

We must all hang together, gentlemen, or else we shall most assuredly hang separately.

— Benjamin Franklin, at the signing of the Declaration of Independence (July 4, 1776)

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde

A ship in port is safe, but that is not what ships are for.

— Rear Admiral Grace Murray Hopper

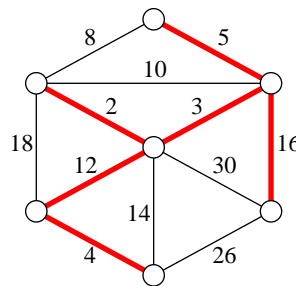
15 Minimum Spanning Trees

15.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w: E \rightarrow \mathbb{R}$ 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  $(i, j)$ 
   $\langle\langle$  if  $\max(i, j) < \max(k, l)$   $\rangle\rangle$  return  $(k, l)$ 

```

15.2 The Only Minimum Spanning Tree Algorithm

There are several different methods for computing 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 minimum spanning tree 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 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 minimum spanning tree.

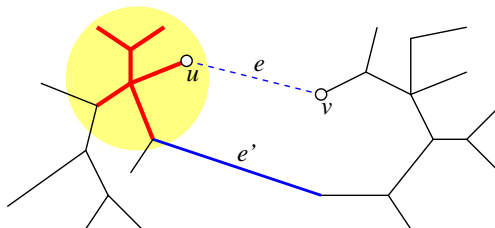
The intermediate spanning forest F induces two special types of edges. An edge is *useless* if it is not an edge of F , but both its endpoints are in the same component of F . For each component of F , we associate a *safe* edge—the minimum-weight edge with exactly one endpoint in that component. Different 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.*¹

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 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.

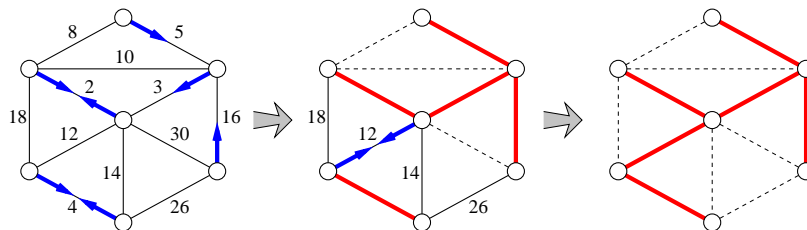
¹This lemma is actually a special case of two more general theorems. First, for any partition of the vertices of G into two disjoint subsets, the minimum-weight edge with one endpoint in each subset is in the minimum spanning tree. Second, the maximum-weight edge in any cycle in G is **not** in the minimum spanning tree.

15.3 Borůvka’s Algorithm

The oldest and arguably 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, Łukaziewicz, Perkal, Stienhaus, and Zubrzycki in 1951; and again by Sollin some time in the early 1960s. Because Sollin was the only Western computer scientist in this list—Choquet was a civil engineer; Florek and his co-authors were anthropologists—this is often called ‘Sollin’s algorithm’, especially in the parallel computing literature.

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

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( $V, E$ ):
   $F = (V, \emptyset)$ 
  while  $F$  has more than one component
    choose leaders using DFS
    FINDSAFEEDGES( $V, E$ )
    for each leader  $\bar{v}$ 
      add safe( $\bar{v}$ ) to  $F$ 
    
```

```

FINDSAFEEDGES( $V, E$ ):
  for each leader  $\bar{v}$ 
    safe( $\bar{v}$ )  $\leftarrow \infty$ 
  for each edge  $(u, v) \in E$ 
     $\bar{u} \leftarrow \text{leader}(u)$ 
     $\bar{v} \leftarrow \text{leader}(v)$ 
    if  $\bar{u} \neq \bar{v}$ 
      if  $w(u, v) < w(\text{safe}(\bar{u}))$ 
        safe( $\bar{u}$ )  $\leftarrow (u, v)$ 
      if  $w(u, v) < w(\text{safe}(\bar{v}))$ 
        safe( $\bar{v}$ )  $\leftarrow (u, v)$ 
    
```

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 $O(E \log V)$.

