Lecture 6: Minimum Spanning Trees and Prim's Algorithm

Read: Section 9.5 in Weiss.

Minimum Spanning Trees: A common problem in communications networks and circuit design is that of connecting together a set of nodes (communication sites or circuit components) by a network of minimal total length (where length is the sum of the lengths of connecting wires). We assume that the network is undirected. To minimize the length of the connecting network, it never pays to have any cycles (since we could break any cycle without destroying connectivity and decrease the total length). Since the resulting connection graph is connected, undirected, and acyclic, it is a *free tree.*

The computational problem is called the *minimum spanning tree* problem (MST for short). More formally, given a connected, undirected graph $G = (V, E)$, a *spanning tree* is an acyclic subset of edges $T \subseteq E$ that connects all the vertices together. Assuming that each edge (u, v) of G has a numeric weight or cost, $w(u, v)$, (may be zero or negative) we define the cost of a spanning tree T to be the sum of edges in the spanning tree

$$w(T) = \sum_{(u,v)\subset T} w(u, v).$$

A *minimum spanning tree* (MST) is a spanning tree of minimum weight. Note that the minimum spanning tree may not be unique, but it is true that if all the edge weights are distinct, then the MST will be distinct (this is a rather subtle fact, which we will not prove). The figure below shows three spanning trees for the same graph, where the shaded rectangles indicate the edges in the spanning tree. The one on the left is not a minimum spanning tree, and the other two are. (An interesting observation is that not only do the edges sum to the same value, but in fact the same set of edge weights appear in the two MST's. Is this a coincidence? We'll see later.)

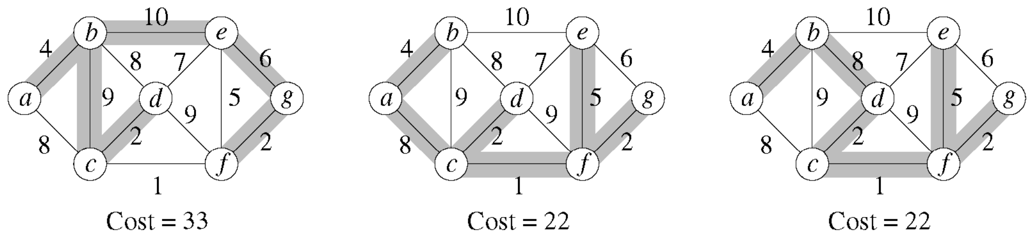

Figure 13: Spanning trees (the middle and right are minimum spanning trees.

Steiner Minimum Trees: Minimum spanning trees are actually mentioned in the U.S. legal code. The reason is that AT&T was a government supported monopoly at one time, and was responsible for handling all telephone connections. If a company wanted to connect a collection of installations by an private internal phone system, AT&T was required (by law) to connect them in the minimum cost manner, which is clearly a spanning tree ... or is it?

Some companies discovered that they could actually reduce their connection costs by opening a new bogus installation. Such an installation served no purpose other than to act as an intermediate point for connections. An example is shown in the figure below. On the left, consider four installations that that lie at the corners of a 1×1 square. Assume that all edge lengths are just Euclidean distances. It is easy to see that the cost of any MST for this

[1]Copyright, David M. Mount, 2001

configuration is 3 (as shown on the left). However, if you introduce a new installation at the center, whose distance to each of the other four points is $1/\sqrt{2}$. It is now possible to connect these five points with a total cost of of $4/\sqrt{2} = 2\sqrt{2} \approx 2.83$. This is better than the MST.

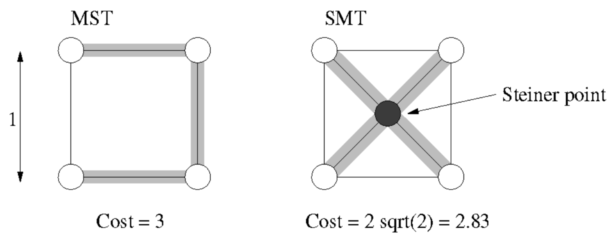

Figure 14: Steiner Minimum tree.

