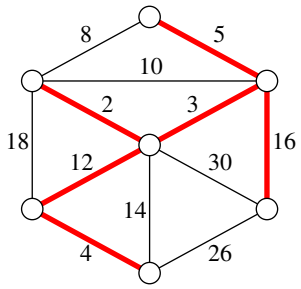
11 Minimum Spanning Trees (March 2)

11.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w\colon E \to \mathbb{R}$ that assigns a *weight* $w(e)$ to each edge e. For this lecture, we'll assume that the weights are real numbers. Our task is to find the *minimum spanning tree* of G, *i.e.*, the spanning tree T minimizing the function

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

To keep things simple, I'll assume that all the edge weights are distinct: $w(e) \neq w(e')$ for any pair of edges e and e'. Distinct weights guarantee that the minimum spanning tree of the graph is unique. Without this condition, there may be several different minimum spanning trees. For example, if all the edges have weight 1, then *every* spanning tree is a minimum spanning tree with weight $V - 1$.



A weighted graph and its minimum spanning tree.

If we have an algorithm that assumes the edge weights are unique, we can still use it on graphs where multiple edges might have the same weight, as long as we have a consistent method for breaking ties. One way to break ties consistently is to use the following algorithm in place of a simple comparison. SHORTEREDGE takes as input four integers i, j, k, l, and decides which of the two edges (i, j) and (k, l) has smaller weight.

```
SHORTEREDGE(i,j,k,l)
    if w(i,j) < w(k,l) return (i,j)
    if w(i,j) > w(k,l) return (k,l)
    if min(i,j) < min(k,l) return (i,j)
    if min(i,j) > min(k,l) return (k,l)
    if max(i,j) < max(k,l) return (i,j)
    /* if max(i,j) < max(k,l) */ return (k,l)
```

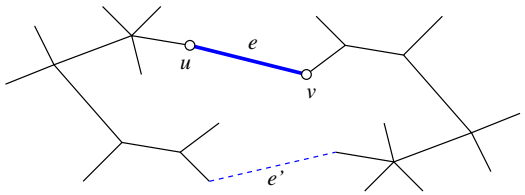
11.2 The Only MST Algorithm

There are several different ways to compute minimum spanning trees, but really they are all instances of the following generic algorithm. The situation is similar to the previous lecture, where we saw that depth-first search and breadth-first search were both instances of a single generic traversal algorithm.

The generic MST algorithm maintains an acyclic subgraph F of the input graph G, which we will call an *intermediate spanning forest*. F is a subgraph of the minimum spanning tree of G, and every component of F is a minimum spanning tree of its vertices. Initially, F consists of n one-node trees. The generic MST algorithm merges trees together by adding edges between them. When the algorithm halts, f consists of a single n-node tree, which must be the minimum spanning tree.

We say that an edge $e \in E$ is *light* if it is the minimum-weight edge joining two trees in F, and *heavy* otherwise. In particular, if both endpoints of e are in the same component of F, then then e is heavy. Since edge weights are unique, any two trees in F have at most one light edge between them.

Heavy edges cannot be in the minimum spanning tree. To prove this, suppose the minimum spanning tree T contains a heavy edge $e = (u, v)$. The endpoints u and v must be in different components of F, since otherwise T would contain a cycle. Let e' be the light edge joining the components of F containing u and v. By definition, $w(e') < w(e)$. Then $T \setminus \{e\} \cup \{e'\}$ is a spanning tree of G, and since $w(e') < w(e)$, it has smaller total weight than T. But this is impossible, since T is the *minimum* spanning tree.



The minimum spanning tree has no heavy edges.

The generic MST algorithm joins together trees in F by adding one or more light edges. The algorithm has to be a little careful about *which* light edges it uses, since there are light edges that don't belong to the MST. As the trees in F coalesce, certain edges that used to be light become heavy. When the algorithm beings, *every* edge in the graph is light, and when the algorithm ends, every edge is heavy.

Suppose we build a graph G' whose nodes are the components of F and whose edges are the light edges in G. Then we can think of the generic minimum spanning tree algorithm as *merging* pairs of nodes in G' until there is only one node left.

11.3 Borůvka's Algorithm

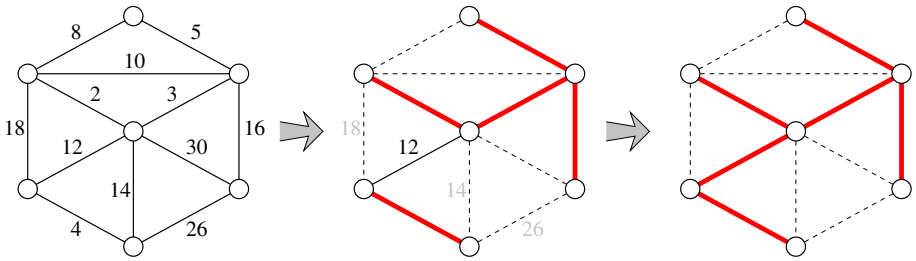
The oldest minimum spanning tree algorithm was discovered by Borůvka in 1926 long before there were such things as computers or algorithms, and practically before the invention of graph theory![1] Borůvka's algorithm repeats the following two steps until F consists of a single tree:

1. Mark the lightest edge incident to each component of F.
2. Add the marked edges to F.

If we think in terms of the component graph G', we can describe Borůvka's algorithm in one line:

> Merge each node in G' with its nearest neighbor and recurse.

