Lecture 20: Minimum Spanning Tree

built on 2019/06/26 at 18:25:47

1 Spanning Trees

Consider the following graph (the graph on the right will be used soon)

```
a --- b --- c --- d --- e
 \               |    /
  --------------- f ---
```

```
a -2- b -5- c -1- d -9- e
 \                |8   /
  -----2--------- f -4-
```

We wish to use the fewest number of edges to make sure all the vertices can reach each other. *How many edges do we need?* Which edges?

After some experiments, you probably have a hunch that

you need $n-1$ edges for an n-vertex graph.

Why? With some work, we can show that if a graph is a connected graph, the structure that has the fewest edges which can connect up all the vertices is a *spanning tree*. As the name suggests, a spanning tree is a tree on all the vertices—and it spans, as in it stretches out to all these vertices and connect them up.

In the above example, there are many spanning trees and there are all equally good, in the sense that it has exactly $n-1$ edges.

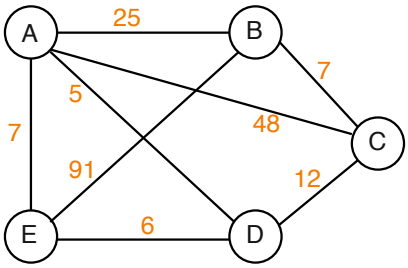
But what if some edges turn out to be more expensive than others:

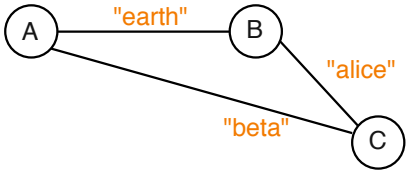
- A moment's thought shows that we still need $n-1$ edges to connect them up....
- However, we should prefer certain edges over some others—because they are less expensive. *Which edges to pick?* Experiment?

2 Weighted Graphs

In this class so far, our graphs are made up simply of vertices and edges connecting vertices. They could be directed or undirected. At some level, all the edges are equal. In many applications, though, we may wish to annotate each of these edges with a label or sometimes a number representing, e.g., its strength or its cost. We begin today's lecture with a discussion of how to do exactly that.

To add such annotations to a graph, we graphically just add these labels next to the edges. For example, the graphs below show the connections as well as labels on the edges:





But how do we represent these annotations mathematically? Previously, a graph is an ordered pair $G = (V, E)$. To add these annotations, which are often *weights*, a *weighted graph* is usually denoted by $G = (V, E, w)$, where V and E are the same as before, and w is a function $w\colon E \to \mathsf{Label}$ from the set of edges to a set of labels. Often, the label set is simply $\mathbb{R}_+$ (the positive reals). The idea of this "weight" function is that for every edge $e \in E$, $w(e)$ gives the label on that edge.

How do we store such information in our code? Idea #1: we could follow the mathematical definition and store a map from every edge to its key. For example, we would have `Map<Edge, Integer>`, where this is perhaps kept as a `HashSet`. Details are left as exercise to the reader.

Idea #2: Most often, when we want the label on an edge, it is at the same time that we access that edge. So it seems natural to bundle the label with the place that stores that edge itself. To see how this might work, let's remember how our favorite representation, adjacency table, works. In this representation, we keep a map from each vertex to its set of neighbors—in Java type, `Map<Vertex, Set<Vertex>>`, where `Vertex` represents the vertex type (in our case, it's mostly Integer).

If we need to store information on the edges, we could store, instead of a set of vertices, a map of vertices. Here the keys of the map would be the neighbors and the values would be the labels corresponding to the key. This means, we'll keep a map `Map<Vertex, Map<Vertex, Label>>`.

