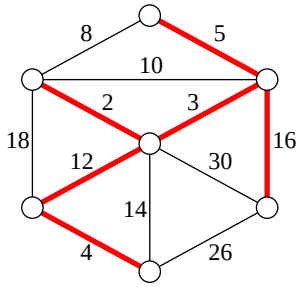
# 13 Minimum Spanning Trees (October 29)

## 13.1 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 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)
```

## 13.2 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 certain edges between them. When the algorithm halts, $F$ consists of a single $n$-node tree, which must be the minimum spanning tree. Obviously, we have to be careful about *which* edges we add to the evolving forest, since not every edge is in the eventual minimum spanning tree.

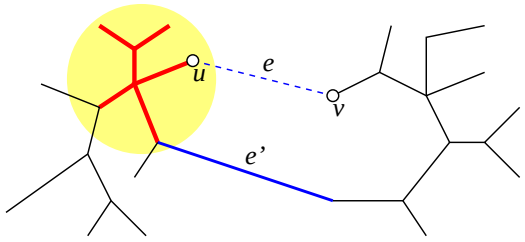
The intermediate spanning forest $F$ induces two special types of edges. An edge is *useless* if it is not an edge of $F$, but both its endpoints are in the same component of $F$. For each component of $F$, we associate a *safe* edge—the minimum-weight edge with exactly one endpoint in that component.[1] Different components might or might not have different safe edges. Some edges are neither safe nor useless—we call these edges *undecided*.

All minimum spanning tree algorithms are based on two simple observations.

**Lemma 1.** *The minimum spanning tree contains every safe edge and no useless edges.*

**Proof:** Let $T$ be the minimum spanning tree. Suppose $F$ has a 'bad' component whose safe edge $e = (u, v)$ is not in $T$. Since $T$ is connected, it contains a unique path from $u$ to $v$, and at least one edge $e'$ on this path has exactly one endpoint in the bad component. Removing $e'$ from the minimum spanning tree and adding $e$ gives us a new spanning tree. Since $e$ is the bad component's safe edge, we have $w(e') > w(e)$, so the the new spanning tree has smaller total weight than $T$. But this is impossible—$T$ is the *minimum* spanning tree. So $T$ must contain every safe edge.

Adding any useless edge to $F$ would introduce a cycle. □




Proving that every safe edge is in the minimum spanning tree. The 'bad' component of $F$ is highlighted.

So our generic minimum spanning tree algorithm repeatedly adds one or more safe edges to the evolving forest $F$. Whenever we add new edges to $F$, some undecided edges become safe, and others become useless. To specify a particular algorithm, we must decide which safe edges to add, and how to identify new safe and new useless edges, at each iteration of our generic template.

## 13.3 Borůvka's Algorithm

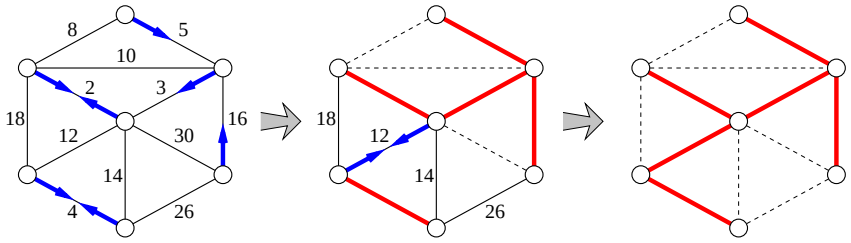
The oldest and possibly simplest minimum spanning tree algorithm was discovered by Borůvka in 1926, long before computers even existed, and practically before the invention of graph theory![2] The algorithm was rediscovered by Choquet in 1938; again by Florek, Lukaziewicz, Perkal, Stienhaus, and Zubrzycki in 1951; and again by Sollin some time in the early 1960s.

The Borůvka/Choquet/Florek/Lukaziewicz/Perkal/Stienhaus/Zubrzycki/Sollin algorithm can be summarized in one line:

[1] This is actually a special case of a more general definition: For any partition of $F$ into two subforests, the minimum-weight edge with one endpoint in each subforest is light. A few minimum spanning tree algorithms require this more general definition, but we won't talk about them here.

[2] 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: Add all the safe edges and recurse.




Borůvka's algorithm run on the example graph. Thick edges are in $F$.
Arrows point along each component's safe edge. Dashed edges are useless.

At the beginning of each phase of the Borůvka algorithm, each component elects an arbitrary 'leader' node. The simplest way to hold these elections is a depth-first search of $F$; the first node we visit in any component is that component's leader. Once the leaders are elected, we find the safe edges for each component, essentially by brute force. Finally, we add these safe edges to $F$.

