CS 3510 — Algorithms, Spring 2022

Lecture 15: Minimum Spanning Trees

March 15, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

1 Minimum Spanning Tree

A minimum spanning tree is the topic of today's lecture. First, we can define each part independently. A tree is a graph that is connected and acylic. Spanning means every vertex of some graph G is spanned by this, and minimum means the sum of the edges in this tree is minimal (compared to all other possible trees).

Theorem 1.1. *T is a tree on n vertices if and only if there are $n-1$ edges*

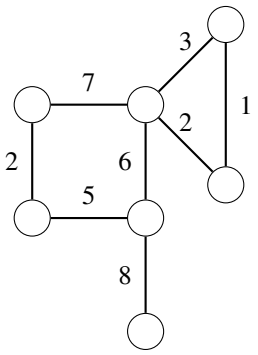
Proof. Strong induction both ways (bidirectional). Proof not included.

Before we can begin discussing algorithms to build minimum spanning trees, we first need to discuss greedy algorithms. A greedy algorithm is an algorithm that builds up a solution by taking the step that immediately looks best.

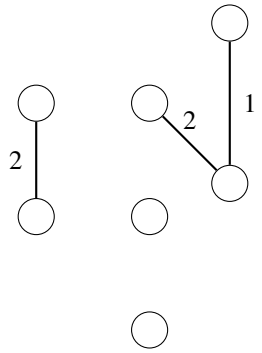
1.1 Kruskal's Algorithm

Kruskal algorithm essentially takes the lightest edge that does not cause a cycle and keeps doing that until there are $n-1$ edges (which means this would be a spanning tree, the claim is that this is minimal).

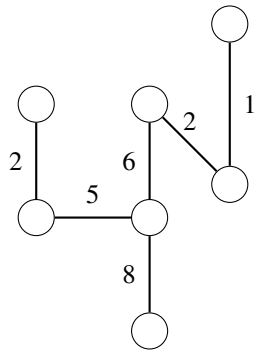
Let's run this algorithm by hand first, see the following example:



In the above, we keep taking the smallest edge such that the edge does not cause a cycle, hence, we first take edge with weight 1, then 2, then the other 2.



Now, we notice that the next edge to take is 3, but this is not possible because this would create a cycle, hence we skip this. Next, we add the edge with weight 5, then 6, then consider 7, but adding 7 would cause a cycle, hence we do not take the 7 edge.



Now, we have $|V| - 1$ edges and therefore have a minimal spanning tree. Let's see how to implement Kruskal:

```
1 Function Kruskal(G = (V, E), weights):
2     sort(E by weights)
3     X ← {} for e ∈ E do
4         if e does not add cycle then
              /* add edge e to X                    */
5             X ← add(X, e)
```

This algorithm makes a pretty large assumption that we know how to determine if e does not add a cycle, hence we need to specify how to do this.

1.2 Union Find Datastructure

Union-Find data structure is a data structure that has two pursposes. It has `find`(x) that returns the component x belongs to and `union`(x, y) joins the component of x and y into a single component. Each component is represented by the root of that component (where each component is saved as a tree).

At the beginning, each vertex is stored in the data structure as its own component. How do we know what the root is for a component? This is done by the vertex keeping track of its parent (it also keeps track of its rank, which will be used below). It can simply return the parent of its parent, and if that parent is null, then this is the root.

```
1 Function Find(x):
2     if x = parent(x) then
3         return x
4     return Find(parent(x))
```

For the `union` method, we need to make sure that this does not degenerate to a Linked List, so we keep track of the rank or depth of each vertex and ensure when merging or union-ing two components, we correctly optimize for the tree structure to be taken advantage of.

```
1 Function Union(x, y):
2     if Rank(x) < Rank(y) then
3         parent(Find(x)) = Find(y)
4     else if Rank(x) > Rank(y) then
5         parent(Find(y)) = Find(x)
6     else
7         parent(Find(y)) = Find(x)
8         Rank(x) ← Rank(x) + 1
```

What is the runtime of the Union-Find datastructure methods? The `Find` method takes roughly $\mathcal{O}(\log n)$ as the algorithm simply walks up the tree and the height has maximum of $\mathcal{O}(\log n)$ levels. The same is true for `Union`.

