CS 3510 Algorithms 9/20/2023

Lecture 7: Kruskal's Algorithm for Minimum Spanning Trees

Lecturer: Abrahim Ladha *Scribe(s): Aditya Kumaran*

1 Greedy Algorithms

We're going to talk about a class of algorithms which is very intuitive for people: when you have an optimization problem, pick the local best, and then solve the rest of the problem without that piece. The obvious issue though is that picking the local maximums might not lead to a global maximum.

As an example, say you're trying to pack the most amount of luggage into a given amount of space. Say you have suitcases of size 40, 40, and 70, but you only have 100 units of space. If you picked the largest piece of luggage, you'd pick 70, even though you could pick the two 40s to get 80 units of luggage packed.

This is an unlucky case, and is very close to a more interesting problem we'll look at later. However it's not the rule - in fact, in many situations, Greedy Algorithm's work well. Two examples are Huffman Coding for compression and Kruskal's algorithm for computing minimum spanning trees. These algorithms and most other greedy algorithms are simple, but that ease comes with the challenge of proving that greed will lead to the best outcome.

2 Minimum Spanning Tree

What is the minimum spanning tree (MST) problem though? To put it succinctly, given a weighted graph, give a graph which is still connected but with the smallest weights. An MST is a set of edges such that it is

- Minimum, the sum of the edge weights is less than or equal to any other MST
- Spanning, each $v \in V$ is an endpoint of atleast one edge in the MST
- Tree, there are exactly $|V| - 1$ edges and no cycle.

This problem has many real world applications, in fact this algorithm and many other interesting graph problems came out of Bell Labs. If you think about laying out phone lines between towns, the cost of the phone lines, could be the distance between the towns, and you want to spend the least wire possible (and therefore, money) while connecting them.

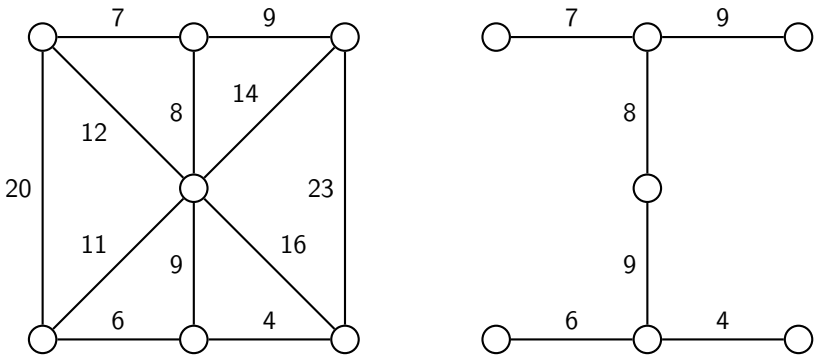

Figure 1: A weighted graph and it's MST.

Of course, in that scenario, you might want more connections for redundancy but, for simply finding the minimum spanning tree, we don't care. Although this exercise reveals a couple interesting properties about minimum spanning trees. There will be no cycles in the resulting MST because if there were a cycle, we could remove the highest value edge and get a smaller spanning tree. Also as the problem requires, the graph must be as small as possible while being connected, so it will have $|V| - 1$ edges.

Trees are graphs with special properties, and come up often in computer science. They have many equivalent characterizations. Trees are acyclic, have $n - 1$ edges and n nodes, and for any two vertices u, v in the tree, there exists a unique path from u to v. (If there were two or more paths, compose them to make a cycle and now its not a tree anymore).

Formally, given a weighted undirected graph G, a minimum spanning tree (MST) is a set of (n-1) edges such that:

1. **All vertices $\in$ V are at one end-point**
2. **The sum of the edges is the smallest of any other spanning such tree**

3 Kruskal's Algorithm

Since it's so simple, we'll present Kruskal's algorithm now, and then prove it's correctness later. Kruskal's algorithm is rather simple and what you might come up with by thinking about this problem: **at each step, add the smallest edge to a set which does not form a cycle with edges within that set**.

