15.1 Introduction

Today we’re talking about Minimum Spanning Trees (MSTs). For this problem we are given an undirected, connected graph \( G = (V, E) \) and edge weights \( w : E \to \mathbb{R}_{\geq 0} \). A spanning tree is a set of edges \( T \subseteq E \) such that \( T \) is a tree on \( V \) (the graph \( (V, T) \) is connected and acyclic). Note that the connectivity requirement implies that every node is incident on at least one edge of \( T \), since otherwise there would be a connected component consisting of just one node. Our goal is to find a spanning tree \( T \subseteq E \) with minimum total weight \( w(T) = \sum_{e \in T} w(e) \). In other words, if \( T \) is the set of all spanning trees, we want to find a spanning tree \( T \in T \) such that \( w(T) = \min_{T' \in T} w(T') \).

This is a classic, fundamental, and important problem in graph algorithms. It has a number of obvious applications, e.g. building a network of minimum cost that allows every pair of nodes to communicate. But it is also a surprisingly useful primitive, which comes up in a variety of somewhat surprising places. As we’ll see, it’s also one of the simplest examples of some standard proof techniques and algorithmic techniques. In particular, it is one of the most famous places where greedy algorithms work well.

Today we’ll discuss the two most famous MST algorithms, as well as some extensions. Both of them are greedy algorithms, but they work in slightly different ways.

15.2 Generic Greedy

Before we talk about particular algorithms, let’s give a generic greedy algorithm. Both of the algorithms we’ll talk about are special cases of this generic algorithm. First, we need a definition.

Definition 15.2.1 Suppose that \( A \) is a subset of some minimum spanning tree. If \( A \cup \{\{u, v\}\} \) is also a subset of a minimum spanning tree, then we say that \( \{u, v\} \) is safe for \( A \).

Now consider the following algorithm, which we’ll call Generic-MST:

\[
A \leftarrow \emptyset; \\
\text{while } (A \text{ is not a spanning tree}) \{ \\
\quad \text{find an edge } \{u, v\} \text{ that is safe for } A; \\
\quad A \leftarrow A \cup \{\{u, v\}\} \\
\} \\
\text{return } A;
\]

It is obvious that this algorithm returns an MST, but this can be formally proved by induction.

Theorem 15.2.2 Generic-MST returns an MST.

Proof: Let’s prove that \( A \) is always a subset of an MST. This is true initially since \( A = \emptyset \). Now suppose it is true after the \( i \)th iteration of the while loop. Then by the definition of a safe edge,
it is true after the \((i + 1)\)st iteration of the while loop. Hence \(A\) is always a subset of an MST, so when Generic-MST returns \(A\) it is an MST (since it is not a spanning tree which is a subset of an MST, and is thus an MST).

Of course, the tricky thing is figuring out how to find a safe edge. The first thing to do is make a simple observation about the structure of trees (minimum or otherwise).

**Lemma 15.2.3** Let \((V, T)\) be a spanning tree, and let \(u, v \in V\). Let \(P\) denote the path between \(u\) and \(v\) in \(T\). If \(\{u, v\} \not\in T\), then \(T' = T \cup \{\{u, v\}\} \setminus \{\{x, y\}\} \) is a spanning tree for any \(\{x, y\} \in P\).

**Proof:** We need to prove that \(T'\) is connected and acyclic. Clearly \(T \cup \{\{u, v\}\}\) has exactly one cycle, consisting of \(P\) together with \(\{u, v\}\). Removing \(\{x, y\}\) breaks this cycle. Hence \(T'\) is acyclic. To see that it is connected, note that removing \(\{x, y\}\) breaks the tree into exactly two connected components, and that \(u\) is in one component and \(v\) is in the other. Hence adding \(\{u, v\}\) connects these two components, so \(T'\) is connected. Thus \(T'\) is a spanning tree.

Now let’s use this to analyze safe edges. Intuitively, we need some way of certifying than an edge is part of an MST. It turns out that one way of doing this is to analyze cuts. We’ll talk a lot more about cuts next week, but today is the beginning.