Now, we can implement Kruskal again but using this data structure for cycles:

```
1 Function Kruskal(G = (V, E), weights):
2     X ← {}
3     R ← init Union-Find(V)
4     sort(E by weights)
5     for (u, v) ∈ E do
6         if R.Find(u) ≠ R.Find(v) then
              /* add edge e to X                    */
7             X ← add(X, e)
8             R.Union(u, v)
```

The runtime of this algorithm is $\mathcal{O}((|V| + |E|) \log |V|) + \mathcal{O}(|E| \log |V|)$ because the sorting of the edge weights takes $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$. We also iterate through each edge and run `Find` $\mathcal{O}(|E|)$ time where `Find` takes $\mathcal{O}(|E| \log |V|)$ and we also run `Union` $\mathcal{O}(|V|)$ times. This all combines to the runtime given above.

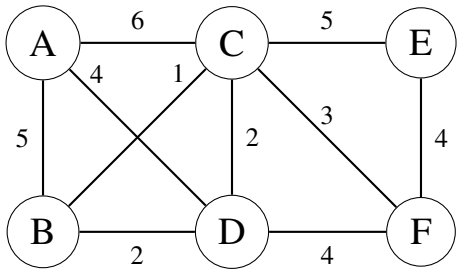
1.3 Prim's Algorithm

This algorithm is also a greedy algorithm, but rather than simply taking the lightest edge that doesn't cause a cycle, we take the lightest non-cycle causing edge that connects to the tree that we have built so far.

This algorithm may make more sense as we incrementally build up a single tree, rather than picking unrelated edges like in Kruskal.

We start at a random vertex and find all the vertices that are closest by. In essence, we are running Dijkstra with a different priority value. Let us work through an example (see next page).

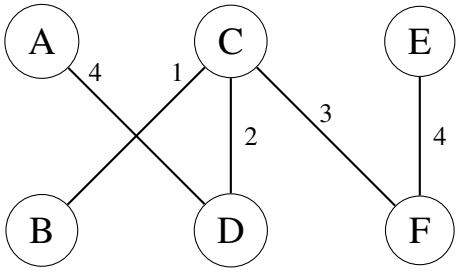
Using Prim, let us find the MST of the following graph:



First, we initialize a Priority Queue with ∞ for every vertex in the graph.

Iteration Number	Vertex in PQ					
	A	B	C	D	E	F
1	∞	∞	∞	∞	∞	∞
2	$0^\star$	5	6	4	∞	∞
3		2	2	$4^\star$	∞	4
4		$2^\star$	1		∞	4
5			$1^\star$		5	3
6					4	$3^\star$
7					$4^\star$	
8						

Through each iteration, we pop off the minimum priority element from the priority queue (labeled with $\star$) and update the keys for all the neighbors of that edge in the PQ if the edge from the current vertex to that vertex is smaller than the existing priority in the PQ. Hence, the MST is constructed with edges $2, 1, 4, 4$, and 3.



On the next page, we will see an implementation of Prim.

An implementation of Prim could look like:

```
1  Function Prim(G = (V, E), weights):
       /* set up priority queue with distance ∞ for all vertices        */
2      Q ← PriorityQueue(V)
3      root ← random vertex
4      DecKey(Q, root, 0)
5      while Q is not empty do
6          v ← PopMin(Q)
7          for (v, w) ∈ E do
8              alt ← weight[v, w]
9              if alt < Q[w] then
10                 Q[w] ← alt
11                 DecKey(seen, w, alt)
```

Since this algorithm is very similar to Dijkstra, it makes sense that the runtime of this algorithm is $\mathcal{O}((|V|+|E|)\log|V|)$ by a similar derivation to Dijkstra runtime.