CS 6783 (Applied Algorithms) – Lecture 5

Antonina Kolokolova*

January 19, 2012

1 Minimum Spanning Trees

An *undirected graph* G is a pair (V, E); V is a set (of vertices or nodes); E is a set of (undirected) edges, where an edge is a set consisting of exactly two (distinct) vertices. For convenience, we will sometimes denote the edge between u and v by $[u, v]$, rather than by $\{u, v\}$.
The *degree* of a vertex v is the number of edges touching v. A *path* in G between v_1 and v_k is a sequence $v_1, v_2, \ldots, v_k$ such that each $\{v_i, v_{i+1}\} \in E$. G is *connected* if between every pair of distinct nodes there is a path. A *cycle* (or *simple cycle*) is a closed path $v_1, \ldots, v_k, v_1$ with $k \geq 3$, where $v_1, ..., v_k$ are all distinct. A graph is *acyclic* if it has no cycle. A *tree* is a connected acyclic graph. A *spanning tree* of a connected graph G is a subset $T \subseteq E$ of the edges such that (V, T) is a tree. (In other words, the edges in T must connect all nodes of G and contain no cycle.)

If a connected G has a cycle, then there is more than one spanning tree for G, and in general G may have exponentially many spanning trees, but each spanning tree has the same number of edges.

Lemma 1 *Every tree with n nodes has exactly $n - 1$ edges.*

The proof is by induction on n, using the fact that every (finite) tree has a *leaf* (i.e. a node of degree one).

We are interested in finding a minimum cost spanning tree for a given connected graph G, assuming that each edge e is assigned a *cost* $c(e)$. (Assume for now that the cost $c(e)$ is a nonnegative real number.) In this case, the cost $c(T)$ is defined to be the sum of the costs of the edges in T. We say that T is a *minimum cost spanning tree* (or an optimal spanning tree) for G if T is a spanning tree for G, and given any spanning tree T' for G, $c(T) \leq c(T')$.

Given a connected graph $G = (V, E)$ with n vertices and m edges $e_1, e_2, \ldots, e_m$, where $c(e_i) =$ "cost of edge e_i", we want to find a minimum cost spanning tree. It turns out (miraculously) that in this case, an obvious greedy algorithm (Kruskal's algorithm) always works. Kruskal's algorithm is the following: first, sort the edges in increasing (or rather nondecreasing) order of costs, so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$; then, starting with an initially empty tree T, go through the edges one at a time, putting an edge in T if it will not cause a cycle, but throwing the edge out if it would cause a cycle.

*This set of notes is based on the course notes of U. of Toronto CS 364 as taught by Stephen Cook

1.1 Kruskal's Algorithm:

Sort the edges so that: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \leftarrow \emptyset$
for $i : 1..m$
(*) if $T \cup \{e_i\}$ has no cycle then
    $T \leftarrow T \cup \{e_i\}$
  end if
end for

But how do we test for a cycle (i.e. execute (*))? After each execution of the loop, the set T of edges divides the vertices V into a collection $V_1 \ldots V_k$ of *connected components*. Thus V is the disjoint union of $V_1 \ldots V_k$, each V_i forms a connected graph using edges from T, and no edge in T connects V_i and V_j, if $i \neq j$.

A simple way to keep track of the connected components of T is to use an array $D[1..n]$ where $D[i] = D[j]$ iff vertex i is in the same component as vertex j. So our initialization becomes:

$T \leftarrow \emptyset$
for $i : 1..n$
  $D[i] \leftarrow i$
end for

To check whether $e_i = [r, s]$ forms a cycle with T, check whether $D[r] = D[s]$. If not, and we therefore want to add e_i to T, we merge the components containing r and s as follows:

$k \leftarrow D[r]$
$l \leftarrow D[s]$
for $j : 1..n$
  if $D[j] = l$ then
    $D[j] \leftarrow k$
  end if
end for

The complete program for Kruskal's algorithm then becomes as follows:

Sort the edges so that: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \leftarrow \emptyset$
for $i : 1..n$
  $D[i] \leftarrow i$
end for
for $i : 1..m$
  Assign to r and s the endpoints of e_i
  if $D[r] \neq D[s]$ then
    $T \leftarrow T \cup \{e_i\}$
    $k \leftarrow D[r]$
    $l \leftarrow D[s]$
    for $j : 1..n$
      if $D[j] = l$ then
        $D[j] \leftarrow k$
      end if
    end for

end if
end for

We wish to analyze the running of Kruskal's algorithm, in terms of n (the number of vertices) and m (the number of edges); keep in mind that $n-1 \leq m$ (since the graph is connected) and $m \leq \binom{n}{2} < n^2$. Let us assume that the graph is input as the sequence $n, I_1, I_2, \ldots, I_m$ where n represents the vertex set $V = \{1, 2, \ldots, n\}$, and I_i is the information about edge e_i, namely the two endpoints and the cost associated with the edge. To analyze the running time, let's assume that any two cost values can be either added or compared in one step. The algorithm first sorts the m edges, and that takes $O(m \log m)$ steps. Then it initializes D, which takes time $O(n)$. Then it passes through the m edges, checking for cycles each time and possibly merging components; this takes $O(m)$ steps, plus the time to do the merging. Each merge takes $O(n)$ steps, but note that the total number of merges is the total number of edges in the final spanning tree T, namely (by the above lemma) $n - 1$. Therefore this version of Kruskal's algorithm runs in time $O(m \log m + n^2)$. Alternatively, we can say it runs in time $O(m^2)$, and we can also say it runs in time $O(n^2 \log n)$. Since it is reasonable to view the size of the input as n, this is a polynomial-time algorithm.

1.2 Union-Find data structure

A better way to implement testing for connectivity is by using the Union-Find data structure. That allows to bring the running time of Kruskal's algorithm down to $O(m \log n)$ time.

Union-Find data structure supports three operations:

1) $MakeUnionFind(S)$: for a set of elements S, returns a Union-Find data structure with each element of S in its own disjoint set. This is used in Kruskal's algorithm when initializing the structure with all vertices of the graph (no edges at that point).

2) $Find(u)$ returns the name of a set containing u (usually a set is named after one of its elements). In Kruskal's algorithm, the check whether $Find(u) == Find(v)$ checks if u and v are in the same connected component.

3) $Union(A, B)$: merge sets A and B (e.g., merge two connected components in Kruskal's).

The array implementation we discussed before is not the most efficient one for this data structure. A better implementation is pointer-based: for every element, a node is created. When two sets are merged in a union operation, the pointer of a node at the root of the tree representing the smaller set is pointed to the root representing the larger set. That is, merging two disjoint nodes results in a tree with a root and one child; merging another disjoint node to it gives a tree with the same root as before, but two children and so on. To know which set is larger, we need an additional field keeping the size of a tree rooted in a given node. In this representation, $Union()$ operation takes constant time (update the pointer of a smaller set's root and the size of the larger set's root). The $Find()$ takes $O(logn)$ time because in the worst case one has to follow the path from a leaf to the root of the tree; however because of every time the smaller tree gets attached to the root of the larger one, if a path increased by 1 the size of the whole tree at least doubled. And $MakeUnionFind()$ takes $O(n)$ time, which is OK since it is only done once.

1.3 Correctness of Kruskal's Algorithm

It is not immediately clear that Kruskal's algorithm yields a spanning tree at all, let alone a minimum cost spanning tree. We will now prove that it does in fact produce an optimal spanning tree. To show this, we reason that after each execution of the loop, the set T of edges can be expanded to an optimal spanning tree

using edges that have not yet been considered. Hence after termination, since all edges have been considered, T must itself be a minimum cost spanning tree.

We can formalize this reasoning as follows:

Definition 1 *A set T of edges of G is* promising after stage i *if T can be expanded to a optimal spanning tree for G using edges from $\{e_{i+1}, e_{i+2}, \ldots, e_m\}$. That is, T is promising after stage i if there is an optimal spanning tree T_{opt} such that $T \subseteq T_{opt} \subseteq T \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$.*

Lemma 2 *For $0 \leq i \leq m$, let T_i be the value of T after i stages, that is, after examining edges $e_1, \ldots, e_i$. Then the following predicate $P(i)$ holds for every i, $0 \leq i \leq m$:*

$P(i)$: T_i is promising after stage i.