Of course, checking for cycles is easier said than done; although we humans can do it quickly, the best algorithm we've discussed talks linear time, meaning this would take quadratic time (good, but not great).

```
def kruskals(G, w):
    for all v in V:
        makeset(v)

    X = {}
    sort E by weight
    for all (u, v) in E
        if find (u) is not find(v):
            x = x + {(u, v)}
            union (find(u), find(v))
```

Figure 2: Explore routine on a graph and node.

To check for cycles, this algorithm relies on a unique data structure: union find. Now, this data structure isn't terribly important for the class, but it's very important for understanding this algorithm. It operates on sets, and does two things: set union, and finding the set an element is within. Both operations are logarithmic. [1]

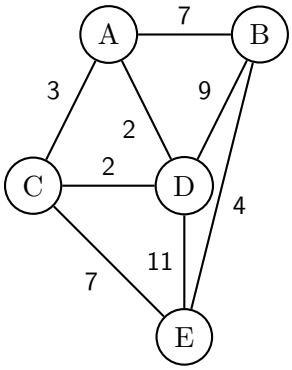

Figure 3: A simple weighted graph.

Let's walk through how this algorithm works. If we sort the edges in this graph we get $[2, 2, 3, 4, 7, 7, 9, 11]$. As we iterate over this list we get the following results.

[1] The proof of these runtimes is in the book. It's fairly interesting, but not important for today. If you're interested in learning more refer to 5.1.4.

	{A}, {B}, {C}, {D}, {E}
2	{A, D}, {B}, {C}, {E}
2	{A, D, C}, {B}, {E}
3	find(A) equals find(C)
4	{A, D, C}, {B, E}
7	find(A) equals find(C)
7	{A, D, C, B, E}
9	find(B) equals find(D)
11	find(D) equals find(E)

Figure 4: The sets created by Kruskal's algorithm while iterating over edges.

Now let's analyze the runtime of this algorithm. Making the sets for union find will take $\mathcal{O}(|V|)$ time. Sorting E by weight will be $\mathcal{O}(|E| \log |E|)$. There will be $|E|$ iterations over the sorted edges, each doing $\log |V|$. Adding these together we'll get $\mathcal{O}(|V| + |E| \log |E| + |E| \log |V|)$. Note that we know $O(\log |E|) = O(\log |V|)$, so the overall runtime is $\mathcal{O}(|E| \log |E|)$.

4 Proof and The Cut Property

Even though we have an intuitive algorithm, we need to show it is correct. How would we do this? Contradiction is one good option: we can assume a greedy approach does not give the optimal solution and show why the greedy approach would have chosen this other solution instead, creating a contradiction. In this proof, we'll use this form with induction layered on top.

Say we have some edges already chosen X which a part of some MST T ($X \subset T$). We want to show that the next e chosen by the algorithm will also be part of an MST ($X + e \subset T'$). [2] It's important to note here that T is some MST, but no where in our proof do we try to show that T will be a certain MST.

[2] There can be multiple MSTs in a graph; if you're not sure why this would be look at one of the graphs above, with the change that all edges are of equal weight.

