vertex at the other end. This contributes 1 to the vertex count and the edge count. $\mathcal{QED}$

2 Prim's Algorithm

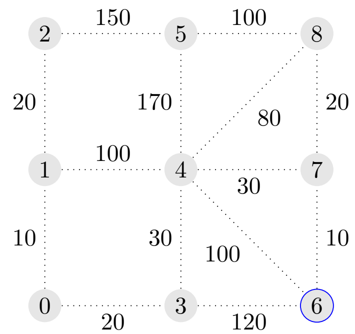
2.1 The Algorithm

Prim's Algorithm works by growing a tree $T \subseteq G$. In what follows denote by $v(H)$ the vertex set of a graph H and by $e(H)$ the edge set of a graph H:

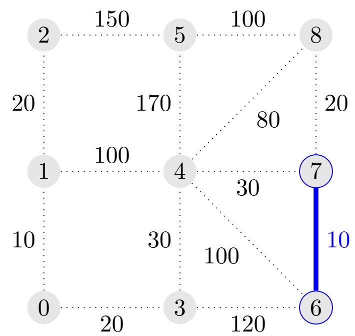
1. Define T to be a graph consisting of one vertex in G. That is, $v(T) = x$ for any $x \in v(G)$ and $e(T) = \emptyset$.
2. While $v(T) \neq v(G)$ pick a minimal weight edge $e(x, y)$ with $x \in v(T)$ and $y \in v(G) - v(T)$ and add the edge and the vertex to T, so $v(T) = v(T) + y$ and $e(T) = e(T) + edge(x, y)$.

Note 2.1.1. Note that the condition that the minimum cost edge adds a new vertex to the tree is equivalent to insisting that the minimum cost edge does not form a cycle.

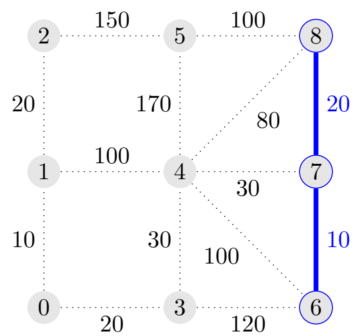
Example 2.1. Consider the example from earlier. Here we have excluded all edges by dashing them. We'll start with an arbitrary choice of vertex 6.



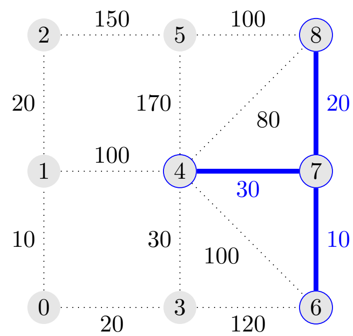
We now choose a minimum weight edge going between a vertex in $\{6\}$ and a vertex in $\{0, 1, 2, 3, 4, 5, 7, 8\}$ We choose $edge(6, 7)$ which picks up vertex 7:



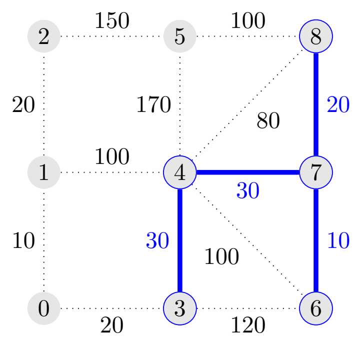
We now choose a minimum weight edge going between a vertex in $\{6, 7\}$ and a vertex in $\{0, 1, 2, 3, 4, 5, 8\}$ We choose $edge(7, 8)$ which picks up vertex 8:



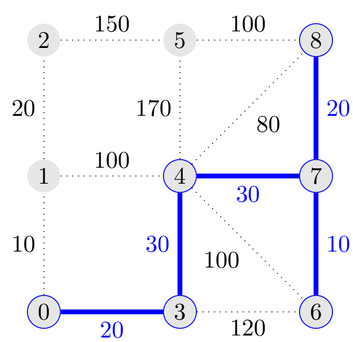
We now choose a minimum weight edge going between a vertex in $\{6, 7, 8\}$ and a vertex in $\{0, 1, 2, 3, 4, 5\}$ We choose $edge(4, 7)$ which picks up vertex 4:



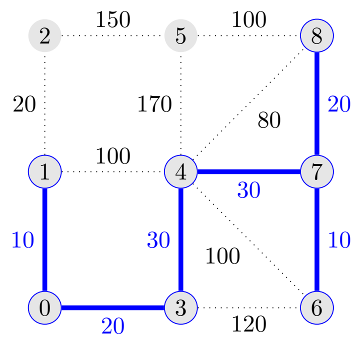
We now choose a minimum weight edge going between a vertex in $\{4, 6, 7, 8\}$ and a vertex in $\{0, 1, 2, 3, 5\}$ We choose $edge(3, 4)$ which picks up vertex 3:



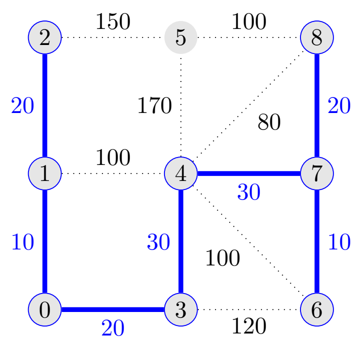
We now choose a minimum weight edge going between a vertex in $\{3, 4, 6, 7, 8\}$ and a vertex in $\{0, 1, 2, 5\}$ We choose $edge(0, 3)$ which picks up vertex 0:



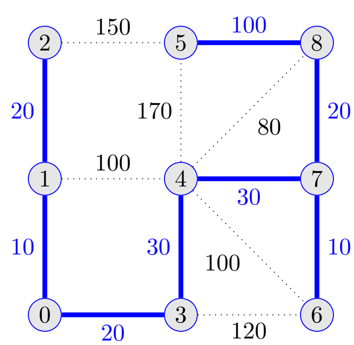
We now choose a minimum weight edge going between a vertex in $\{0, 3, 4, 6, 7, 8\}$ and a vertex in $\{1, 2, 5\}$ We choose $edge(0, 1)$ which picks up vertex 1:



We now choose a minimum weight edge going between a vertex in $\{0, 1, 3, 4, 6, 7, 8\}$ and a vertex in $\{2, 5\}$ We choose $edge(1, 2)$ which picks up vertex 2:



We now choose a minimum weight edge going between a vertex in $\{0, 1, 2, 3, 4, 6, 7, 8\}$ and a vertex in $\{5\}$ We choose $edge(5, 8)$ which picks up vertex 8:



We have now picked up all vertices and we are done. We now have a minimum cost spanning tree. The total cost is 240.

2.2 Mathematics for Prim

Theorem 2.2.1. Prim's Algorithm creates a minimal spanning tree.

Proof. Let G be a weighted and connected graph on which we have run Prim's Algorithm resulting in the tree which contains the edges $\{e_1, ..., e_{n-1}\}$.

Since the empty set $\{\}$ may be extended (somehow) to a minimal spanning tree but the set $\{e_1, ..., e_{n-1}\}$ may not (it's already a tree and adding any edge will add a cycle and ruin this), there is some maximum i such that $S = \{e_1, ..., e_{i-1}\}$ may be extended (somehow) to a minimal spanning tree but $\{e_1, ..., e_i\}$ may not be extended (somehow) to a minimal spanning tree. Note that S may be empty here!

Define W to be the set of nodes selected by Prim's Algorithm just before e_i is added. If we put $e_i = edge(x, y)$ this means that x is in W but y is not.

Define U to be any minimal spanning tree containing S. Since there is certainly a path P from x to y in U, and since x is in W but y is not in W, if we follow this path we eventually encounter some node x' which is still in W followed by a node y' which is not in W.

Define the set:

$$T = U - \{edge(x', y')\} + \{edge(x, y)\}$$

Observe that the deletion disconnects the tree U because it removes the only connection from x' to y' and then the addition reconnects it because x' and y' are now connected by following P from x' to x, then going to y, then following P from y to y'.

Consider now:

- If $c(x, y) < c(x', y')$ then the total cost of T is less than that of U which contradicts the fact that U is a minimal spanning tree.
- If $c(x, y) > c(x', y')$ then Prim's Algorithm would have chosen (x', y') (which it could have done since $x' \in U$) instead of (x, y) but it didn't.
- If $c(x, y) = c(x', y')$ then T is also a minimal spanning tree. However note that T contains the edges in S and edge $e_i = edge(x, y)$, this tells us that $\{e_1, e_2, ..., e_{i-1}, e_i\}$ can be extended to a minimal spanning tree (that being T) which contradicts our assumption.