As an example, for the graph on the left above, we would store the graph as following type:

```
Map<String, Map<String, Integer>>
```

because the vertices are strings and the labels are integer. Each map could be kept as a `HashMap` or a `TreeMap`. The entries would be as follows (using Python's dictionary notation):

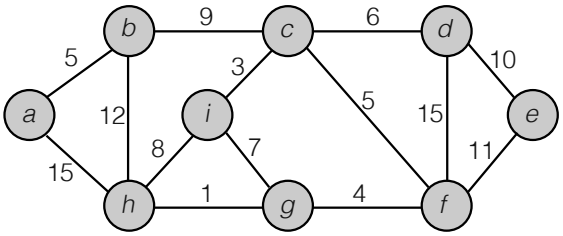
```
G = { 'A' : { 'B': 25, 'C': 48, 'D': 5, 'E': 7 },
      'B' : { 'A': 25, 'E': 91, 'C': 7 },
      'C' : { 'B': 7,  'A': 48, 'D': 12 },
      'D' : { 'C': 12, 'A': 5,  'E': 6 },
      'E' : { 'A': 7,  'B': 91, 'D': 6 }
    }
```

Modifications to our Undirected Graph ADT: In the view of our undirected graph ADT from previous lectures, we would make `adj` return a `Map<Vertex, Label>` instead of a `Set<Vertex>`. We could add a method `Label getWeight(Vertex u, Vertex v)`.

3 The Minimum Spanning Tree Problem

We have just seen how to model a graph so that the edges can carry information. In a road map, the edge weights could represent the distance or fuel cost to drive along them. In a power distribution network, the edges could represent transmission lines and the weights could represent the lengths or how much power is needed.

To motivate what we're about to study next, consider a fictitious situation where you're building roads in a village. There are n points of interests, modeled as vertices. And you've surveyed the cost of building a road between point A and point B. Not all pairs of points can be directly connected. From the survey, you have come up with the following graph:



You want to connect all these points up. To interconnect n points, you can show that you need at least $n-1$ connections. Now you wish to know which $n-1$ connections to put in to minimize the cost.

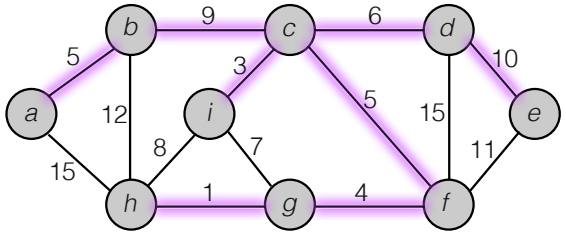
For starters, it is easy to see that these $n-1$ connections (edges) should not form any loop. After all, a loop doesn't connect you to a new location. Hence, the structure we want is what's known as a spanning tree:

> A *spanning tree* of a graph G is a connected subgraph (using a subset of the edges of G) that includes all vertices of G and that has no cycles.

The minimum spanning tree problem is therefore to find a spanning tree whose total cost is the smallest possible among all spanning trees on the given graph. More formally, the minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, where each edge e has weight $w(e) \geqslant 0$, find a spanning tree of minimum weight (i.e., the sum of the weights of the edges). That is to say, we are interested in finding the spanning tree T that minimizes

$$w(T) = \sum_{e \in E(T)} w(e).$$

For the example above, the spanning tree that has the smallest possible cost—that is, a minimum spanning tree—is as follows: edges shaded in light purple are the edges that make up the MST.



To continue our quest, we ask the question, *given a connected graph, how many MSTs are there?* The following lemma makes our lives easy when the weights are distinct:

Lemma 3.1 *There is a unique minimum spanning tree of G provided that G is connected and has distinct weights.*

We leave the proof to our next assignment. We note, however, that even though there may be duplicate weights, we could break ties in a consistent way and make them distinct for the purpose of MST. Hence, we'll assume throughout the rest of this lecture that the weight edges are distinct (this is not necessary for the implementation in practice but simplifies the exposition).

3.1 Underlying Principles: Light Edge Rule

The main property that underlines many MST algorithms is a simple fact about cuts in a graph. To begin, we'll make a few observations about a tree:

Observation 3.1 *If T is a tree,*

- *adding an edge between vertices of T creates a cycle.*
- *removing an edge from T breaks it into exactly two trees.*

(In some sense, a tree is the minimally connected graph on this set of vertices.)