A cut \((S, V \setminus S)\) is just a partition of \(V\) into two pieces. Sometimes we’ll abbreviate this notation to \((S, \tilde{S})\) or just to \(S\) if the context is clear. Note that \(S\) does not have to be connected, small, or anything else – any partition of the vertices into two pieces is a cut. An edge \(e = \{u, v\}\) crosses the cut \((S, \tilde{S})\) if one endpoint of \(e\) is in \(S\) and the other is not in \(S\). Given a set of edges \(A\), a cut \((S, \tilde{S})\) respects \(A\) if no edge in \(A\) crosses \((S, \tilde{S})\). Finally, we say that an edge \(e\) is a light edge crossing a cut if it crosses the cut and has the minimum weight of any edge crossing the cut. Note that since weights are not necessarily distinct, there can be many light edges for each cut.

Now the following theorem will allow us to find safe edges.

**Theorem 15.2.4** Let \(A \subseteq E\) be a subset of some MST, let \((S, \tilde{S})\) be a cut respecting \(A\), and let \(e = \{u, v\}\) be a light edge crossing \((S, \tilde{S})\). Then \(e\) is safe for \(A\).

**Proof:** Let \(T\) be an MST that includes \(A\). If \(T\) includes \(e\) then we are done. Otherwise, consider the path \(P\) in \(T\) between \(u\) and \(v\). Since \(\{u, v\}\) crosses \((S, \tilde{S})\), we know that \(u\) and \(v\) are on different sides of the cut. Hence there is some edge \(e' = \{x, y\}\) in \(P\) where \(x \in S\) and \(y \in \tilde{S}\).

Consider \(T' = T \cup \{e\} \setminus \{e'\}\). We claim that \(T'\) is an MST containing \(A \cup \{e\}\), which would imply that \(\{u, v\}\) is safe for \(A\) and hence would prove the theorem. First, **Lemma 15.2.3** implies that \(T'\) is still a spanning tree. Since \((S, \tilde{S})\) respects \(A\), we know that \(e' \not\in A\) and hence \(T'\) contains \(A \cup \{e\}\).

So \(T'\) is a spanning tree containing \(A \cup \{e\}\) and hence we just need to show that \(T'\) is a minimum spanning tree. Since \(e\) is a light edge for \((S, \tilde{S})\) and \(e'\) crosses \((S, \tilde{S})\), we know that \(w(e) \leq w(e')\). Thus \(w(T') = w(T) - w(e') + w(e) \leq w(T)\). Since \(T\) is an MST, this implies that \(T'\) is also an MST (and in fact that \(w(e') = w(e)\)).

### 15.3 Prim’s Algorithm

The first true algorithm that we’ll talk about is **Prim’s Algorithm**. In Prim’s algorithm we start with a node and grow an MST out of it. Because of this approach, the algorithm actually looks a lot.
like Dijkstra’s shortest-path algorithm, but instead of computing a shortest-path tree it computes an MST.

We start with an arbitrary node $u$ and include it in $S$. We then find the lowest-weight edge incident on $u$, and add this to $A$. We then repeat, always adding the minimum-weight edge that has exactly one endpoint in $S$ to $T$. In other words, we grow a tree out of $u$, by always adding the edge of minimum weight with exactly one endpoint already in the tree. Slightly more formally, Prim’s algorithm is the following:

$$A ← ∅$$

Let $u$ be an arbitrary node, and let $S ← \{u\}$;

while ($A$ is not a spanning tree) {
    Find an edge $\{u,v\}$ with $u ∈ S$ and $v ∉ S$ that is light for $(S, S)$;
    $A ← A ∪ \{\{u,v\}\}$;
    $S ← S ∪ \{v\}$;
}

return $A$;

As always, we’ll want to do two things: prove correctness and running time. The running time is pretty obvious—it’s just the same as Dijkstra. That is, we can maintain a heap of all of the edges with at least one endpoint in $T$, and in each iteration we do Extract-Mins until we see an edge that has one endpoint in $T$ and one endpoint not in $T$. So this takes time $O(m \log n)$ using a binary heap, or $O(m + n \log n)$ using a Fibonacci heap. It’s actually even faster (though not asymptotically) than Dijkstra, since we never need to do any Decrease-Key operations.