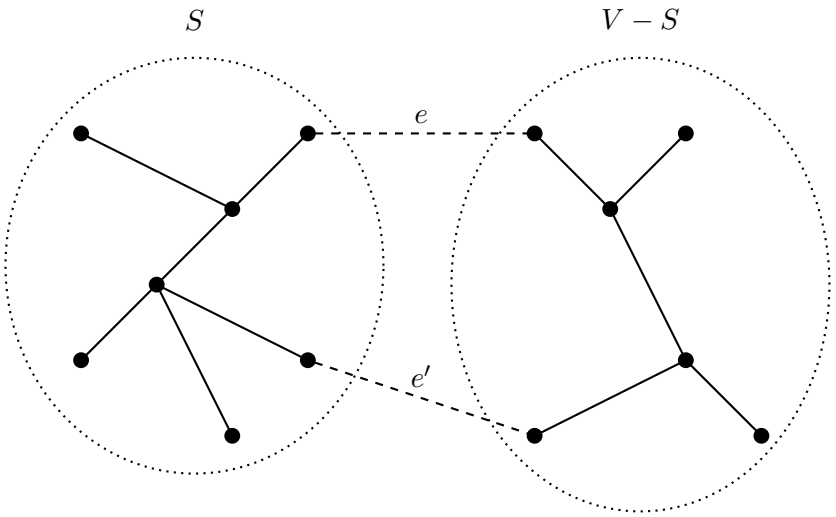

Figure 5: A cut creating two parts of a graph with edges e and e' going through the cut.

An important of this proof is the Cut Property which states that if you have have a set of edges X, a set of vertices S such that no edge in X crosses S to $V - S$, and e be the lightest edge across the partition, then e will be part of an MST. Intuitively we know that one edge across the cut must be part of all spanning trees. If a spanning tree consisted of none of the edges of a cut, it would not be spanning. However why must the lightest edge be part of some MST? Say edges X are part of some MST T; if e is also part of T there is nothing to prove. However say e is not part of T, we'll build a new MST T'.

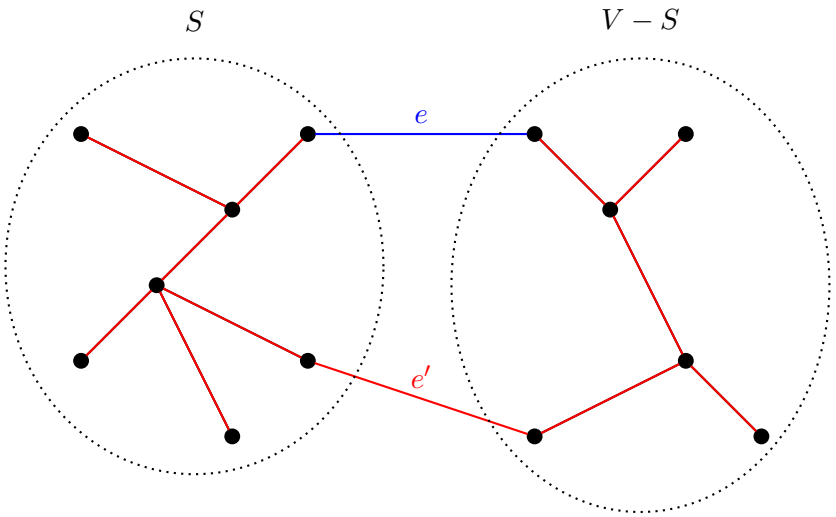

Figure 6: An MST crossing a cut with edge e', and another edge crossing the cut e.

Say we have the possible MST, T in red above, and we add edge e to this MST; we have now formed a cycle. This cycle has some other edge across the cut e', and if we remove this edge, we get the MST $T' = X + e - e'$. T' is also a tree because it both spans the graph

and has $|V| - 1$ edges. However we must also show it is the minimum spanning tree as we will do.

We can calculate the weight of the new tree T', as the weight of the old tree T minus the removed edge e' plus the additional edge e: $w(T') = w(T) - w(e') + w(e)$. We know that e is the lightest edge by definition, meaning $w(e) \leq w(e')$. Then we know that $w(T') \leq w(T)$, but since T is an MST, $w(T)$ is minimal. It must be the case that they are of equal weight. [3]

Now the justification of Kruskal's algorithm is rather simple. At each step, we have a partial solution X and the lightest edge e which hasn't yet been considered. We check that e does not form a cycle, implying that it would be the lightest edge in some cut because it is the first edge considered crossing said cut. Now, we know that it satisfies the cut property, making the algorithm find an MST.

[3] Notice here what must be true about the ordering of $w(e')$ and $w(e)$ if $w(T) = w(T')$. Consider how this might effect the number of MSTs, and how the number of MSTs might change if all edge weights were unique.