Introduction and Background

Consider a very natural problem: we are given a set of locations \( V = \{v_1, v_2, ..., v_n\} \). We want to build a road system that connects these locations. Building a road between locations \((v_i, v_j)\) costs some amount of money, \( c(v_i, v_j) > 0\). Hence we want our road system to be as cheap as possible.

As the notation suggests, we can model this as a graph problem: we are given a set of vertices \( V = \{v_1, v_2, ..., v_n\}\), a set of edges \( E \), and a mapping \( w : E \rightarrow \mathbb{R}^+ \) from edges to positive real numbers (we will discuss applications with negative real numbers later). Here \( w \) is called the weight function. We assume the graph \( G = (V, E) \) is connected, otherwise one can apply the results of this section to each connected component separately.

Stated this way, our goal is to find a subset of edges \( T \subseteq E \) such that the graph \((V, T)\) is connected and the total cost, defined as \( w(T) = \sum_{e \in T} w(e) \), is as small as possible. We call such a graph \( T \) a minimum weight spanning subgraph:

**Definition** A spanning subgraph of \( G \) is a subgraph of \( G \) that contains all the vertices in \( G \).

A minimum weight spanning subgraph or minimum spanning subgraph is a spanning subgraph whose total cost is the minimum over all other spanning subgraphs.

**Note** We will often abuse notation and interchange \( T \) and the graph \((V, T)\). For example, we may say “consider the graph \( T \)” even though \( T \) is technically a set of edges. In most cases, there is no ambiguity.

When the edge weights are all positive, we have the following result:

**Proposition 1.** Let \( T \) be a minimum cost set of edges as described above. Then \((V, T)\) is a tree.

**Proof.** We know \( T \) is connected by how it is defined. Seeking contradiction, supposed \( T \) contained a cycle \( C \). Choose any edge \( e \) on this cycle, and remove it from \( T \). This forms a new graph with edge set \( T' = T - \{e\} \). Clearly this resulting graph is still connected, however it’s weight is \( w(T') = w(T) - w(e) < w(T) \) since all the edge weights were positive. This contradicts the assumption that \( T \) is the minimum cost set of edges that connect the graph. \( \square \)

Hence the graph \( T \) is in fact a minimum spanning tree (MST), and our goal will be to design an algorithm that efficiently computes the MST of a given graph.

Clearly the above proposition doesn’t hold if there are negative weight edges: the critical part of the proof above was that \( e \) had positive weight. On the other hand, if we allow some edges to have weight 0, then
even though there may be multiple minimum spanning subgraphs, there will always exist at least one MST. Why? Note that by the above proof, there cannot be any cycles in which every edge has positive weight. That is, every cycle has at least one edge with weight 0. Hence if you remove all such edges, you get rid of all the cycles, and the resulting graph has the same weight as the original. Moreover, it is a tree.

MST Algorithms

It turns out that many of the algorithms used to find MSTs are very simple greedy procedures. We will consider two of the more popular algorithms here.

Prim's Algorithm

Prim’s algorithm is very similar to Dijkstra’s shortest-paths algorithm. Let \( G = (V, E) \) be a connected graph. We choose any starting node \( s \in V \) and greedily grow a tree outward from \( s \) by simply adding the node that can be attached as cheaply as possible to the partial tree we already have.

More rigorously, we maintain a set \( T \), where initially \( T = \{s\} \). At each step, we consider all the vertices in \( V - T \) which have an edge to some vertex in \( T \). From these, we choose the one that has the minimum weight edge.

It is similar to Dijkstra’s algorithm in that we will be maintaining a priority queue to determine which vertex is the one that has the minimum weight edge to some vertex in \( T \). In fact, the pseudocode is identical to Dijkstra’s, the only difference being how we update the keys in the priority queue. The tree \( T \) in the pseudocode is represented by parent pointers, while the set \( V - T \) is represented by the vertices in the priority queue.

\[ P \text{rim}(G, s) \]

\[ \text{Let PQ be a priority queue containing every vertex in } G \]
\[ \text{for each } v \in V \text{ do} \]
\[ \quad v.\text{key} = \infty \]
\[ \quad \text{parent}[v] = \text{NIL} \]
\[ s.\text{key} = 0 \]

\[ \text{while PQ is not empty do} \]
\[ \quad v = \text{PQ.ExtractMin()} \]
\[ \quad \text{for each } u \in \text{Adj}[v] \text{ do} \]
\[ \quad \quad \text{if } w(u,v) < u.\text{key} \text{ then} \]

\[ \text{Prim’s Algorithm for MSTs} \]

\textit{Input:} A connected graph \( G = (V, E) \) given as an adjacency list, a weight function \( w : E \rightarrow \mathbb{R}^+ \), and a starting node \( s \).