Correctness is more interesting, but is straightforward using the results of the previous section.

**Theorem 15.3.1** Prim’s algorithm correctly computes an MST.

**Proof:** Using Theorem [15.2.2](#), we just need to prove that the edge that Prim’s algorithm adds is always safe for $A$, since that will imply that Prim’s is just a particular instantiation of Generic-MST. But this is obvious from Theorem [15.2.4](#). We just need to prove that $(S, S)$ always respects $A$, which is obvious since $A$ is always a connected component with vertices $S$.

### 15.4 Kruskal’s Algorithm

The next algorithm we’ll consider is a bit different, despite also being a greedy algorithm (and an instantiation of Generic-MST). Prim’s algorithm maintains a graph which is always a tree, and stops when it is spanning. Kruskal’s algorithm does not maintain a tree, but instead maintains a forest (a collection of trees), and stops when the forest becomes a single tree (note though that some “trees” in the forest might just be single nodes). So it is in some ways the opposite of Prim’s algorithm, and is a more “pure” greedy algorithm.

Kruskal’s algorithm is the following:

$$A ← ∅$$

Sort the edges by their weight (from smallest to largest);

foreach edge $e$ in this order {
If $A \cup \{e\}$ has no cycles, add $e$ to $A$;
}
return $A$;

Let’s start by proving correctness.

**Theorem 15.4.1** Kruskal’s algorithm correctly computes an MST.

**Proof:** We argue that this is a particular instantiation of Generic-MST. In particular, we claim that every edge added is safe for the current $A$, and that when the algorithm returns it is in fact a spanning tree. This suffices to prove that the set $A$ it returns is an MST.

We first show that the final set $A$ returned is a spanning tree. To see this, note that by construction it has no cycles, so it is a forest. To see that it is a tree, suppose for contradiction that it is not, i.e. $A$ has more than one connected component. Since $G$ is connected, there is some edge $\{u, v\} \in E \setminus A$ such that $u$ is in one connected component and $v$ is in a different connected component. Thus when Kruskal’s algorithm considers $\{u, v\}$, adding it would not cause a cycle and so $\{u, v\}$ would be added to $A$. This is a contradiction to the definition of $\{u, v\}$.

Now we need to show that if at some iteration Kruskal’s adds $e$ to $A$, then $e$ is safe for $A$. This would imply that Kruskal’s is just an instantiation of Generic-MST, and so would imply correctness by Theorem 15.2.2. To prove this, consider some point in the algorithm where we add edge $\{u, v\}$ to $A$. Since we add it to $A$, we know that this doesn’t create any cycles, and hence $u$ and $v$ are in two different connected components of $(V, A)$. Suppose that $u$ is in component $C$ and $v$ is in component $C'$. Consider the cut $(C, \bar{C})$. This cut clearly respects $A$, and since Kruskal’s algorithm looks at the edges in sorted order we know that $\{u, v\}$ is a light edge for $(C, \bar{C})$. Thus Theorem 15.2.4 implies that $\{u, v\}$ is safe for $A$.

What about the running time? Sorting takes $O(m \log m) = O(m \log n)$. As we’ll see, this actually turns out to be the dominant term. Once the edges are sorted, $m$ times, we have to check whether inserting an edge would cause a cycle to be created. In other words, we have to check whether the two endpoints of an edge are in the same connected component. Instead of a heap like we used for Prim, the important data structure here is Union-Find! We’ll have a set for each tree in $F$, when we add an edge we’ll need to do a Union, and when we test whether to add an edge we’ll have to do two Finds. So we need to do $n$ initial Make-Set, $2m$ Finds, and $n - 1$ Unions. If we use the simple list-based data structure, this gives running time of $O(m + n \log n)$. If we use the fancier tree-based data structures with path compression and union by rank, we get running time of $O(m \cdot \alpha(n))$, where $\alpha(n)$ is the inverse-Ackermann function applied to $n$. This is an improvement if $m$ is extremely sparse, but for large $m$ the simple list-based data structure actually performs better since there are so many more Finds. But either way, the running time is dominated by the cost of sorting.