```
BORŮVKA(V, E):
  F = (V, ∅)
  while F has more than one component
      choose leaders using DFS
      FINDSAFEEDGES(V, E)
      for each leader v̄
          add safe(v̄) to F
```

```
FINDSAFEEDGES(V, E):
  for each leader v̄
      safe(v̄) ← ∞
  for each edge (u, v) ∈ E
      ū ← leader(u)
      v̄ ← leader(v)
      if ū ≠ v̄
          if w(u, v) < w(safe(ū))
              safe(ū) ← (u, v)
          if w(u, v) < w(safe(v̄))
              safe(v̄) ← (u, v)
```

Each call to FINDSAFEEDGES takes $O(E)$ time, since it examines every edge. Since the graph is connected, it has at most $E + 1$ vertices. Thus, 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, the while loop iterates $O(\log V)$ times. Thus, the overall running time of Borůvka's algorithm is $\boxed{O(E \log V)}$.

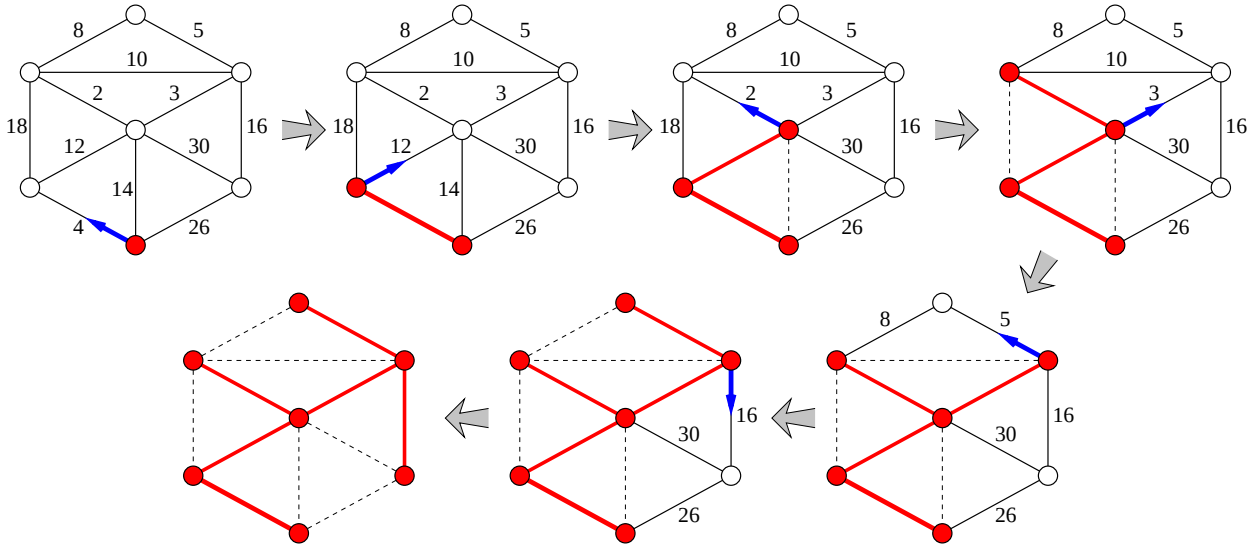
Despite its relatively obscure origin, early algorithms researchers were aware of Borůvka's algorithm, but dismissed it as being "too complicated"! As a result, despite its simplicity and efficiency, Borůvka's algorithm is rarely mentioned in algorithms and data structures textbooks.

## 13.4 Jarník's ('Prim's') Algorithm

The next oldest minimum spanning tree algorithm was discovered by the Vojtěch Jarník in 1930, but it is usually called Prim's algorithm. Prim independently rediscovered the algorithm in 1956 and gave a much more detailed description than Jarník. The algorithm was rediscovered again in 1958 by Dijkstra, but he already had an algorithm named after him. Such is fame in academia.

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:

JARNÍK: Find $T$'s safe edge and add it to $T$.




Jarník's algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in $T$, an arrow points along $T$'s safe edge, and dashed edges are useless.

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 $T$. If not, we add the edge to $T$ 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 the graph traversal algorithm visits each vertex only once. Rather than keeping edges in the heap, we can keep a heap of vertices, where the key of each vertex $v$ is the length of the minimum-weight edge between $v$ and $T$ (or $\infty$ if there is no such edge). Each time we add a new edge to $T$, we may need to decrease the key of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JARNÍKINIT initializes the vertex heap. JARNÍKLOOP is the main algorithm. The input consists of the vertices and edges of the graph, plus the start vertex $s$.

