Illinois Institute of Technology
Department of Computer Science

Lecture 20: November 10, 2010

CS 430 Introduction to Algorithms
Fall Semester, 2010

1 Minimum Spanning Trees

Consider a connected, undirected graph $G = (V, E)$. For many applications it is useful to find a subgraph $T = (V, E')$ of G, where E' is the smallest collection of edges sufficient to make T a connected graph, i.e. T is a tree. Subgraphs such as T are called *spanning trees* of G. This problem can be made more general by assigning each edge of E a weight and finding the spanning tree whose edges have the smallest combined weight. Such a spanning tree is called a *minimum spanning tree*. It turns out that such trees can be found efficiently. We examine a number of algorithms for doing so.[1]

1.1 Prim's Algorithm

The first algorithm we examine for the minimum spanning tree problem is called Prim's algorithm. In this algorithm, the spanning tree is "grown" from a single initial vertex, called the *source vertex*. Each step of the algorithm involves adding a single edge and vertex to the existing tree. This is done in a greedy manner by adding the cheapest edge from the current tree to some "outside" vertex. This can be implemented efficiently by putting the vertices into a priority queue. The source vertex is initially assigned weight 0 and all other vertices are assigned infinite weight. These weights are an estimate of the distance from the vertex to the current tree. When a vertex u is removed from the queue all its edges are examined. For each outgoing edge which goes to a vertex not already in the tree, the cost of including that edge is compared to the current weight of that vertex. If the examined edge is cheaper, then the distance estimate is revised by calling DECREASE-KEY to adjust the weight of that vertex. If the vertices each remember the cheapest edge connecting them to the tree, then the edges which should be added to the tree can be easily determined as well.

Now we show that this algorithm produces a minimum spanning tree. As with other greedy algorithms, we do this by looking at the algorithm's first mistake. Let this first mistake occur when we add some vertex v to the tree T using an edge uv which is not in any optimal minimum spanning tree of the graph. Let T_{opt} be an optimal tree which includes the edges in T. There must be a path from u to v in this tree. Since u in inside T and v is not, at some point this path must cross from vertices in T to vertices not in T. Let wx be the first edge leaving the vertices of T along this path. Clearly removing wx from T_{opt} and replacing it with uv creates a spanning tree. Furthermore, since we choose to connect a vertex via the cheapest outgoing edge at each step, the edge wx must cost at least as much as uv. Thus, the new spanning tree we create must cost no more than T_{opt}, making it a minimum spanning tree. However, since $T + uv$ is a subset of this tree, we must not have made a mistake adding uv. Therefore, the algorithm produces a minimum spanning tree.

What is the running time of this algorithm? If we use a Fibonacci heap to implement the priority queue, the vertices can be added to the queue in $O(V)$ time, the vertices can be extracted in a total of $O(V \log V)$ time, and the keys can be decreased in a total of $O(E)$ time. Thus the entire algorithm runs in $O(E + V \log V)$ time.

[1] This problem is discussed in chapter 23 of CLRS, though it is given a slightly different treatment here.

1.2 Kruskal's Algorithm

Kruskal's algorithm is a different greedy approach to finding a minimum spanning tree. Rather than starting from a single source vertex as we do with Prim's algorithm, Kruskal's algorithm grows separate subtrees to grow until they are eventually connected to form a minimum spanning tree. Initially, each vertex is a trivial subtree. At each subsequent step, we add the cheapest edge which connects different subtrees.

First we show that this algorithm produces a minimum spanning tree, again by analyzing the first "mistake" made by the algorithm. Suppose that some set S of edges has already been correctly added before some edge uv not in any minimum spanning tree is added. Let T_{opt} be a minimum spanning tree which uses the edges of S. There must be a path between vertices u and v in T_{opt}. Since u and v were in separate components before that edge was added, there must be some first edge wx not in S along this path. Swap uv for this edge in T_{opt}. Since we choose edges in order of weight and have not chosen wx yet, the resulting spanning tree must also be minimal and therefore the algorithm did not make a mistake.

Now we consider the running time of Kruskal's algorithm. First the algorithm sorts the edges by weight, in $O(E \log E)$ time. In order to keep track of which vertices are already connected, we keep track of the components with a disjoint set data structure. Thus, for initialization we build $|V|$ new sets (calling the MAKE-SET operation on each vertex) in $O(V)$ time. For each edge (u, v) considered in increasing order, we determine if u and v are in the same tree by comparing FIND-SET(u) with FIND-SET(v). If u and v are indeed in different trees, we add the edge (u, v) and UNION the two trees. Since at worst we do 2 FIND-SET operations per edge, this loop takes at most $O(E\alpha(V)) \leq O(E \log V)$ time. Thus the full algorithm takes $O(E \log E)$ time. This is asymptotically more than the running time of Prim's algorithm, but Kruskal's may be more practical for small problem instances since it does not require a Fibonacci heap, which introduces some very large constants.