In all cases we have contradictions and we are done.

$\mathcal{QED}$

2.3 Pseudocode and Time Complexity

Loosely speaking the pseudocode is easy:

```
\\ PRE: G is a graph
T = graph consisting of an arbitrary vertex in G
while vertices in T != vertices in G
  select (u,v) = min cost edge (u,v) with u in T and v in G-T
  T = T + (u,v)
end
\\ POST: T is a minimal spanning tree
```

The devil is in the details, however, specifically in the critical line:

`select (u,v) = min cost edge (u,v) with u in T and v in G-T`

Here are some options:

- This can be easily done in $\Theta(V^3)$:

  If G is stored as an adjacency matrix AM we create a boolean list INT of length V storing whether each vertex is in T.

  Then to accomplish our critical line we scan every entry in AM and for each we use INT to check if one vertex is in T and one is not, and we pick the one with minimum cost and include it in T.

  The scanning is $\Theta(V^2)$ and the checks are $\Theta(1)$ and since the while loop runs V times, this entire process is $\Theta(V^3)$.

- There is a more refined approach with an adjacency matrix which is $\Theta(V^2)$:

  If G is stored as an adjacency matrix then in addition to INT above we create a distance list $DIST$ of length V full of keys each equal to `inf` except we set $DIST[s] = 0$ and we create a predecessor list $PRED$ of length V.

  To jump-start the process we first include s in T and go through all vertices adjacent to s and for each we label their $DIST$ to be the edge weight to s and for each we label their $PRED$ to be s.

  Then we iterate. To accomplish our critical line we scan pick the vertex x with minimum $DIST$ and which is not already in T and we include it in T along with the edge stored in $PRED$. Then for each vertex y adjacent to x and not in T we update $DIST$ to be the minimum of its current value and its distance to x. If we update $DIST$ the we also update $PRED$ to update the predecessor of x.

  The scanning is $\Theta(V)$ and the "for each vertex" is $\Theta(V)$ but these are done in series so together they are $\Theta(V + V) = \Theta(V)$ and since the while loop runs V times, this entire process is $\Theta(V^2)$.

- Using various other data structures this can be brought down futher. For example with a min heap to store the edge weights and an adjacency list we can obtain $\Theta(E \lg V)$ and using a Fibonacci heap (which is a collection of min heaps) to store the edges weights and adjacenty list we can obtain $\Theta(E + V \lg V)$.

3 Kruskal's Algorithm

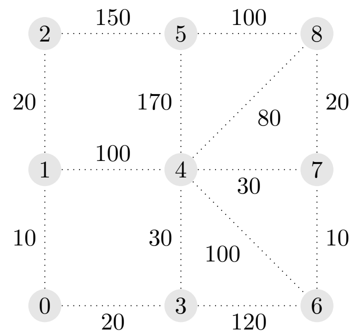
3.1 The Algorithm

Kruskal's Algorithm works as follows:

1. For now, exclude all edges.
2. Pick an excluded edge of minimum cost which does not form a cycle when included. Include it.
3. Repeat step 2 until we span the original graph.

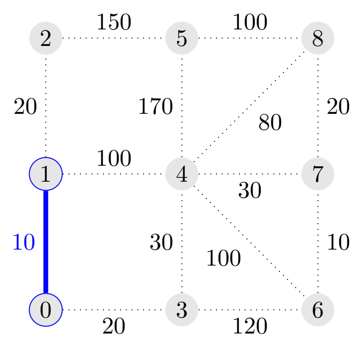
Note 3.1.1. We will eventually span the original graph because the graph spans itself. Moreover we can do this without adding a cycle because cycles are not necessary to achieve spanning.

Example 3.1. Consider the example from earlier. Here we have excluded all edges by dashing them:

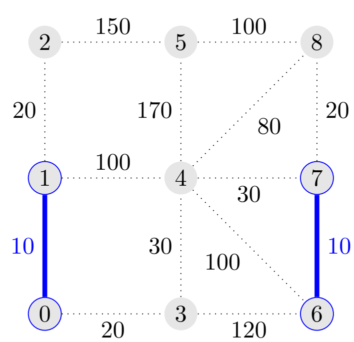


Since there are 9 vertices we'll need 8 edges so this will be an 8-step procedure.