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. On the other hand, more

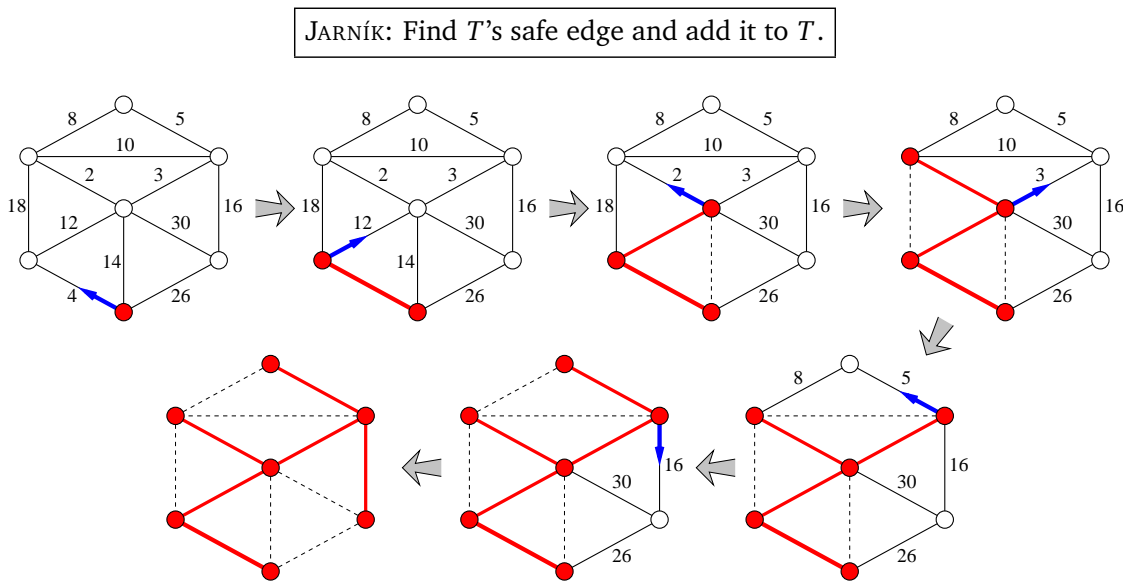
²Leonard Euler published the first graph theory result, his famous theorem about the bridges of Königsburg, in 1736. However, the first textbook on graph theory, written by Dénes König, was not published until 1936.

recent algorithms to compute minimum spanning trees are all generalizations of Borůvka's algorithm, not the other two classical algorithms described next.

15.4 Jarník's ('Prim's') Algorithm

The next oldest minimum spanning tree algorithm was first described by the Polish mathematician Vojtěch Jarník in a 1929 letter to Borůvka. The algorithm was independently rediscovered by Kruskal in 1956, by Prim in 1957, by Loberman and Weinberger in 1957, and finally by Dijkstra in 1958. Prim, Loberman, Weinberger, and Dijkstra all (eventually) knew of and even cited Kruskal's paper, but since Kruskal also described two other minimum-spanning-tree algorithms in the same paper, *this* algorithm is usually (incorrectly) called 'Prim's algorithm', or sometimes even 'the Prim/Dijkstra algorithm', even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

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'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 ∞ 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 $v \in V \setminus \{s\}$
 if $(v, s) \in E$
 $\text{edge}(v) \leftarrow (v, s)$
 $\text{key}(v) \leftarrow w(v, s)$
 else
 $\text{edge}(v) \leftarrow \text{NULL}$
 $\text{key}(v) \leftarrow \infty$
 INSERT(v)

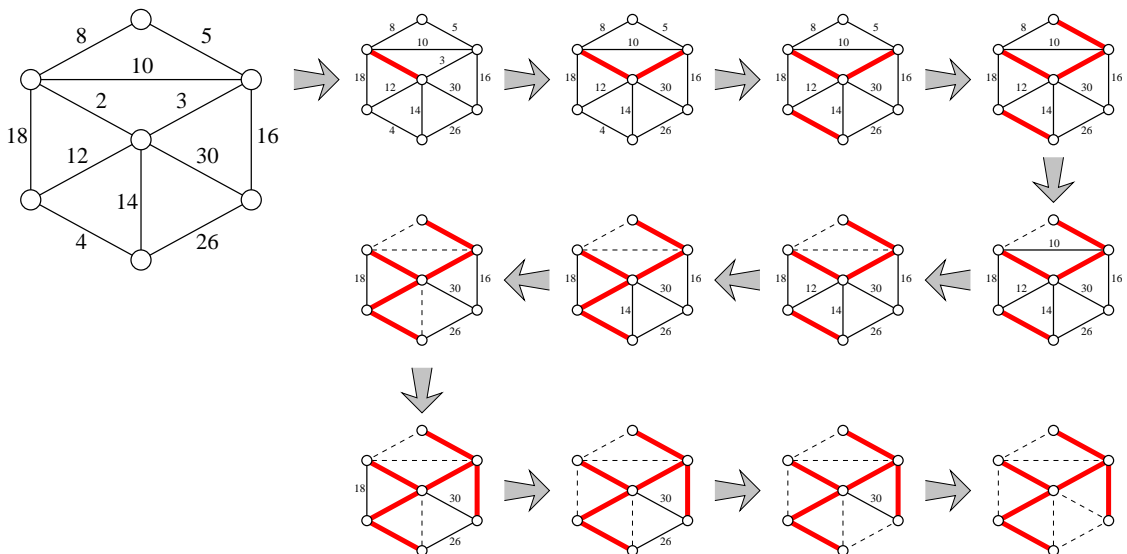
JARNÍKLOOP(V, E, s):
 $T \leftarrow (\{s\}, \emptyset)$
 for $i \leftarrow 1$ to $|V| - 1$
 $v \leftarrow \text{EXTRACTMIN}$
 add v and $\text{edge}(v)$ to T
 for each edge $(u, v) \in E$
 if $u \notin T$ and $\text{key}(u) > w(u, v)$
 $\text{edge}(u) \leftarrow (u, v)$
 DECREASEKEY($u, w(u, v)$)