What is a cut? For a graph $G = (V, E)$, a *cut* is defined in terms of a subset $U \subsetneq V$. This set U partitions the graph into $(U, V \backslash U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \backslash U$. The subset U might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \varnothing$ and $U \neq V$).

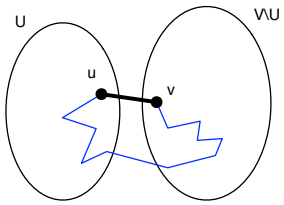
An Example: Using the graph above, if we use $U = \{a, b, h, i\}$, then the other vertices will be in $V \backslash U$—and the cut looks like this:

```
Draw an example
```

Light Edge Rule. The following theorem states that the lightest edge across a cut is in the MST of G:

Theorem 3.2 (Light-Edge Rule) *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge e between U and $V \backslash U$ is in the minimum spanning tree MST(G) of G.*

Proof: The proof is by contradiction. Let a cut $(U, V\backslash U)$ be given. We'll denote by $e = u, v$ the lightest edge going across this cut. Now consider the minimum spanning tree (MST) T. Suppose for a contradiction that e is not in the MST T. Since the MST spans the graph, there must be a path P connecting u and v using just edges of the MST. The path must cross the cut between U and $V\backslash U$ at least once since u and v are on opposite sides. By attaching e to P, we form a cycle (recall that by assumption $e \notin T$). If we remove the maximum weight edge from P and replace it with e, we will still have a spanning tree, but it will be have less weight. This is a contradiction to the fact that T is a minimum spanning tree. Hence, we conclude that e—the minimum-weighted edge across the cut $(U, V\backslash U)$—must appear in the MST. ■



Note that the last step in the proof uses the facts that (1) adding an edge to a spanning tree creates a cycle, and (2) removing any edge from this cycle creates a tree again. Finally, we contradict the fact that the minimum spanning tree is the spanning tree that has the least cost.

3.2 Prim's Algorithm

The first efficient algorithm we'll discuss is what's known in the literature as *Prim's algorithm*. The idea is to maintain a single, connected tree and keep growing it one edge at a time using the light-edge rule until we finally have the MST. In pseudocode, we have the following:

Algorithm 1: `Prim`$(G = (V, E, w))$ — Prim's algorithm for MST

```
Pick a starting vertex s ∈ V arbitrarily
Let U_0 ← {s}, MST ← {}
for i = 1, 2, ..., n − 1 do
    Apply the light-edge rule on the cut (U_{i−1}, V\U_{i−1})
    Let e_i = (x, y) be the minimum-weighted edge across the cut
    Add e_i to MST
    Set U_i ← U_{i−1} ∪ {x, y} // Either x or y was already in U_{i−1}
return MST
```

Correctness of this algorithm follows immediately from the light-edge rule. Notice that by construction, at the end of iteration i, *MST* is connected and is a spanning tree on the vertex set U_i. Furthermore, there cannot be any cycle because each e_i joins U_{i-1} with a vertex outside of U_{i-1}, extending it to U_i.

We show how Prim works in action in Figure 1. How would we implement this algorithm in Java? Hint: Use a priority queue. We'll discuss some implementation details toward the end of the lecture.

3.3 Kruskal's Algorithm

The second algorithm for MST is Kruskal's algorithm. While still based on the light-edge rule, it maintains a forest (pieces that will eventually grow to be the MST) and joins pieces of this forest by considering graph edges in increasing order of their weights (i.e., small to large). Initially, the forest consists of n trees, each a lone vertex.

In pseudocode, the algorithm is as follows:

The question is, how can we check quickly whether x can or cannot reach y in the MST? We already know how to do this last time: the union-find ADT. Hence, we can refine the algorithm as follows:

Correctness of Kruskal's algorithm hinges on the following observation:

> When we consider $e_i = (x, y)$, if e_i doesn't form a cycle with existing MST edges (test performed by union-find, we know that it crosses a cut described as follows: Let U_x be all the vertices reachable from x using *MST* so far. The cut of interest is the cut $(U_x, V\backslash U_x)$. Notice that e_i is the minimum-weighted edge crossing this cut since after all the edges have been sorted and we consider them from small to large.

