

# 1 Minimum-Cost Spanning Trees (Section 11.5)

An undirected graph  $G$  is a pair  $(V, E)$ ;  $V$  is a set of vertices.  $E$  is a set of undirected edges, where an edge is a set consisting of exactly two distinct vertices. The degree of a vertex  $v$  is the number of edges touching  $v$ .  $G$  is connected if between every pair of distinct vertices there is a path. 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 cycles ).

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 (the proof is by induction on  $n$ ):

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

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  $G$ ,  $c(T) \leq c(T')$ .

## 2 Kruskal's Algorithm

Given a connected graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges,  $e_1, e_2, \dots, e_m$ , where  $c(e_i) = \text{"cost of edge } e_i\text{"}$ , we want to find a minimum cost spanning tree. It turns out 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  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$ ; 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.

Kruskal( $G$ )

Sort the edges so that :  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$

$T = \{\}$

for  $i = 1$  to  $m$

if  $T \cup \{e_i\}$  has no cycle (\*)

$T = T \cup \{e_i\}$

end if

end for

end

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  $V_1, \dots, V_k$  of connected components. Thus  $V$  is the disjoint union of  $V_1, \dots, V_k$ , 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  $D[1 \dots n]$  where  $D[i] = D[j]$  iff vertex  $i$  is in the same component as vertex  $j$ . So our initialization become:

```
T = {};
for i = 1 to n
    D[i] = 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 = D[r]
l = D[s]
for j = 1 to n
    if D[j] = l
        D[j] = k
    end if
end for
```

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

```
Kruskal(G)
    Sort the edges so that :  $c(e_1) \leq c(e_2) < \dots < c(e_m)$ 
    T = {}
    for i = 1 to n
        D[i] = i
    end for
    for i = 1 to m
        Assign to r and s the endpoints of  $e_i$ 
        if  $D[r] \neq D[s]$  then
            T = T union  $\{e_i\}$ 
            k = D[r]
            l = D[s]
            for j = 1 to n
                if  $D[j] = l$ 
                    D[j] = k
                end if
            end for
        end if
    end for
end
```

We now analyze the running time of Kruskal's algorithm, in terms of  $n$  and  $m$ . Keep in mind that  $n - 1 \leq m$  (since the graph is connected) and  $m \leq \binom{n}{2} \leq n^2$ . 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 take  $O(m)$  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  $(n - 1)$ . Therefore this version of Kruskal's algorithm runs in time  $O(m \log m + n^2)$ . Since  $m < n^2$ , we have

$$m \log m < n^2 \log n^2 < 2n^2 \log n,$$

hence, we can also say it runs in time  $O(n^2 \log n)$ .

### 3 Correctness of Kruskal

**Lemma 2.** (*Exchange Lemma*) Let  $G$  be a connected graph, let  $T_1$  be any spanning tree of  $G$ , and  $T_2$  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  $e \in T_2 - T_1$  and  $e = \{u, v\}$ . Since there is a path from  $u$  to  $v$  in  $T_1$ ,  $T_1 \cup \{e\}$  contains a simple cycle  $C$ , and  $C$  is the only simple 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  $G$ .

To show that Kruskal's algorithm yields a minimum cost spanning tree, 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.

**Definition 3.** A set  $T$  of edges of  $G$  is promising after stage  $i$  if  $T$  can be expanded to an optimal spanning tree for  $G$  using edges from  $\{e_{i+1}, e_{i+2}, \dots, e_m\}$ . That is,  $T$  is promising after stage  $i$  if there is an optimal spanning tree  $T_{opt}$  such that  $T \subseteq T_{opt} \subseteq T \cup \{e_{i+1}, e_{i+2}, \dots, e_m\}$ .

**Lemma 4.** For  $0 \leq i \leq m$ , let  $T_i$  be the value of  $T$  after  $i$  stages, that is, after examining edges  $e_1, \dots, e_i$ . Then the following predicate  $P(i)$  holds for every  $i$ ,  $0 \leq i \leq m$

$$P(i) : T_i \text{ is promising after stage } i.$$

We prove by mathematical induction.  $P(0)$  holds because  $T$  is initially empty. Since the graph is connected, there exists some optimal spanning tree  $T_{opt}$ , and  $T_0 \subseteq T_{opt} \subseteq T_0 \cup \{e_{i+1}, e_{i+2}, \dots, e_m\}$ .

For the induction step, let  $0 \leq i \leq m$  and assume that  $P(i)$ . We show that  $P(i + 1)$ . That is, we assume that  $T_i$  is promising for stage  $i$ , let  $T_{opt}$  be an optimal spanning tree such that  $T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \dots, e_m\}$ , we want to prove that  $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \dots, e_m\}$ . There are three cases I)  $e_{i+1}$  is rejected, or II)  $e_{i+1}$  is accepted and  $e_{i+1} \in T_{opt}$ , or III)  $e_{i+1}$  is accepted and  $e_{i+1} \notin T_{opt}$ . Case I and Case II would be straightforward to prove. Case III uses the exchange lemma to show that  $T_i \cup e_{i+1}$  would be promising too.