Lecture Notes for CSCI 311: Algorithms
Set 17-Minimum Spanning Trees

Professor Talmage

November 18, 2024

---

1 Overview

Given an arbitrary graph, it can be difficult or expensive to conclude anything immediately, since the connectivity can be quite complex, and if $|E| = \Theta(|V|^2)$, there are a lot of edges to work through. Trees are in many ways much simpler (for example, there is only one path between any pair of nodes in a tree, and you cannot get stuck in a cycle). We want to build a tree that is in some way representative of an entire graph, but is of known, small size. For our purposes here, we will consider undirected graphs, as connectivity in directed graphs is more complex and difficult to keep track of.

Recall the following facts about trees:

Definition 1. A *tree* is a connected, acyclic graph.

Theorem 1. *An undirected graph $G = (V, E)$ is a tree if any of the following equivalent statements is true:*

1. *For any $u, v \in V$, there exists a unique simple path $u \rightsquigarrow v$.*
2. *G is connected, but removing any one edge disconnects G.*
3. *G is connected and $|E| = |V| - 1$.*
4. *G is acyclic and $|E| = |V| - 1$.*

We will not prove this theorem here, instead relying on your previous mathematical background. What this theorem gives us is four different ways to detect whether a graph is a tree, then once we have detected a tree, we can use any of these properties.

Definition 2. Given a graph $G = (V, E)$, $T \subseteq E$ is a *spanning tree* of G if (V, T) is a tree.

This is our representative tree for graph G. A spanning tree connects all the vertices of a graph using the fewest edges possible. One oddity about this definition is that a tree is not strictly a set of edges, but since the set of vertices remains fixed, specifying the edges we collect is sufficient.

Exercise: How may edges does a spanning tree have?

Every tree has one fewer edges than vertices, so all of the potentially many spanning trees of a graph have $|V| - 1$ edges. Which makes the question of a *minimum* spanning tree moot. For this to be interesting, we need to consider weighted graphs.

Definition 3. A graph is *weighted* if there is a function $w : E \to \mathbb{R}$. For an edge e, $w(e)$ is e's *weight*. We typically specify weighted graphs as $G = (V, E, w)$.

We can now say that a spanning tree is minimum if the sum of the weights of its edges is minimum among all spanning trees and formally state our problem:

MST Problem:

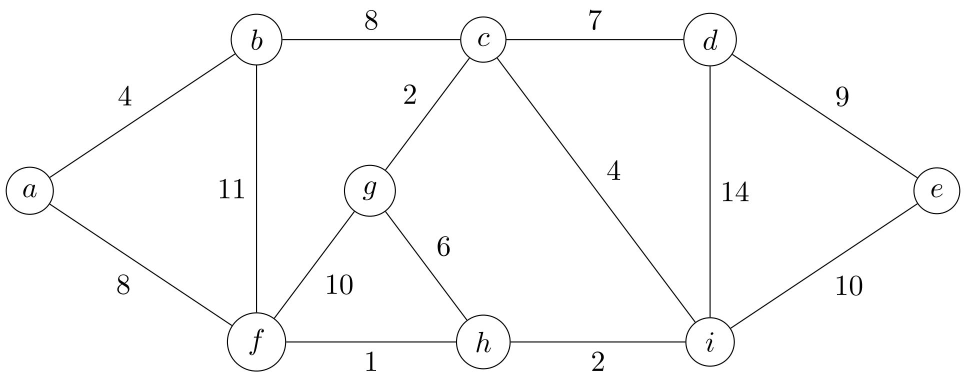
Input: Connected, weighted, undirected graph $G = (V, E, w)$.

Output: A minimum spanning tree (MST) of G. Alternately, $T \subseteq E$ s.t. (V, T) is an MST of G.

It is worth noting here that if edge weights are unique, then a graph's MST is unique, but otherwise it may not be.

2 Algorithms

Running Example:



Exercise: Give three different spanning trees of this graph and calculate each one's weight. Do you think any of these are minimum?

2.1 Generic Algorithm

To develop different solutions for the MST problem, we will start with a generic algorithmic idea, then consider different ways we might implement it.

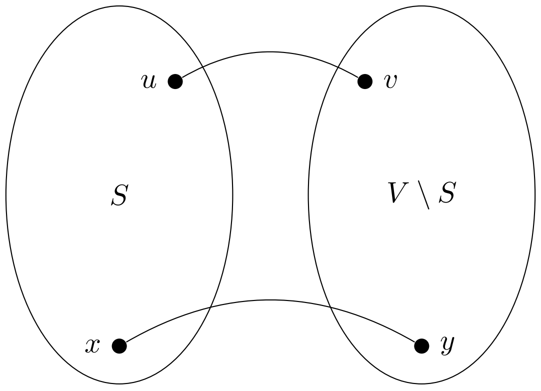
Algorithm 1 Generic algorithmic idea for finding an MST.

```
1: function GENERIC-MST($G = (V, E, w)$)
2:     $T = \emptyset$
3:     while not done ($|T| < |V| - 1$) do
4:         find a safe edge $e \in E$
5:         add $e$ to $T$
6:     end while
7:     return $T$
8: end function
```

The question, of course, is what is a "safe edge"?

Exercise: What properties must a safe edge have?

