Chris Gregg
CS106B

Section #8
May 28, 2018

Section Handout #8: More Graphs

1. Prim's Algorithm

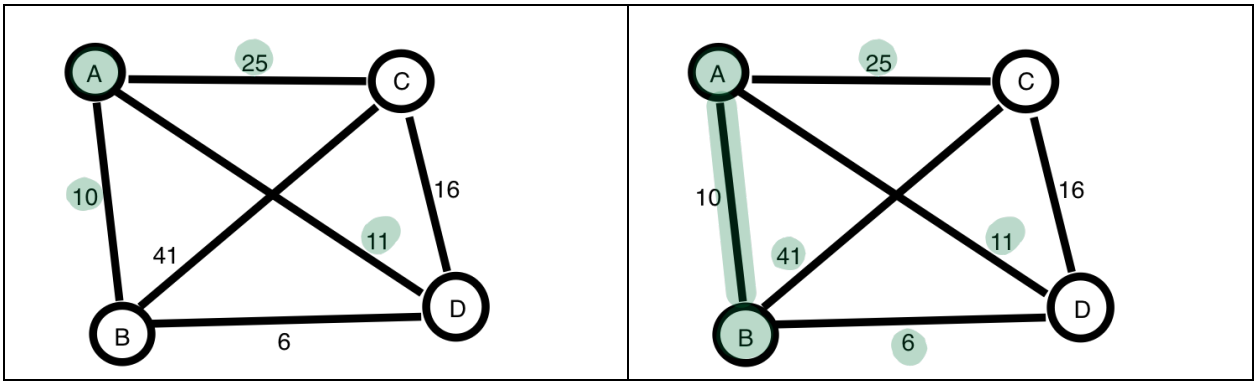
In lecture, we saw Kruskal's algorithm for constructing a minimum spanning tree (MST) of a graph, which is a subgraph that is a tree with the smallest total weight that connects all its vertices. There is also another well-known MST algorithm, called *Prim's Algorithm*, which goes as follows:

(1) **Start with a single vertex V from the original graph to the tree**
(2) **At each step, add the edge (and the vertex it goes to) with the lowest cost that has exactly one endpoint in our current tree**
(3) **Stop once we have all original vertices added to our tree**

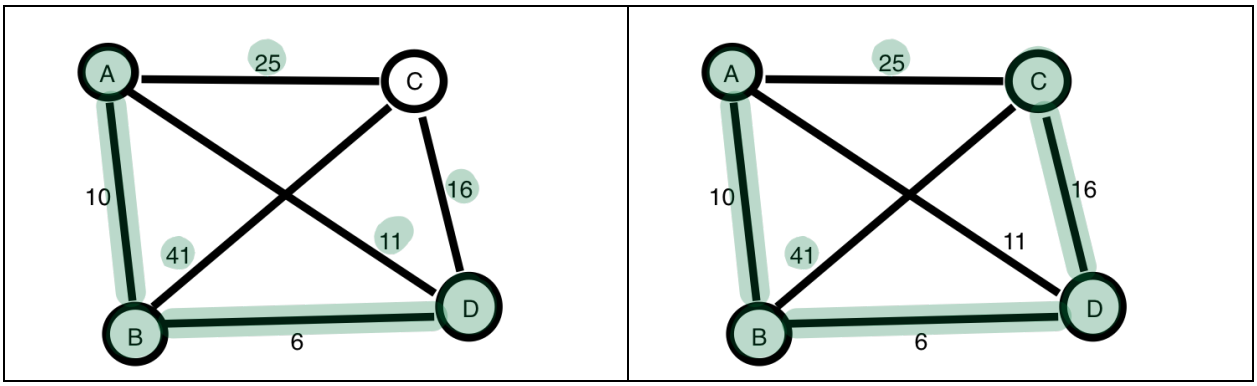
The specific approach we recommend is the following[1]:

- Pick an arbitrary starting node
- Create a priority queue to help sort edges
- At each step, add to the priority queue all edges from the *last vertex added* that have a second endpoint outside of the current group of nodes you have chosen
- Then, dequeue an edge – this edge, and the node it goes to outside of our current tree, should be added to our tree.