Proof:
We will prove this by induction. $P(0)$ holds because T is initially empty. Since the graph is connected, there exists *some* optimal spanning tree T_{opt}, and
$T_0 \subseteq T_{opt} \subseteq T_0 \cup \{e_1, e_2, \ldots, e_m\}$.

For the induction step, let $0 \leq i < m$, and assume $P(i)$. We want to show $P(i+1)$. Since T_i is promising for stage i, let T_{opt} be an optimal spanning tree such that
$T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$. If e_{i+1} is rejected, then $T_i \cup \{e_{i+1}\}$ contains a cycle and $T_{i+1} = T_i$. Since $T_i \subseteq T_{opt}$ and T_{opt} is acyclic, $e_{i+1} \notin T_{opt}$. So
$T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.

Now consider the case that $T_i \cup \{e_{i+1}\}$ does *not* contain a cycle, so we have $T_{i+1} = T_i \cup \{e_{i+1}\}$. If $e_{i+1} \in T_{opt}$, then we have $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.
So assume that $e_{i+1} \notin T_{opt}$. Then according to the Exchange Lemma below (letting T_1 be T_{opt} and T_2 be T_{i+1}), there is an edge $e_j \in T_{opt} - T_{i+1}$ such that $T'_{opt} = T_{opt} \cup \{e_{i+1}\} - \{e_j\}$ is a spanning tree. Clearly $T_{i+1} \subseteq T'_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$. It remains to show that T'_{opt} is optimal. Since $T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$ and $e_j \in T_{opt} - T_{i+1}$, we have $j > i+1$. So (because we sorted the edges) $c(e_{i+1}) \leq c(e_j)$, so $c(T'_{opt}) = c(T_{opt}) + c(e_{i+1}) - c(e_j) \leq c(T_{opt})$. Since T_{opt} is optimal, we must in fact have $c(T'_{opt}) = c(T_{opt})$, and T'_{opt} is optimal.

This completes the proof of the above lemma, except for the Exchange Lemma.

Lemma 3 *(Exchange Lemma) Let G be a connected graph, let T_1 be any spanning tree of G, and let T_2 be be a set of edges not containing a cycle. Then for every edge $e \in T_2 - T_1$ there is an edge $e' \in T_1 - T_2$ such that $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G.*

Proof:
Let T_1 and T_2 be as in the lemma, and let $e \in T_2 - T_1$. Say that $e = [u, v]$. Since there is a path from u to v in T_1, $T_1 \cup \{e\}$ contains a cycle C, and it is easy to see that C is the only cycle in $T_1 \cup \{e\}$. Since T_2 is acyclic, there must be an edge e' on C that is not in T_2, and hence $e' \in T_1 - T_2$. Removing a single edge of C from $T_1 \cup \{e\}$ leaves the resulting graph acyclic but still connected, and hence a spanning tree. So $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G. □

We have now proven Lemma 4. We therefore know that T_m is promising after stage m; that is, there is an optimal spanning tree T_{opt} such that $T_m \subseteq T_{opt} \subseteq T_m \cup \emptyset = T_m$, and so $T_m = T_{opt}$. We can therefore state:

Theorem 1 *Given any connected edge weighted graph G, Kruskals algorithm outputs a minimum spanning tree for G.*

1.4 Prim's algorithm

A somewhat different algorithm for Min Spanning Tree problem is due to Prim. This algorithm resembles Dijkstra's algorithm for the shortest-path problem. There is no initial step of sorting; instead, a min spanning tree is grown starting from a root node by adding, at every step, a minimum-weight edge connecting the partial tree with a node outside of a tree. Here, as in Dijstra's algorithm, a priority queue is the main data structure being used: nodes are stored in a priority queue, with their keys being minimal attachment cost to the tree so far.