To be safe, an edge must not add a cycle, and must not make the tree we are building be greater than minimum weight. To find such an edge (and be confident that we have), suppose we partition V into two (disjoint) sets, S and $V \setminus S$. This is called a *cut* of the graph (technically, S is the cut, but specifying S specifies $V \setminus S$). Choose the the lightest edge crossing the cut. First, we observe that an MST must connect S to $V \setminus S$, as it must be connected. Second, by contradiction, if we took an edge across a cut that was not the lightest, we could replace it with the lightest edge to still have a spanning tree, but of lower weight. Finally, while T can have more than one edge crossing a cut, if it does, we could have a cycle, since T connects all nodes in S and all nodes in $V \setminus S$. Of course, if some nodes in S are only connected by crossing and recrossing the cut, this does not necessarily imply a cycle, but we want to be careful adding multiple edges across a cut. We will start by only adding edges across cuts which we have not already crossed.



Lemma 1. *Suppose (u, v) is the lightest edge crossing some cut S, with $w(u, v) = t$. If some partial MST P contains only edges $A \subseteq S$, then $A \cup \{(u, v)\}$ is part of some MST T extending P.*

Exercise: What kind of claim is this?

This is a greedy choice property, so our proof will follow the outline we learned for proving a GCP.

Proof. If $(u, v) \in T$, then we are done, so assume that there is no T containing (u, v). Then we can alter T to get another MST containing (u, v). Since T connects all vertices in the graph, there must be at least one edge on the unique simple path $u \rightsquigarrow v$ in T which crosses the cut S. Call one such edge (x, y). Consider $T' = T \setminus \{(x, y)\} \cup \{(u, v)\}$. First, $|T'| = |V| - 1$, since T was a spanning tree and we removed and added one edge each. Second, $\textit{weight}(T') \leq \textit{weight}(T)$, since we defined (u, v) as the lightest edge crossing S, so $w(u, v) \leq w(x, y)$. Third, removing an edge disconnects a tree, so $T \setminus \{(x, y)\}$ has no path from u to v. Thus, when we add (u, v), we do not create a cycle. Any acyclic graph with $|V| - 1$ edges is a tree, so we conclude that T' is a tree, and thus a spanning tree.

(x, y) is not in the partial MST A, since A is a subset of S and (x, y) crosses cut S, so T' is an MST containing $A \cup \{(u, v)\}$. □

Thus, to find a safe edge, we can define a cut S containing all edges we have added so far and choose the lightest edge crossing that cut.

Exercise: Return to the working example and find an MST using this method.

2.2 Prim's Algorithm

The direct implementation of this algorithmic idea, where at each step we define S as the set of vertices connected so far, is called *Prim's Algorithm.* We can start at any node, as an MST connects all nodes, and greedily choose the lightest edge leaving our partial MST in each step.

Complexity: The primary cost of Prim's algorithm is determining the lightest edge leaving the partial MST in each step. Naively, we could search all neighbors, which could be $O(|E|)$ per iteration, leading to a runtime of $O(|V||E|)$. To find our next edge more efficiently, we can maintain at each node its current (known) distance to the connected portion. These distances are all initially ∞, and when we add a node to the partial tree, we update the distances of all of that node's neighbors to the weight of the edge between them. We can store nodes in a min-Priority Queue keyed on those distances, so after updating our new node's neighbors, the next node we add will be that the priority queue returns.

An efficient priority queue implementation (such as a heap) has all operations (`insert`, `update`, `extractMin`) run in $O(\log n)$, so our total runtime is $O((|V| + |E|) \log |V|)$, since it takes $|V| \log |V|$ to insert all nodes once and extract each once, plus $2|E| \log |V|$ to update the heap each time we update a distance (we touch each edge once from each end).

Exercise: Run Prim's algorithm on our running example, starting from a.

2.3 Kruskal's Algorithm

Another way to implement our generic MST algorithm finds an efficient way to try all possible cuts and grab the lightest edge crossing any of them, by not apparently considering cuts at all. Instead, the idea is to add the lightest edge in the entire graph at each step. We must simply check at each step that the lightest edge does not create a cycle, and if it does, discard that edge and continue with the next lightest edge.

Algorithm 2 Kruskal's MST algorithm

```
1: function KruskalMST($G = (V, E, w)$)
2:     Sort $E$ in order of increasing weight
3:     $T = \emptyset$
4:     for all $e \in E$ (in sorted order) do
5:         if adding $e$ would not create a cycle then
6:             add $e$ to $T$
7:         end if
8:     end for
9:     return $T$
10: end function
```

Complexity: Sorting the edges takes $O(|E| \log |E|)$ time. It then takes $O(|E|)$ times the cost of checking whether an edge is safe. But this we can do with a disjoint set structure, since the sets of nodes connected in partial trees must be disjoint. Recall that a tree-based implementation of disjoint sets using Union by Rank and Path Compression has $O(\log^* |V|)$ amortized time per operation. Because we are using many calls to `Union` and `FindSet`, amortized cost is appropriate, and gives us a total cost for Kruskal's algorithm of $O(|E| \log |V| + |E| \log^* |V|) = O(|E| \log |V|)$.

Exercise: How did we reduce $O(|E| \log |E|) = O(|E| \log |V|)$?

Correctness: This is still a greedy algorithm, so we need to prove the greedy choice property. Let e_i be the first edge our algorithm chooses which is **not** in some optimal solution (assuming for the sake of contradiction that our algorithm is incorrect). Adding e_i to an MST creates a cycle, so delete some edge in that cycle which is not in the set Kruskal's algorithm returns. The weight cannot have increased, as we are choosing the lightest edges in the graph, and we assumed e_i is the lightest edge the algorithm chooses which is not in the optimal solution. The result is connected, since we took an edge out of a cycle, and has $|V| - 1$ edges, so is a tree. Thus, Kruskal's algorithm returns an MST.

Exercise: Run Kruskal's algorithm on our running example.