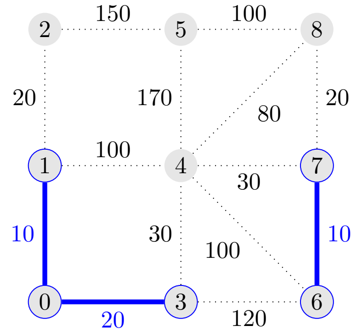
We add edge $0 - 1$ which does not create a cycle:



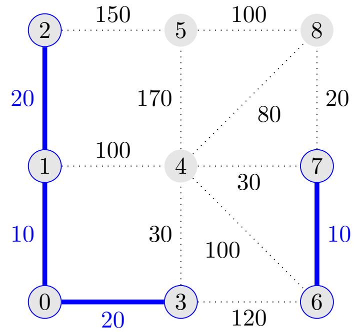
We add edge $6 - 7$ which does not create a cycle:



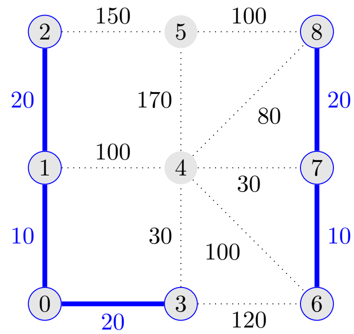
We add edge $0-3$ which does not create a cycle:



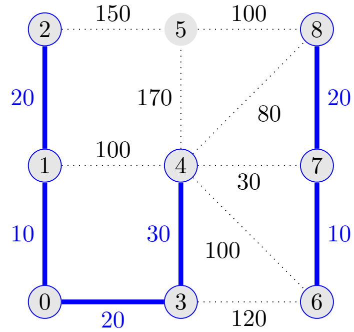
We add edge $1-2$ which does not create a cycle:



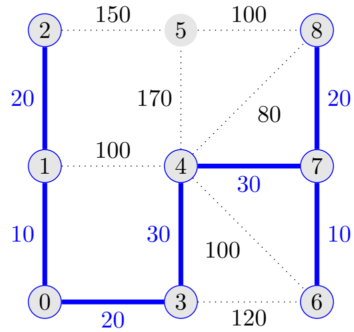
We add edge $7-8$ which does not create a cycle:



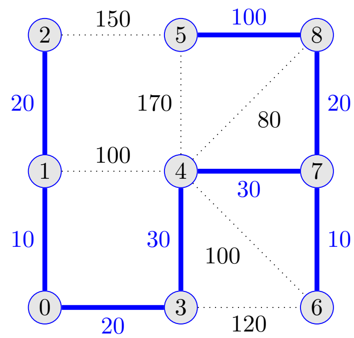
We add edge $3-4$ which does not create a cycle:



We add edge $4-7$ which does not create a cycle:



The next minimum cost edge is $4-8$ but adding it creates the cycle $4-8-7$ so we skip over it. and instead we add edge $5-8$ which does not create a cycle:



We now have a minimum cost spanning tree. The total cost is 240.

3.2 Mathematics for Kruskal's Algorithm

Theorem 3.2.1. Kruskal's Algorithm creates a MST.

Proof. The fact that we actually obtain a spanning tree follows from the fact that we proceed until we span the graph and from the fact that we do not add an edge if it would create a cycle, therefore we get a tree.

We now claim that at each iteration of the algorithm that if S is the set of edges we have included then S is contained in a MST.

If this is true then when we have obtained a spanning tree at the end this spanning tree is contained in a MST but since the only spanning tree it it contained in is itself, it must be that MST.

We proceed by structural induction, meaning we show that at the start S is contained in a MST and that if at any iteration this is true then it is still true after Kruskal adds another edge.

The statement is obviously true at the start because no edges are included and hence any MST will work.

Suppose the statement is true at some S and let $S \subseteq T$ where T is the MST. Kruskal's Algorithm adds some edge, $S + \{e\}$. We need to prove that there is a MST containing $S + \{e\}$.

Consider two cases:

- $e \in T$:

  Then $S + \{e\} \subset T$ and so T is a MST for $S + \{e\}$.

- $e \notin T$:

  Since T is a tree and we have added an edge, $T + \{e\}$ contains a cycle.