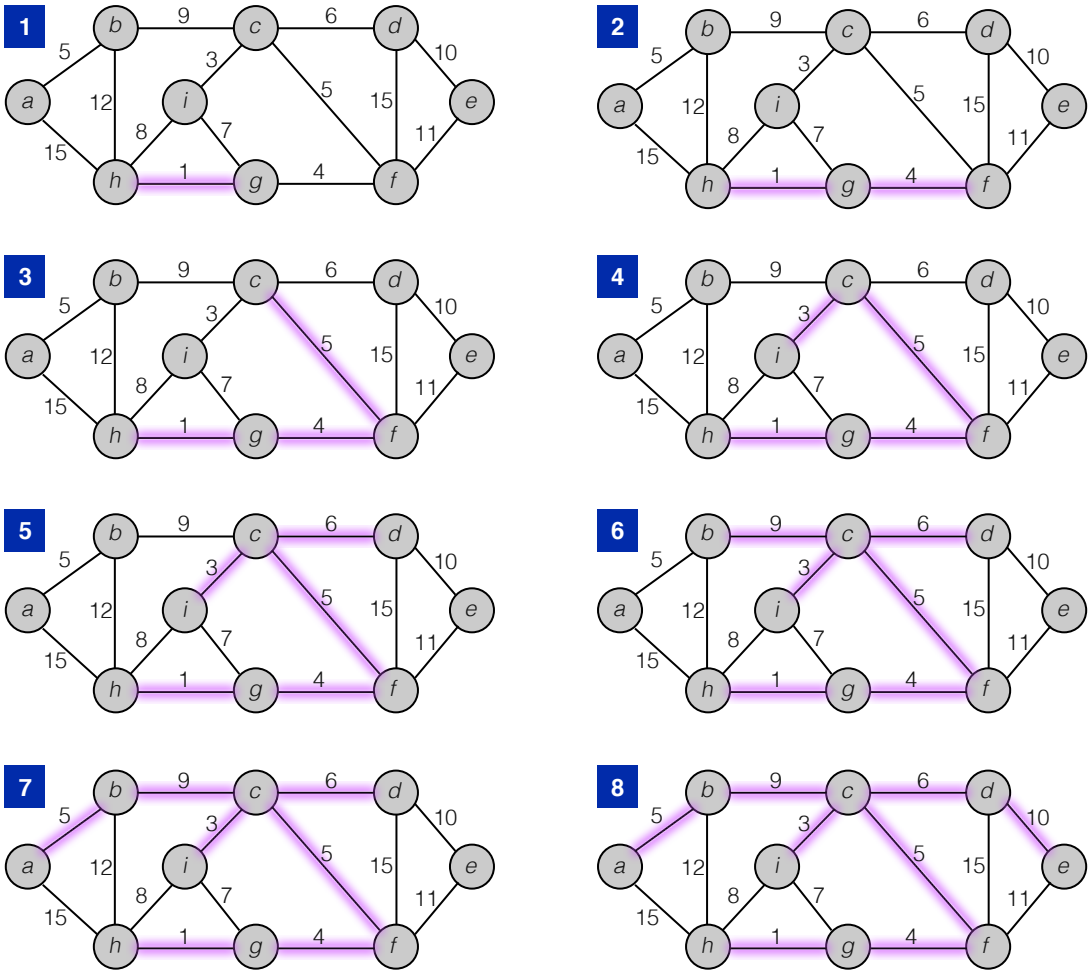


Figure 1: Steps taken by Prim's algorithm on the sample graph.

We further note that *MST* cannot contain any cycle because union-find discards every edge that leads to a cycle. To see how this works in action, we show the steps taken by Kruskal's on the example graph in Figure 2.