In general, the problem of determining the lowest cost interconnection tree between a given set of nodes, assuming that you are allowed additional nodes (called *Steiner points*) is called the *Steiner minimum tree* (or SMT for short). An interesting fact is that although there is a simple greedy algorithm for MST's (as we will see below), the SMT problem is much harder, and in fact is NP-hard. (By the way, the US Legal code is rather ambiguous on the point as to whether the phone company was required to use MST's or SMT's in making connections.)

Generic approach: We will present a *greedy algorithm* (called Prim's algorithm) for computing a minimum spanning tree. A *greedy algorithm* is one that builds a solution by repeated selecting the cheapest (or generally locally optimal choice) among all options at each stage. An important characteristic of greedy algorithms is that once they make a choice, they never "unmake" this choice. Before presenting these algorithms, let us review some basic facts about free trees. They are all quite easy to prove.

Lemma:

- A free tree with n vertices has exactly $n-1$ edges.
- There exists a unique path between any two vertices of a free tree.
- Adding any edge to a free tree creates a unique cycle. Breaking *any* edge on this cycle restores a free tree.

Let $G = (V, E)$ be an undirected, connected graph whose edges have numeric edge weights (which may be positive, negative or zero). The intuition behind Prim's algorithms is simple, we maintain a subtree A of the edges in the MST. Initially this set is empty, and we will add edges one at a time, until A equals the MST. We say that a subset $A \subseteq E$ is *viable* if A is a subset of edges in some MST (recall that it is not unique). We say that an edge $(u, v) \in E - A$ is *safe* if $A \cup \{(u, v)\}$ is viable. In other words, the choice (u, v) is a safe choice to add so that A can still be extended to form an MST. Note that if A is viable it cannot contain a cycle. Prim's algorithm operates by repeatedly adding a *safe* edge to the current spanning tree.

When is an edge safe? We consider the theoretical issues behind determining whether an edge is safe or not. Let S be a subset of the vertices $S \subseteq V$. A *cut* $(S, V - S)$ is just a partition of the vertices into two disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in $V - S$. Given a subset of edges A, we say that a cut *respects* A if no edge in A crosses the cut. It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST, and we wish to know which edges can be added that do *not* induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

An edge of E is a *light edge* crossing a cut, if among all edges crossing the cut, it has the minimum weight (the light edge may not be unique if there are duplicate edge weights). Intuition says that since all the edges that cross a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice. The main theorem which drives both algorithms is the following. It essentially says that we can always augment A by adding the minimum weight edge that crosses a cut which respects A. (It is stated in complete generality, so that it can be applied to both algorithms.)

MST Lemma: Let $G = (V, E)$ be a connected, undirected graph with real-valued weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V - S)$ be any cut that respects A, and let (u, v) be a light edge crossing this cut. Then the edge (u, v) is *safe* for A.

Proof: It will simplify the proof to assume that all the edge weights are distinct. Let T be any MST for G. If T contains (u, v) then we are done. Suppose that no MST contains (u, v). We will derive a contradiction.

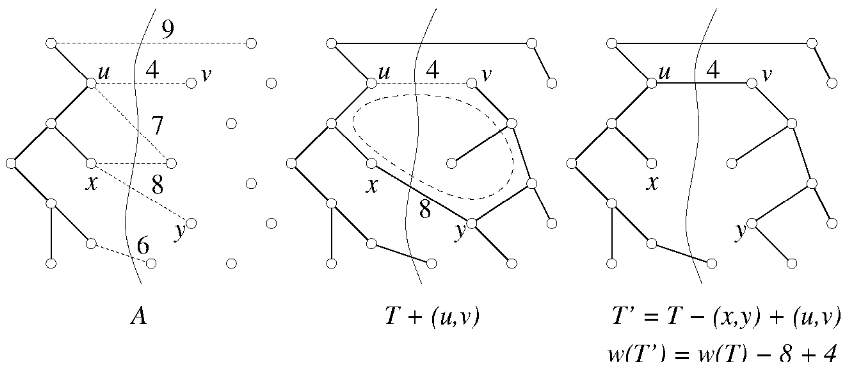

Figure 15: MST Lemma.

