CS 3510 Algorithms Last updated: September 24, 2023

Lecture 9: Minimum Spanning Tree

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian*

1 Minimum Spanning Tree

We'll use the remaining time to start the next section: Minimum Spanning Tree. Essentially in this problem you have a graph $G = (V, E), w : E \to \mathcal{R}$. These graphs are undirected and connected. The goal is to find a subset $T \subset E$ s.t. Tremains connected & $\sum_{e \in T} w(e)$ is minimized. The tree is spanning because the graph remains connected and it is minimum because it has the smallest total weight.

We'll be able to use variant of depth first search to solve this, and it will give us an understanding of greedy algorithms.

Before we get started, we should note we could never have an MST which is a cycle. If there was a cycle, you could takek the largest edge, remove it and have a spanning tree which is cheaper. Another thing worth noting before getting into the problem is that if all the edge weights were zero, we could simply return any spanning tree, so the weights make the problem interesting.

1.1 Cut Lemma

The Cut Lemma is a powerful tool and will allow us to develop algorithms to solve the MST problem. Imagine you have some cut in the graph between a set of vertices S and a set of vertices $V|S$, the cheapest edge must be part of some minimum spanning tree. [1] We'll prove this lemma by contradiction, so assume there exists a cut $(S, V|S)$ s.t. the cheapest cross edge does not belong to the minimum spanning tree. Say the edge e', is currently a part of the minimum spanning tree, but the cheapest edge to cross the cut is e. We can remove the edge e' and add e, and we will create a cheaper spanning tree because we know by assumption that $w(e) < w(e')$. Now we have created a contradiction because the new spanning tree is smaller than the minimum spanning tree.

An edge case was brought up "what if the cut only has one edge crossing it?" Yes our proof does not work, but we can simply create another contradiction: that edge must be part of the MST for the tree to be spanning.

[1] From now on, we will assume all edge weights are distinct. If edges are not distinct, it's possible to have multiple MSTs which is a little harder to argue about.

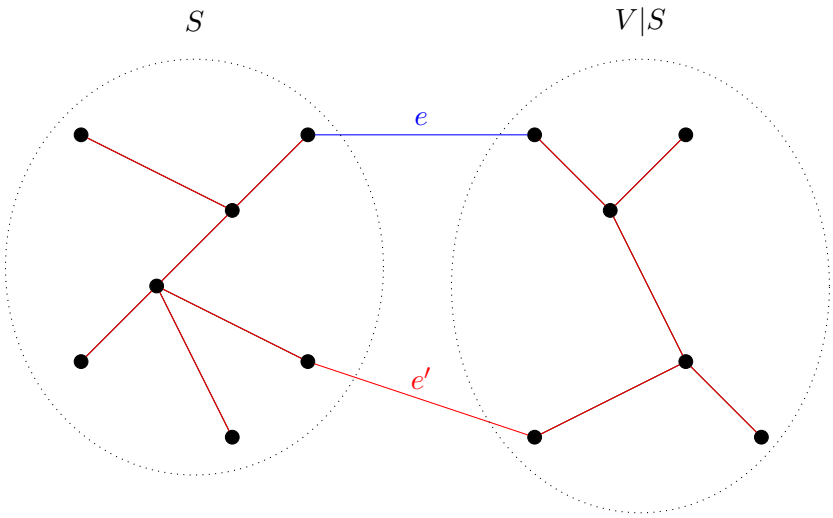

Figure 1: An MST crossing a cut with edge e', and another edge crossing the cut e.

1.2 Prim's Algorithm

We'll use the Cut Lemma to develop an algorithm, let's say we have some set of vertices we'll call R. We know the smallest edge leaving R will be the smallest edge crossing the cut $(R, V|R)$, and we can add that vertex to the MST, and the node it connects to, to the region R. We can start at an arbitrary vertex called root and add it to R. If we grow R to be V, the edges we have will the entire MST.

Algorithm 1 Prim's Algorithm

```
1:  procedure PRIM'S(G, r)
2:      Q ← all edges of r and their weights
3:      T ← ∅
4:      R = {r}
5:      while Q ≠ ∅ do
6:          (u, v) ← Q.remove
7:          if v ∉ R then
8:              T ← T ∪ (u, v)
9:              A ← R ∪ v
10:             Q.add(edges adjacent to v)
11:         end if
12:     end while
13: end procedure
```

Above is the algorithm, we have two sets T to maintain the edges of the tree and R to maintain the vertices in our region we've explored. We maintain a priority queue Q with edges ordered by their weights. When we remove a edge from the queue we check if we've already seen the node it connects to and if not we add it to our tree (since it must be the smallest), and we add the node it connects to, to the region of vertices we've seen. We also

add all of it's outgoing edges to the priority queue.

The correctness can be done using induction with the cut lemma which you can work out on your own. The running time of this algorithm is $\Theta(|E|)$ insertions and deletions of queue, which can be $\mathcal{O}(|E| \log |V|)$.

2 Greedy Algorithms

Prim's algorithm is a great example of a greedy algorithms which make the best choices at each step and hope that the overall result is also the best. Essentially there are the following steps: you have some choices at each step (e.g. edges), you assign some score or priority to the options, you select the best option, and then take another step. These algorithms don't always work, but they are pretty nice when they do work; you can think about this strategy like classes of algorithms like divide and conquer of dynamic programming.

2.1 Kruskal's Algorithm

Another instance of a greedy algorithm is Kruskal's algorithm, we will sort the edges by weight. Start with a tree $T = \varnothing$. If the next edge e does not create a cycle when adding to T, $T = T \cup e$. At each time step you could think of this algorithm as having a forest of trees which may not be connected.

The naive runtime for this would be $\mathcal{O}(|E| \log |E| + \text{detecting cycles})$. You can actually make detecting cycles cheap enough that the time is negligible compared to the sorting. Then the total runtime would be $\mathcal{O}(|E| \log |E|)$. [2]

The correctness of this algorithm is similar to Prim's: induction by steps and the Cut Lemma. Essentially at each step, you have some forest and you have the cheapest edge that does not create a cycle, which means the edge is between two trees in the forest. We can add this edge because of the Cut Lemma.

This greedy algorithm works even for matroids. We're not going to discuss matriods in this class, but they are an interesting class of object which relates to the minimum spanning tree.

2.2 Scheduling Problem

Given a machine and n jobs where each job has some time it will take and the priority or weight. How can we schedule these jobs? As you can imagine, there are different metrics we can try to minimize; we'll try to minimize total completion time. The completion time can be thought of as the time people are in the shop as well as the time to process their job. The total completion time is equal to

$$\sum_{i=1}^{n} \left[\left(\sum_{j:\sigma(j)<\sigma(i)} s_j \right) + s_i \right] w_i$$

[2] If you're interested in a data structure which helps with this, you can refer to the book. You may have learned it in your data structures class.