4 Implementation Notes and Running Time

Kruskal's: We begin with Kruskal's algorithm as it arguably involves fewer moving parts. It is pretty straightforward to implement Kruskal's algorithm: we need to be able to sort; we need a fast union-find data structure. But that's about it. We leave an actual implementation to the reader.

The running time can be analyzed as follows:

- To sort the edges initially costs $O(m \log m)$, which is actually $O(m \log n)$ [why? convince yourself].
- Once that's done, for each of the m edges, the most work one can do is to follow through with the `if`. That is, run two `finds` on the union-find data structure, add e_i the MST (say this is a list), and run a `union` operation. Therefore, each edge costs $O(\log n)$, for a total of $O(m \log n)$
- Hence, Kruskal's is an $O(m \log n)$ algorithm.

Prim's: More tricky to implement is Prim's algorithm. Although it doesn't involve a union-find data structure, it requires being able to answer the following question quickly: *Given a set of vertices U_i, what is the minimum-weighted edge out of U_i?*

Answering this question requires some setting up and a bit imagination. We are going to keep a priority queue of *candidate vertices* that can be reached from U_i ordered by their weights. Also, instead of maintaining

Algorithm 2: `Kruskal`$(G = (V, E, w))$ — Kruskal's algorithm for MST

Sort the edges E so that $w(e_1) < w(e_2) < \cdots < w(e_m)$, where e_i's are the edges of G
for $i = 1, 2, \ldots, m$ **do**
  Let $\{x, y\} \leftarrow e_i$ (i.e., x and y are the endpoints of e_i)
  if *x cannot reach y in MST* **then**
    Add e_i to *MST // Add to MST*
return *MST*

Algorithm 3: `Kruskal`$(G = (V, E, w))$ — Kruskal's algorithm for MST

Sort the edges E so that $w(e_1) < w(e_2) < \cdots < w(e_m)$, where e_i's are the edges of G
Let `UF` be a union-find structure
for $i = 1, 2, \ldots, m$ **do**
  Let $\{x, y\} \leftarrow e_i$ (i.e., x and y are the endpoints of e_i)
  if *`UF.find`*$(x) \neq$ *`UF.find`*(y) **then**
    Add e_i to *MST // Add to MST*
    `UF.union`(x, y) *// Join trees*
return *MST*

multiple U_i's, we're only gonna keep the current version and call that `visited`. Then, the algorithm (sketch) is as follows:

```
List<Edge> primMST(WeightedUndirectedGraph G) {
  PriorityQueue<Edge> pq = new PriorityQueue<Edge>(
       (Edge x, Edge y) -> x.cost.compareTo(y.cost)
    );
  List<Edge> mst = new ArrayList<>();
  Set<Integer> visited = new HashSet<>();
  visit(0, G, pq, visited);
  while (!pq.isEmpty()) {
    Edge e = pq.poll();
    if (visited.contains(e.u) &&
        visited.contains(e.v)) continue;
    mst.add(e);
    if (!visited.contains(e.u)) visit(e.u, G, pq, visited);
    if (!visited.contains(e.v)) visit(e.v, G, pq, visited);
  }
  return mst;
}
void visit(int vtx, WeightedUndirectedGraph G,
           PriorityQueue<Edge> pq, Set<Integer> visited) {
   visited.add(vtx);
   for (Integer w : G.adj(vtx)) {
     if (!visited.contains(w)) pq.add(G.getEdge(vtx, w));
   }
}
```

What's the running time of this code? Here are some things to consider:

- Each vertex can be an argument to `visit` at most once. Each time `visit` is called with v, the cost is $O(\deg(v) \log n)$ as it goes over the neighbors of v, adding each of the $O(\deg(v))$ neighbors to the priority queue. Also, the number of entries added to `pq` by this vertex is $\deg(v)$. Hence, the total cost due to the