Add the edge (u, v) to T, thus creating a cycle. Since u and v are on opposite sides of the cut, and since any cycle must cross the cut an even number of times, there must be at least one other edge (x, y) in T that crosses the cut.

The edge (x, y) is not in A (because the cut respects A). By removing (x, y) we restore a spanning tree, call it T'. We have

$$w(T') = w(T) - w(x, y) + w(u, v).$$

Since (u, v) is lightest edge crossing the cut, we have $w(u, v) < w(x, y)$. Thus $w(T') < w(T)$. This contradicts the assumption that T was an MST.

Prim's Algorithm: There are two well-known greedy algorithms for computing MST's: Prim's algorithm and Kruskal's algorithm. We will discuss Prim's algorithm here. Prim's algorithm runs in $O((V + E) \log V)$ time. Prim's algorithm builds the tree up by adding leaves one at a time to the current tree. We start with a root vertex r (it can be *any* vertex). At any time, the subset of edges A forms a single tree. We look to add a single vertex as a leaf to the tree. The process is illustrated in the following figure.

Observe that if we consider the set of vertices S currently part of the tree, and its complement $(V - S)$, we have a cut of the graph and the current set of tree edges A respects this cut. Which edge should we add next? The MST Lemma from the previous lecture tells us that it

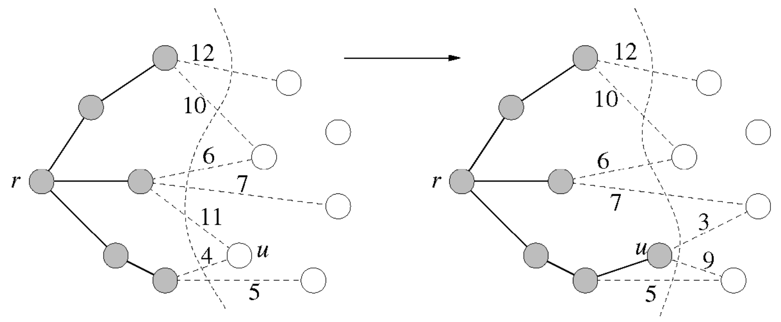

Figure 16: Prim's Algorithm.

is safe to add the *light edge.* In the figure, this is the edge of weight 4 going to vertex u. Then u is added to the vertices of S, and the cut changes. Note that some edges that crossed the cut before are no longer crossing it, and others that were not crossing the cut are.

It is easy to see, that the key questions in the efficient implementation of Prim's algorithm is how to update the cut efficiently, and how to determine the light edge quickly. To do this, we will make use of a *priority queue* data structure. Recall that this is the data structure used in HeapSort. This is a data structure that stores a set of items, where each item is associated with a *key* value. The priority queue supports three operations.

`ref = insert(u, key)`: Insert vertex u with the key value *key* in the priority queue. The operation returns a pointer `ref` to allow future access to this item.

`u = extractMin()`: Extract the vertex with the minimum key value in Q.

`decreaseKey(ref, newKey)`: Decrease the value of the entry with the given reference `ref`. The new key value is *newKey.*

A priority queue can be implemented using the same heap data structure used in heapsort. The above operations can be performed in $O(\log n)$ time, where n is the number of items in the heap. Some care needs to be taken in how references are handled. The various heap operations cause items in the heap to be moved around. But the references cannot be moved, because they are our only way of accessing heap nodes from the outside. This can be done by having the heap store pointers to the reference objects.

What do we store in the priority queue? At first you might think that we should store the edges that cross the cut, since this is what we are removing with each step of the algorithm. The problem is that when a vertex is moved from one side of the cut to the other, this results in a complicated sequence of updates.