This process is outlined in the diagrams below, read from left to right, top to bottom. Note that if you dequeue an edge that *does not* go outside the current tree, ignore it. Enqueued edges have their costs highlighted, while chosen nodes and edges are fully highlighted.



[1] Thanks to https://algs4.cs.princeton.edu/lectures/43DemoPrim.pdf.



Unlike in Kruskal's algorithm, which centers around adding edges one by one to make a tree, Prim's algorithm adds vertices one at a time, starting with a starting vertex. In certain cases, variations of Prim's algorithm can run faster on certain types of graphs than Kruskal. It also means that if you stop Kruskal's algorithm early, you will end up with clusters of connected vertices, while if you stop Prim's algorithm early you will always have one connected tree (do you see why?)

Implement Prim's algorithm in the following function:

```
Set<Edge *> prim(BasicGraph &graph) { …
```

This function takes in a graph and returns the set of edges in that graph that make up an MST.

2. Clustering

MST algorithms are extremely useful for a variety of problems, including *clustering* related items into groups[2]. The goal is to divide **n** "items" (vertices) into **k** groups such that the minimum distance between items in different groups is maximized. One way to do this is the following:

- Maintain a set of connected components in the graph, starting out with each vertex as its own connected component
- Each time, combine the clusters with the two closest items
- Stop once we have exactly **k** clusters

This should seem familiar – this is a variation on Kruskal's MST algorithm! There, we kept track of sets of vertices, and added edges to our tree to combine these sets. Modify Kruskal's algorithm to write the following function that takes a graph and returns **k** sets of vertices that represent the k clusters that fit the criteria mentioned above.

```
Set<Set<string>> cluster(BasicGraph &graph, int k) { …
```

The code for Kruskal's algorithm from lecture is included below.

[2] Thanks to https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/mst.pdf

```
void kruskal(BasicGraph& graph, BasicGraph &mst) {
    // first, copy the graph and remove the edges
    // This is so we can populate it later with the mst edges
    mst = graph;
    mst.clearEdges();
    // put each vertex into a 'cluster', initially containing only itself
    Map<Vertex*, Set<Vertex*>* > clusters;
    Set<Vertex*> allVertices = graph.getVertexSet();
    Vector<Set<Vertex*>* > allSets;   // for freeing later
    for (Vertex* v : allVertices) {
        Set<Vertex*>* set = new Set<Vertex*>();
        set->add(v);
        clusters[v] = set;
        allSets.add(set);
    }

    // put all edges into a priority queue, sorted by weight
    PriorityQueue<Edge*> pq;
    Set<Edge*> allEdges = graph.getEdgeSet();
    for (Edge* edge : allEdges) {
        pq.enqueue(edge, edge->cost);
    }

    // repeatedly pull min-weight edge out of PQ and add it to MST if its
    // endpoints are not already connected
    Set<Edge*> mstEdges;
    while (!pq.isEmpty()) {
        Edge* e = pq.dequeue();
        Set<Vertex*>* set1 = clusters[e->start];
        Set<Vertex*>* set2 = clusters[e->finish];
        if (set1 != set2) {
            mstEdges.add(e);

            // merge the two sets
            set1->addAll(*set2);
            for (Vertex* v : *set1) {
                Set<Vertex*>* setv = clusters[v];
                if (setv != set1) {
                    clusters[v] = set1;
                }
            }
        }
    }

    for (Set<Vertex*>* set : allSets) {
        delete set;
    }

    // populate the graph with the edges
    // We can't add the edge pointers directly
    // because that would cause trouble freeing later
    for (Edge *edge : mstEdges) {
        mst.addEdge(edge->start->name,edge->end->name,edge->cost,false);
    }
}
```