```
JARNÍK(V, E, s):
  JARNÍKINIT(V, E, s)
  JARNÍKLOOP(V, E, s)
```

```
JARNÍKINIT(V, E, s):
  for each vertex v ∈ V \ {s}
      if (v, s) ∈ E
          edge(v) ← (v, s)
          key(v) ← w(v, s)
      else
          edge(v) ← NULL
          key(v) ← ∞
      INSERT(v)
```

```
JARNÍKLOOP(V, E, s):
  T ← ({s}, ∅)
  for i ← 1 to |V| − 1
      v ← EXTRACTMIN
      add v and edge(v) to T
      for each edge (u, v) ∈ E
          if u ∉ T and key(u) > w(u, v)
              edge(u) ← (u, v)
              DECREASEKEY(u, w(u, v))
```

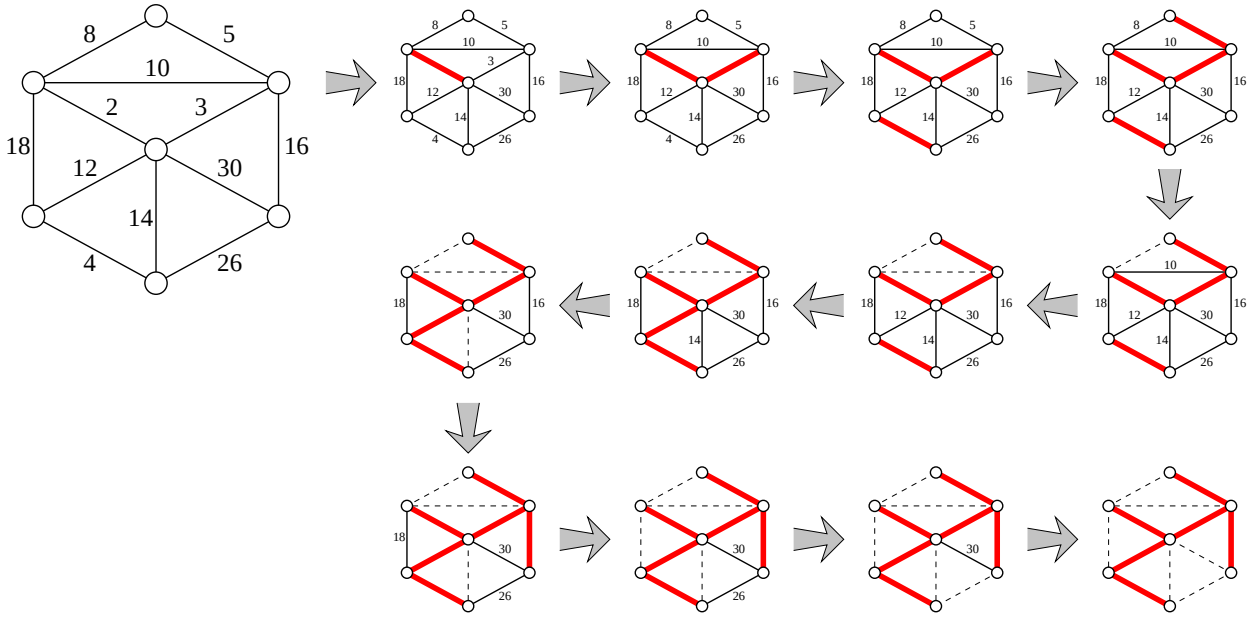
The running time of JARNÍK is dominated by the cost of the heap operations INSERT, EXTRACTMIN, and DECREASEKEY. INSERT and EXTRACTMIN are each called $O(V)$ times once for each vertex except $s$, and DECREASEKEY is called $O(E)$ times, at most twice for each edge. If we use a Fibonacci heap, the amortized costs of INSERT and DECREASEKEY is $O(1)$, and the amortized cost of EXTRACTMIN is $O(\log n)$. Thus, the overall running time of JARNÍK is $\boxed{O(E + V \log V)}$. This is faster than Borůvka's algorithm unless $E = O(V)$.

## 13.5 Kruskal's Algorithm

The last minimum spanning tree algorithm I'll discuss was discovered by Kruskal in 1956.

> KRUSKAL: Scan all edges in increasing weight order; if an edge is safe, add it to $F$.

Since we examine the edges in order from lightest to heaviest, any edge we examine is safe if and only if its endpoints are in different components of the forest $F$. To prove this, suppose the edge $e$ joins two components $A$ and $B$ but is not safe. Then there would be a lighter edge $e'$ with exactly one endpoint in $A$. But this is impossible, because (inductively) any previously examined edge has both endpoints in the same component of $F$.




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

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 no-holds-barred steel-cage grudge match.[3] One of the two emerges victorious as the leader of the new larger component. More formally, we will use our earlier algorithms for the UNION-FIND problem, where the vertices are the elements and the components of $F$ are the sets. Here's a more formal description of the algorithm:

[3]Live at the Assembly Hall! Only $49.95 on Pay-Per-View!

```
KRUSKAL(V, E):
  sort E by wieght
  F ← ∅
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      (u, v) ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add (u, v) to F
  return F
```

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. Using union-by-rank and path compression allows us to perform each UNION or FIND in $O(\alpha(E, V))$ time, where $\alpha$ is the not-quite-constant inverse-Ackerman function. So ignoring the cost of sorting the edges, the running time of this algorithm is $O(E\alpha(E, V))$.

We need $O(E \log E) = O(E \log V)$ additional time just 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 $\boxed{O(E \log V)}$, exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.