CS 341: ALGORITHMS (S18) — LECTURE 14
MINIMUM SPANNING TREES

ERIC BLAIS

Today, we examine weighted graphs and focus on a particularly fundamental problem: finding the minimum spanning tree of a graph.

1. Spanning trees

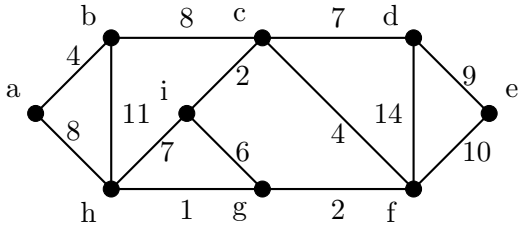
Recall that a *spanning tree* of a connected undirected graph $G = (V, E)$ is a tree $T = (V, E')$ whose edges $E' \subseteq E$ are a subset of the edges of G. We can think of a spanning tree as a "minimum skeleton representation" of a graph: using a spanning tree T of G, we can identify a path in G between any two vertices $u, v \in V$, and yet to store T we only need to store $n - 1$ edges (instead of as many as $\Omega(n^2)$ for G itself).

We already saw that finding a spanning tree of a connected graph can be done in time $O(n + m)$ by running either a depth-first search or a breadth-first search and storing the resulting DFS or BFS tree.

In general, a connected graph G can have multiple spanning trees. Depending on the application we have in mind, we may want to find the "best" spanning tree of a graph. This problem becomes particularly important when we consider *weighted* graphs.

Definition 14.1 (Weighted graphs)**.** A *weighted graph* is a graph $G = (V, E)$ and a function $w : E \to \mathbb{R}$, where for each edge $e \in E$, the value $w(e)$ is the *weight* of e in G.

The following is an example of a weighted graph.



Weighted graphs as we have defined them are sometimes called *edge-weighted graphs*, to distinguish them from the graphs for which we assign weights to the vertices instead of the edges.

There are many problems that map nicely to weighted graphs. For example, with the graph obtained by representing airports with vertices and connecting two vertices with an edge if and only if there is a direct flight going between the two corresponding airports, we can use weights to represent the time of the flight, or the cost of a ticket on one of those flights, for example. The weight of the edges can then be used to solve problems such as finding the cheapest route between two airports, or the one with the shortest total flight time, or other similar problems.

The weights on edges give a very natural way to compare different spanning trees, which gives rise to the *mininum spanning tree* problem.

Definition 14.2 (Minimum spanning tree)**.** An instance of the *minimum spanning tree (MST)* problem is a weighted graph $G = (V, E)$, $w : E \to \mathbb{R}$. A valid solution to this problem is a spanning tree $T = (V, E')$ of G with minimum total weight $\sum_{e \in E'} w(e)$.

The typical example that you can keep in mind when considering the MST problem is the road building problem: you have a number of cities that you want to connect with roads (or with high-speed train lines!). We can represent the pairs of cities that you can connect with a road using edges; the weight of each edge represents the cost of building that road. (Not all vertices need be connected by an edge, as you may not be able to build a road directly between two cities if, for example, they are separated by a formidable mountain chain.) Then the cheapest way to connect all the cities by roads corresponds to the minimum spanning tree of the resulting graph.

2. Kruskal's algorithm

There is a natural greedy strategy for constructing the minimum spanning tree of a graph G: consider the edges of G in order of increasing weight (i.e., from the lightest to the heaviest), adding an edge to our tree whenever its endpoints are two vertices that have not already been connected. This greedy algorithm is known as *Kruskal's algorithm*.

Algorithm 1: Kruskal$(G = (V, E), w)$

```
Sort the edges in E by increasing weight w;
E' ← ∅;
for each edge (u, v) ∈ E do
    if Component(E', u) ≠ Component(E', v) then
        E' ← E' ∪ {(u, v)};
return T = (V, E');
```

To fully specify Kruskal's algorithm, we must also define the algorithm IsConnected. Let us leave this task for a later part of the course, when we study graph exploration algorithms. For now, let's treat that algorithm as a black box and show that Kruskal's algorithm returns a minimum spanning tree of the input graph G. You can see how Kruskal's algorithm finds a minimum spanning tree in the example graph at the beginning of this lecture in the course textbook (CLRS).