There is a much more elegant solution, and this is what makes Prim's algorithm so nice. For each vertex in $u \in V - S$ (not part of the current spanning tree) we associate u with a key value `key[u]`, which is the weight of the lightest edge going from u to any vertex in S. We also store in `pred[u]` the end vertex of this edge in S. If there is not edge from u to a vertex in $V - S$, then we set its key value to $+\infty$. We will also need to know which vertices are in S and which are not. We do this by coloring the vertices in S black.

Here is Prim's algorithm. The root vertex r can be any vertex in V.

Prim's Algorithm

```
Prim(Graph G, Vertex r) {
        for each (u in V) {                     // initialization
            key[u] = +infinity;
            color[u] = white;
            pred[u] = null;
        }
        key[r] = 0;                             // start at root
        Q = new PriQueue(V);                    // put vertices in Q
        while (Q.nonEmpty()) {                  // until all vertices in MST
            u = Q.extractMin();                 // vertex with lightest edge
            for each (v in Adj[u]) {
                if ((color[v] == white) && (w(u,v) < key[v])) {
                    key[v] = w(u,v);            // new lighter edge out of v
                    Q.decreaseKey(v, key[v]);
                    pred[v] = u;
                }
            }
            color[u] = black;
        }
        [The pred pointers define the MST as an inverted tree rooted at r]
    }
```

The following figure illustrates Prim's algorithm. The arrows on edges indicate the predecessor pointers, and the numeric label in each vertex is the key value.

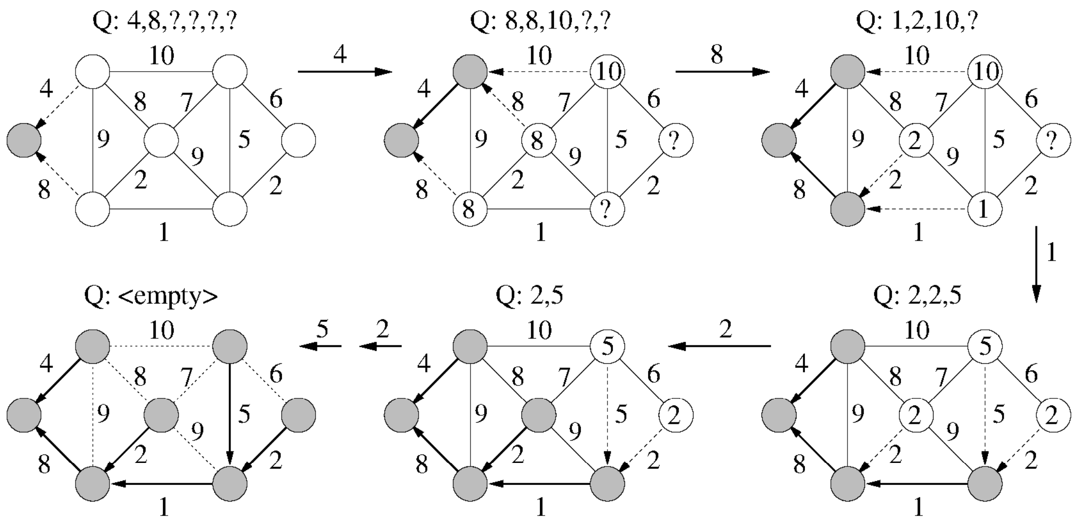

Figure 17: Prim's Algorithm.

To analyze Prim's algorithm, we account for the time spent on each vertex as it is extracted from the priority queue. It takes $O(\log V)$ to extract this vertex from the queue. For each incident edge, we spend potentially $O(\log V)$ time decreasing the key of the neighboring vertex. Thus the time is $O(\log V + deg(u) \log V)$ time. The other steps of the update are constant time. So the overall running time is

$$T(V,E) = \sum_{u\in V} (\log V + deg(u) \log V) = \sum_{u \in V} (1 + deg(u)) \log V$$

$$= \quad \log V \sum_{u \in V} (1 + deg(u)) \;=\; (\log V)(V + 2E) \;=\; \Theta((V+E)\log V).$$

Since G is connected, V is asymptotically no greater than E, so this is $\Theta(E \log V)$.