\textit{Output:} An MST of \( G \).
March 19, 2019

Minimum Spanning Trees

Another way to phrase Prim’s algorithm is that it starts with \( T = \{ s \} \) and at each step, chooses the lightest weight edge crossing the cut \((T, V - T)\) (here we use \( T \) as a set of vertices) and adds it to the growing tree. \( V - T \) represents all the vertices still in the priority queue. This may be easier to visualize, as in the figure below:

![Figure 1: Prim’s Algorithm in the middle of execution. The red lines denote the tree \( T \) that is being built. The next step in the algorithm will be to choose the lightest edge crossing the \((T, V - T)\) cut and add it to \( T \). Here, that means choosing the lightest weight dotted line. In this picture, vertices 10 and 11 have keys equal to \( \infty \) in the priority queue.

We will postpone a proof of correctness for Prim’s algorithm until later. The runtime is easily seen to be \( O(m \lg n) \) by analogy to Dijkstra’s algorithm.

**Kruskal’s Algorithm**

Kruskal’s algorithm takes a different approach: it begins with a graph \( T \) that has no edges. Then it iterates through the edges in increasing order of weight. For each edge \( e \), if adding \( e \) to \( T \) doesn’t create a cycle, then \( T = T \cup \{ e \} \), otherwise you discard \( e \) and move on to the next edge. At the end of the algorithm, \( T \) will be an MST.

In order to determine if adding an edge \((u, v)\) creates a cycle, we will need to use the Union Find (UF) data structure, which has the following methods:
- \textbf{MakeSet}(x): creates a set with the single element $x$
- \textbf{Find}(x): returns the ID of the set to which $x$ belongs
- \textbf{Union}(x,y): combines the set containing $x$ and the set containing $y$

More details are provided in the Union Find notes, which also discusses the runtimes of the above functions.

Using UF, we can implement Kruskal’s algorithm as follows (note how we don’t need to specify a starting vertex like we do in Prim’s algorithm):

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Kruskal’s Algorithm for MSTs}
\Statex \textbf{Input:} A connected graph $G = (V,E)$ given as an adjacency list, a weight function $w : E \rightarrow \mathbb{R}^+$. 
\Statex \textbf{Output:} An MST of $G$.
\State \textbf{Kruskal}(G)
\State \hspace{1em} $T = \emptyset$
\State \hspace{1em} Sort the edges in $E$ in increasing order of weight.
\For {each $v \in V$}
\State \hspace{2em} \textbf{MakeSet}(v)
\EndFor
\For {each $e = (u,v) \in E$}
\If {$\text{Find}(u) \neq \text{Find}(v)$}
\State \hspace{2em} $T = T \cup \{e\}$
\State \hspace{2em} \textbf{Union}(u,v)
\EndIf
\EndFor
\State \Return $T$
\end{algorithmic}
\end{algorithm}

Sorting the edges takes $O(m \log m) = O(m \log n)$ time, and as the Union Find notes prove, this $O(m \log n)$ term in fact dominates the runtime of all the UF operations combined. Hence Kruskal’s also runs in $O(m \log n)$ time, just like Prim’s (and Dijkstra’s).

\section*{Comparing Prim’s and Kruskal’s}

Prim’s algorithm and Kruskal’s algorithm both take different approaches to finding an MST. Prim’s algorithm starts with a source node $s$, then progressively grows an MST outwards from $s$. On the other hand, Kruskal’s algorithm starts with a bunch of different trees (initially just single nodes), and combines these trees by adding edges until there is just one single tree.

In fact, there is another algorithm for finding MSTs called “Reverse Delete” which is kind of like a backwards-Kruskal. It starts with the full graph $G = (V,E)$, then iterates through the edges in \textit{decreasing} order of weight, deleting edges as long as they don’t disconnect the graph.

It may be surprising that there are so many efficient greedy algorithms for solving the MST problem. The proof of correctness for both Prim’s and Kruskal’s will shed some light on why this is the case.
Correctness of Prim’s and Kruskal’s

Since both of these algorithms work by repeatedly inserting edges from a partial solution, it will be useful to characterize when an edge is “safe” to include in the MST. We will also provide a characterization of edges that are guaranteed not to be in any MST. In this analysis we will assume that all the edge weights of the input graph $G = (V, E)$ are distinct—you should think about how to argue that this assumption is WLOG.