To show that Kruskal's algorithm correctly solves the MST problem on all graphs, we establish a powerful lemma that can also be used to prove the correctness of other MST algorithms as well.

Definition 14.3. A *cut* in a graph $G = (V, E)$ is a partition of the vertices V into two sets $(S, V \setminus S)$. An edge is said to *cross* the cut if it connects a vertex in S to one in $V \setminus S$.

Lemma 14.4 (Cut property)**.** *Let X be a set of edges that are part of a minimum spanning tree of the connected graph $G = (V, E)$. Pick any set $S \subseteq V$ such that no edge in X crosses the cut $(S, V \setminus S)$ and let e be the lightest edge that crosses this cut. Then $X \cup \{e\}$ is part of a minimum spanning tree of G.*

Proof. Let T be a MST of G that includes all the edges in X. If T also includes the edge e, then we are done. Otherwise, consider the graph obtained by adding the edge $e = (u, v)$ to T. This edge must create a cycle, which means that there must be another edge $e' \neq e$

in T that crosses the cut $(S, V \setminus S)$. Let T' be the tree obtained by adding e and removing e' from T.[1] The weight of T' is

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

But $w(e) \leq w(e')$ since e is the lightest edge crossing the cut $(S, V \setminus S)$, so $\text{weight}(T') \leq \text{weight}(T)$ and T' is a minimum spanning tree of G that contains all the edges in $X \cup \{e\}$. □

The proof of correctness of Kruskal's algorithm follows easily from the cut property.

Theorem 14.5. *Kruskal's algorithm solves the Minimum Spanning Tree problem.*

Proof. We proceed by induction, showing at every step of Kruskal's algorithm, there is a minimum spanning tree of G that contains the set E' of edges that have been selected up to that point. The base case is the initial set $E' = \emptyset$, for which the assertion is trivially true.

For the induction step, the induction hypothesis states that the set $X = E'$ of edges already selected by the algorithm are part of a MST of G. The next edge e added by Kruskal's algorithm connects two previously unconnected components; call them T_1 and T_2. Let S be the set of vertices in T_1. Then X does not have any edges that cross the cut $(S, V \setminus S)$ and by the order in which we consider the edges e is the lightest edge that crosses the cut, so $X \cup \{e\}$ is also part of a MST of G, as we wanted to show. □

2.1. **Implementing Kruskal's algorithm.** How efficient is Kruskal's algorithm? Naïvely, we might think that it runs in time $\Theta(m \log m)$ since it sorts the edges by weight then considers each edge only once. But there is one non-trivial catch: to obtain this runtime, we need an efficient algorithm for checking if two vertices are in different connected components of the graph. We can do this with DFS and BFS algorithms, but this implementation is not very efficient; to obtain a $O(m \log m)$ time complexity for Kruskal's algorithm, we need to use a more sophisticated approach. This can be done with the *Union Find* data structure, a topic that is covered in CS 466.

3. Prim's algorithm

There is another very natural greedy approach to building the minimum spanning tree of a graph G, that follows almost immediately from the cut property. At each step, we have a tree connecting a subset of the vertices of the graph (initially it is a single vertex) and we grow the tree by finding the lightest edge that connects the tree to another previously-unconnected vertex.

Algorithm 2: Prim$(G = (V, E), w)$

```
Initialize S = {v} for any v ∈ V;
Initialize E' = ∅;
Sort the edges in E by weight w;
while S ≠ V do
    (u, v) ← FindMinCutEdge(E, S);
    E' ← E' ∪ {(u, v)};
    S ← S ∪ {u, v};
return T = (V, E');
```

[1] Exercise: verify that T' is indeed a tree!

We can again use the Cut Property to prove the correctness of this algorithm.

Theorem 14.6. *Prim's algorithm solves the Minimum Spanning Tree problem.*

Proof. By definition, at every iteration of the while loop the edges in E' do not cross the cut $(S, V \setminus S)$ so the correctness of the algorithm again follows from the Cut Property. □

3.1. **Implementation.** Once again, it is not immediately obvious how to implement Prim's algorithm most efficiently. This can be done with a priority queue. A natural implementation of this approach has time complexity $O(m \log m)$. It is also possible to use Fibonacci heaps to do even better, with a final time complexity of $O(m + n \log n)$. See the course textbook for the details.