[1]The first book on graph theory, written by D. König, was published in 1936. Leonard Euler published his famous theorem about the bridges of Königsburg (HW3, problem 2) in 1736. Königsburg was not named after *that* König.



Borůvka's algorithm run on the example graph. Thick edges are in F. Dashed edges are heavy.

At the beginning of each phase of the Borůvka algorithm, every component of F elects a "leader" node. The simplest way to do this is by performing a depth-first search of F; the first node we visit in each component is that component's leader. For every leader $\overline{v}$ of some component of F, let lightest($\overline{v}$) denote the lightest edge adjacent to that component.

```
BORŮVKA(V,E):
  F = (V, ∅)
  while F has less than V − 1 edges
      choose leaders using DFS
      for each leader v̄
          lightest(v̄) ← ∞
      for each edge (u, v) ∈ E
          ū ← u's leader
          v̄ ← v's leader
          if ū ≠ v̄
              if w(u, v) < w(lightest(ū))
                  lightest(ū) ← (u, v)
              if w(u, v) < w(lightest(v̄))
                  lightest(v̄) ← (u, v)
      for each leader v̄
          add lightest(v̄) to F
```

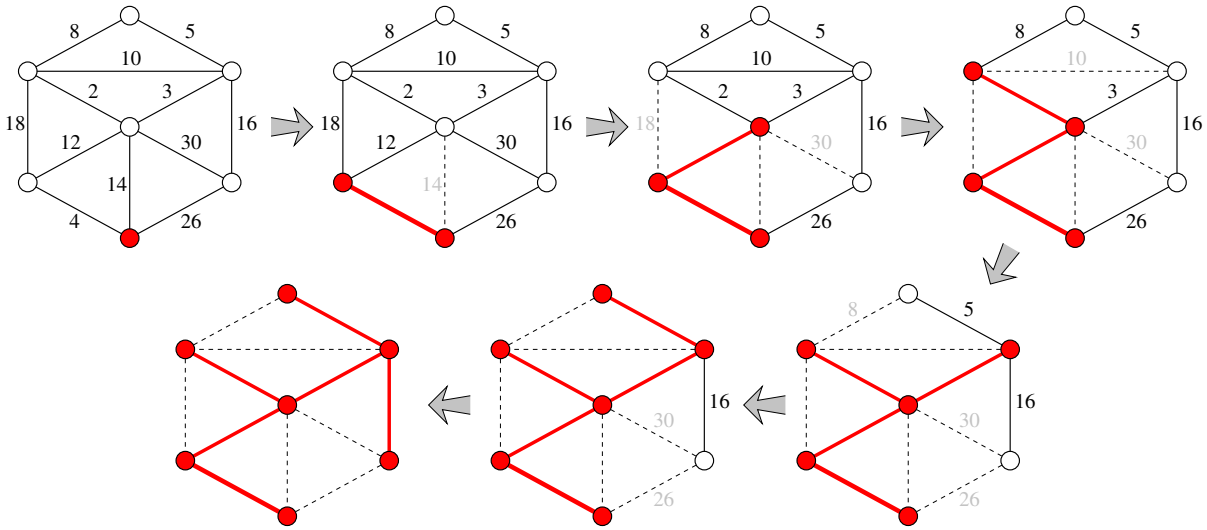
Each iteration of the while loop in BORŮVKA takes $O(E)$ time, assuming the graph is represented by an adjacency list. Each iteration also reduces the number of components of F by at least a factor of two. The worst case occurs when the components coalesce in pairs. Since there are initially V components, there are $O(\log V)$ iterations. Thus, the overall running time of Borůvka's algorithm is $O(E \log V)$.

11.4 Jarník's ("Prim's") Algorithm

The next oldest minimum spanning tree algorithm was discovered by Jarník in 1930, but it usually called Prim's algorithm — Prim independently rediscovered the algorithm in 1956 and gave a much more detailed description than Jarník. Such is fame in computer science.

In Jarník's algorithm, the forest F contains only one nontrivial component T; all the other components are isolated vertices. Initially, T consists of an arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:

> Find the lightest edge adjacent to T and add it to T.



Jarník's algorithm run on the example graph. Thick edges are in F. Dashed edges are heavy.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a heap. When we pull the minimum-weight edge off the heap, we first check whether both of its endpoints are in F. If not, we add the edge to F and then add the new neighboring edges to the heap.

In other words, Jarník's algorithm is just another instance of the generic graph traversal algorithm we saw last time, using a heap as the "bag"! If we implement the algorithm this way, its running time is $O(E \log E) = O(E \log V)$.