We have the following two important results:

**Proposition 2. (Cut Property)** Let $G = (V, E)$ be a connected, undirected graph. Let $S$ be any subset of nodes that is neither empty nor equal to all of $V$. Let $e = (u, v)$ be the minimum cost edge with one end in $S$ and the other in $V - S$. Then every MST of $G$ contains $e$.

**Proof.** Let $T$ be a spanning tree that does not contain $e$. We need to show that $T$ does not have the minimum possible cost. To do this, we will use an exchange argument: we need to find an edge $e'$ in $T$ that is more expensive than $e$ such that $T' = T - \{e'\} \cup \{e\}$ is a tree with lower total weight than $T$.

Since $T$ is a spanning tree, there must be a path $P$ from $u$ to $v$ in $T$. Since one end of $e$ is in $S$ and the other is in $V - S$, the path $P$ must cross the $(S, V - S)$ cut at some point. Let $e' = (u', v')$ be the edge on $P$ that crosses this cut.

Now consider $T' = T - \{e\} \cup \{e'\}$. $T'$ is a spanning tree: since $T$ was a spanning tree, any path in $T$ that went through $e'$ can be “re-routed” in $T'$ through the edge $e$. Also, $T'$ is acyclic because the only cycle in $T' \cup \{e'\}$ is the one composed of $e$ and the path $P$, and this cycle is not present in $T'$ due to the deletion of $e'$.

Now, since $e$ was the lightest edge crossing the $(S, V - S)$ cut, we must have $w(e) < w(e')$. Hence the weight of $T'$ must be strictly less than the weight of $T$, since $T'$ and $T$ are the same except for the edges $e$ and $e'$.

Note the choice of the edge $e'$ is important—you need to make sure that $T'$ is in fact a spanning tree. If you just choose any edge in $T$ that crosses the $(S, V - S)$ cut, such as the edge $f$ in the figure below, then it is not guaranteed that $T' = T - \{f\} \cup \{e\}$ is a spanning tree.
While the Cut Property is all that is needed to prove the correctness of Prim’s and Kruskal’s algorithm, we include the following important result since it is useful for analyzing properties of MSTs (and can also be used to prove the correctness of the Reverse-Delete algorithm):

**Proposition 3. (Cycle Property)** Let \( G = (V, E) \) be a connected, undirected graph. Let \( C \) be any cycle in \( G \) and let \( e = (u, v) \) be the heaviest edge in \( C \). Then \( e \) does not belong to any MST of \( G \).

**Proof.** Let \( T \) be a spanning tree that contains \( e \). We will show \( T \) doesn’t have the minimum possible weight. First, delete \( e \) from \( T \); this partitions the nodes into two components, \( S \) (which contains \( u \)) and \( V - S \) (which contains \( v \)). Consider the cycle \( C \). \( C - \{e\} \) is just a path from \( u \) to \( v \) and hence must cross the cut \( (S, V - S) \) at some point. Let \( e' \) be the edge in \( C - \{e\} \) such that \( e' \) has one endpoint in \( S \) and the other in \( V - S \). Then \( T' = T - \{e\} \cup \{e'\} \) is a spanning tree of \( G \) (the argument is similar to the one in the proof of the cut property) and since \( e \) was the heaviest edge on \( C \), we know \( w(e) > w(e') \) and thus the weight of \( T' \) is strictly less than the weight of \( T \). Hence \( T \) doesn’t have the lowest possible weight, completing the proof.

With the cut property, the correctness of Prim’s and Kruskal’s are straightforward.
Prim’s Algorithm: Correctness

In each iteration of Prim’s, there is a set $S \subseteq V$ on which a partial spanning tree has been constructed, and a node $v$ and edge $e$ are added to minimize $\min_{e=(u,v), u \in S} w(e)$. Hence by definition, $e$ is the lightest edge crossing the $(S, V-S)$ cut so by the cut property, $e$ is in every MST. Since Prim’s outputs a spanning tree such that every edge in the spanning tree is contained in an MST, Prim’s outputs an MST.

Kruskal’s Algorithm: Correctness

Consider any edge $e = (u, v)$ added by Kruskal’s algorithm, and let $S$ be the set of all nodes to which $u$ has a path at the moment just before $e$ is added. Clearly $u \in S$ but $v \notin S$ since adding $e$ doesn’t create a cycle. Moreover, no edge from $S$ to $V-S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have already been added by Kruskal’s algorithm. Thus $e$ is the lightest edge crossing the $(S, V-S)$ cut, and so it belongs to every MST. Since by construction, Kruskal’s algorithm outputs a spanning tree, it follows this output is actually an MST.