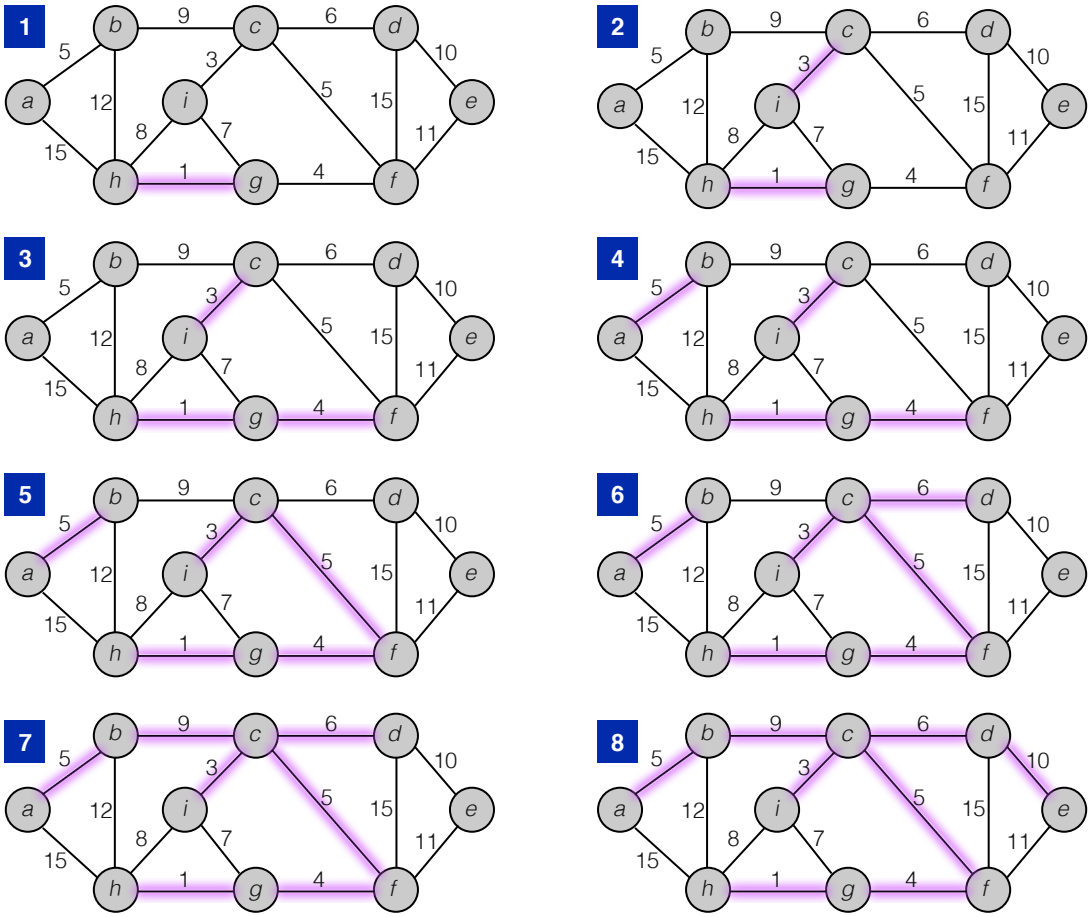


Figure 2: Steps taken by Kruskal's algorithm on the sample graph.

`visit` function is $\sum_{v \in V} \deg(v) \log n = O(m \log n)$. Also, the number of times `pq.add` is called is at most $\sum_{v \in V} \deg(v) = 2m$.

- The cost of the **`while`**-loop can be bounded as follows: We don't know how many times exactly that loop will run. But we do know that (a) each time it's run, it costs at most $O(\log n)$, excluding the cost to the `visit` calls. This $O(\log n)$ is due to the delete min step. Furthermore, we know that (b) the number of times it can delete min is no more than the number times `add` is ever called. But the number of times `add` is ever called is $\leqslant 2m$, so the cost here is, once again, $O(m \log n)$.
- In all, Prim's runs in $O(m \log n)$ time.