  Since $S + \{e\}$ does not contain a cycle (as this is how Kruskal works) there is some edge $f \in \text{cycle} \subseteq T + \{e\}$ with $f \notin S + \{e\}$. Observe now that $S + \{e\} \subseteq T + \{e\} - \{f\}$. We claim that $T + \{e\} - \{f\}$ is a MST for $S + \{e\}$.

  Since $e, f \notin S$ and Kruskal chose e and not f we have $w(e) \leq w(f)$. This tells us $w(T + \{e\} - \{f\}) \leq w(T)$.

  Observe that $T + \{e\} - \{f\}$ is also a spanning tree since we took the spanning tree T, added an edge to create a cycle and then removed an edge within that cycle, getting a tree back. Since T is minimal we also have $w(T + \{e\} - \{f\}) \geq w(T)$.

Together this tells us that $w(T + \{e\} - \{f\}) = w(T)$ and so $T + \{e\} - \{f\}$ is a MST containing $S + \{e\}$ as desired.

$\mathcal{QED}$

3.3 Graph Data, Pseudocode, Time Complexity

Loosely speaking the pseudocode is easy:

```
\\ PRE: G is a graph
T = empty graph
while vertices in T != vertices in G
    find (u,v) = minimum cost edge in G-T
    such that T+(u,v) does not form a cycle
        T = T + (u,v)
  end
end
\\ POST: T is a minimal spanning tree
```

Of course as before the devil is in the details. Keeping track of the edges in G-T is straightforward and selecting a minimum cost edge is not hard (this could be done with a simple linear search or with a binary heap) the significant challenge here is determining when $T + (u, v)$ does not contain a cycle.

- Fairly simple data structures can be used to achieve this and such an implementation can achieve a time complexity of $\mathcal{O}(E \lg E)$. This is equivalent to $\mathcal{O}(E \lg V)$ because $E \lg E = \mathcal{O}(E \lg V^2) = \mathcal{O}(2E \lg V) = \mathcal{O}(E \lg V)$ and $E \lg V = \mathcal{O}(E \lg(2E)) = \mathcal{O}(E \lg E)$.
- A more technical approach here is to use a *disjoint-set data structure* which is an data structure that handles sets well, and quickly, and allows such calculations with low time complexity. This is beyond the scope of this course, however such an implementation of Kruskal's Algorithm can be shown to run in $\mathcal{O}(E\alpha(V))$ time, where α is the inverse of the single-valued Ackermann function. Without diving too deeply into details this function α is "almost constant".

4 Prim v Kruskal

There are a few considerations that come into play when choosing between Prim and Kruskal.

1. If Prim ends early then we still have a tree.
2. Kruskal's selection process allows us to be much more flexible if we with to interact and make specific choices of edges. This is because we can choose any minimal weight edge which does not form a cycle, whereas in Prim we are restricted to growing the tree.
3. Prim allows us to select a starting vertex which gives us a different sense of control, especially when combined with the first point.
4. Prim tends to run faster in dense graphs (lots of edges). This is because even though there are more edges, Prim requires us to grow a tree which restricts the edges we can choose from. Especially earlier on in the algorithm Prim will have far fewer choices even in a very dense graph.
5. Kruskal tends to run faster in sparse graphs (few edges). This is because it's fairly quick to choose from a sparse set of edges.
6. Kruskal requires us to do cycle detection and this in and of itself can be challenging, or at least obscure.

5 Thoughts, Problems, Ideas

1. In Prim's Algorithm we need to check if our minimum-cost edge joins a vertex in T to a vertex in $G - T$. This is equivalent to determining if the edge creates a cycle. However in Kruskal's Algorithm these are not equivalent. Explain
2. In the Graphs chapter exercises you need to write the pseudocode to detect if a graph contains a cycle. Integrate that pseudocode into Kruskal's Algorithm and discuss the time complexity.
3. (a) If G has 5 nodes and 5 edges with weights $\{1, 2, 3, 4, 5\}$ and you construct a Kruskal minimal weight spanning tree, what are the minimum and maximum total possible weights? Explain and draw example graphs. Do not do this by attempting to list all graphs!

   (b) Repeat the previous question for 6 nodes and 6 edges with weights $\{1, 2, 3, 4, 5, 6\}$.
4. ...more to come...