However, we can speed up the implementation by observing that each vertex is visited only once. Rather than keeping edges, we will keep vertices in the heap, where the key of each vertex v is the length of the shortest edge between v and T. Each time we add an edge to T, we may need to decrease the key of some neighboring vertices.

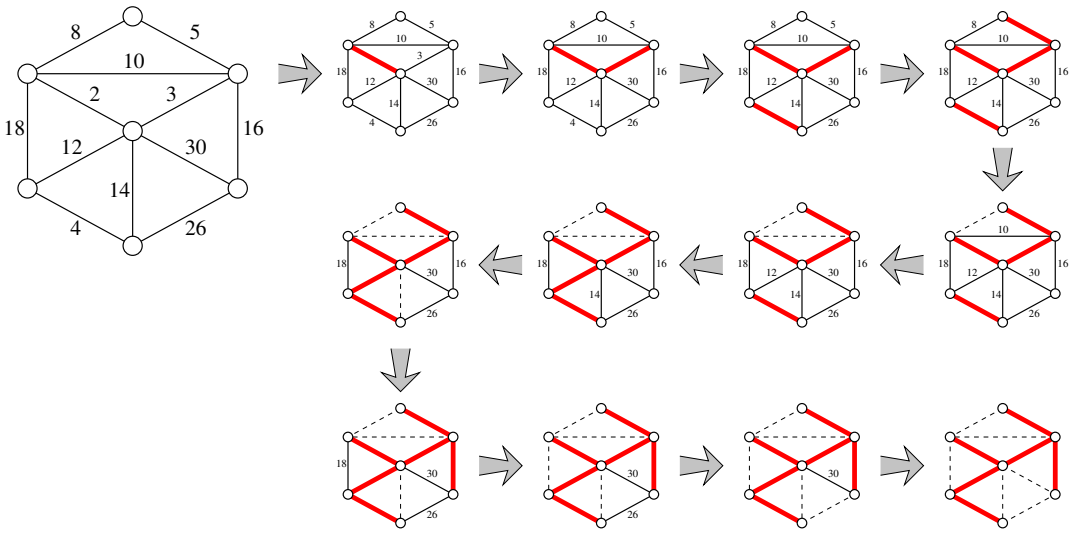
```
JARNÍK(V, E):
  T ← (V, ∅)
  for each vertex v
      key(v) ← ∞
      edge(v) ← NULL
      INSERT(v)

  v ← EXTRACTMIN
  while v ≠ NULL
      if edge(v) ≠ NULL
          add edge(v) to T
      for each edge (u, v)
          if key(u) > w(u, v)
              DECREASEKEY(u, w(u, v))
              edge(u) = (u, v)
      v ← EXTRACTMIN
```

This implementation of JARNÍK is dominated by the cost of the heap operations INSERT, EXTRACTMIN, and DECREASEKEY. INSERT and EXTRACTMIN are each called $O(V)$ times, and DECREASEKEY is called $O(E)$ times. Thus, if we store the nodes in a Fibonacci heap, the overall running time is $O(E + V \log V)$. This is faster than Borůvka's algorithm unless $E = O(V)$.

11.5 Kruskal's Algorithm

The last MST algorithm I'll discuss was discovered by Kruskal in 1956. Kruskal's algorithm scans through the edges of the graph in increasing order of weight and adds each edge to F if it is light. Because of the order in which edges are scanned, an edge is light if and only if its endpoints belong to two different components of F.



Kruskal's algorithm run on the example graph. Thick edges are in F. Dashed edges are heavy.

Just as in Borůvka's algorithm, each component of F has a "leader" node. An edge joins two components of F if and only if the two endpoints have different leaders. But unlike Borůvka's algorithm, we do not recompute leaders from scratch every time we add an edge. Instead, when two components are joined, the two leaders duke it out in a nationally-televised[2] no-holds-barred steel-cage grudge match. One of the two emerges victorious as the leader of the new larger component.

More formally, we need a solution to the so-called UNION-FIND problem. We want to maintain a collection of disjoint sets, initially each with one element, so that we can merge any two sets together (UNION) and identify which set contains a given item (FIND). Next week, we will see a data structure that requires only $O(\log^* n)$ amortized time per UNION or FIND, where n is the total size of the sets. In our case, the sets are components of F and $n = V$. Kruskal's algorithm performs $O(E)$ FIND operations, two for each edge in the graph, and $O(V)$ UNION operations, one for each edge in the minimum spanning tree. So if the edges of the graph are already sorted, Kruskal's algorithm takes $O(E \log^* V)$ time, which is better than Jarník's algorithm when the graph is extremely sparse and always better than Borůvka's algorithm.

However, we are usually *not* given the edges in order, so we need $O(E \log E) = O(E \log V)$ additional time jsut to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is $O(E \log V)$, exactly the same as Borůvka's algorithm.

[2]Only $49.95 on Pay-Per-View!