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 standard binary heap, each of these operations requires $O(\log V)$ time, so the overall running time of JARNÍK is $O((V + E) \log V) = O(E \log V)$. The running time can be improved to $O(E + V \log V)$ using a data structure called a *Fibonacci heap*, which supports INSERT and DECREASEKEY in constant *amortized* time; this is faster than Borůvka's algorithm unless $E = O(V)$.

15.5 Kruskal's Algorithm

The last minimum spanning tree algorithm I'll discuss was first described by Kruskal in 1956, in the same paper where he rediscovered Jarnik's algorithm. Kruskal was motivated by 'a typewritten translation (of obscure origin)' of Borůvka's original paper, claiming that Borůvka's algorithm was 'unnecessarily elaborate'.³ This algorithm was also rediscovered in 1957 by Loberman and Weinberger, but somehow avoided being renamed after them.

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



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

³To be fair, Borůvka's original paper was unnecessarily elaborate, but in his followup paper, also published in 1927, simplified his algorithm to its current modern form. Kruskal was apparently unaware of Borůvka's second paper. Stupid Iron Curtain.

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 .

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.⁴ 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:

```

KRUSKAL( $V, E$ ):
  sort  $E$  by weight
   $F \leftarrow \emptyset$ 
  for each vertex  $v \in V$ 
    MAKESET( $v$ )
  for  $i \leftarrow 1$  to  $|E|$ 
     $(u, v) \leftarrow i$ th lightest edge in  $E$ 
    if FIND( $u$ )  $\neq$  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 α 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 $O(E \log V)$, exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

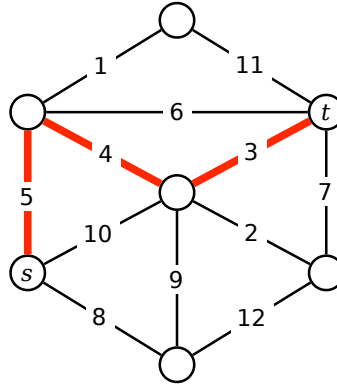
⁴Live at the Assembly Hall! Only \$49.95 on Pay-Per-View!

Exercises

1. Most classical minimum-spanning-tree algorithms use the notions of ‘safe’ and ‘useless’ edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is *dangerous* if it is the longest edge in some cycle in G , and *useful* if it does not lie in any cycle in G .
 - (a) Prove that the minimum spanning tree of G contains every useful edge.
 - (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.
 - (c) Describe and analyze an efficient implementation of the “anti-Kruskal” MST algorithm: Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G . [*Hint: It won't be as fast as Kruskal's algorithm.*]
2. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.
 - (a) Prove that for any partition of the vertices V into two subsets, the minimum-weight edge with one endpoint in each subset is in the minimum spanning tree of G .
 - (b) Prove that the maximum-weight edge in any cycle of G is *not* in the minimum spanning tree of G .
 - (c) Prove or disprove: The minimum spanning tree of G includes the minimum-weighted edge in *every* cycle in G .
3. Throughout this lecture note, we assumed that no two edges in the input graph have equal weights, which implies that the minimum spanning tree is unique. In fact, a weaker condition on the edge weights implies MST uniqueness.
 - (a) Describe an edge-weighted graph that has a unique minimum spanning tree, even though two edges have equal weights.
 - (b) Prove that an edge-weighted graph G has a *unique* minimum spanning tree if and only if the following conditions hold:
 - For any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique.
 - The maximum-weight edge in any cycle of G is unique.
 - (c) Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.
4. Suppose you are given a graph G with weighted edges and a minimum spanning tree T of G .
 - (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.
 - (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. [*Hint: Consider the cases $e \in T$ and $e \notin T$ separately.*]

5. Consider a path between two vertices s and t in an undirected weighted graph G . The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t . (If there are no paths from s to t , the bottleneck distance between s and t is ∞ .)



The bottleneck distance between s and t is 5.

Describe and analyze an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

6. (a) Describe and analyze an algorithm to find the *second smallest spanning tree* of a given graph G , that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.
- (b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k , the k spanning trees of G with smallest weight.
7. We say that a graph $G = (V, E)$ is *dense* if $E = \Theta(V^2)$. Describe a modification of Jarník's minimum-spanning tree algorithm that runs in $O(V^2)$ time (independent of E) when the input graph is dense, using only simple data structures (and in particular, *without* using a Fibonacci heap).
8. (a) Prove that the minimum spanning tree of a graph is also a spanning tree whose maximum-weight edge is minimal.
- (b) Describe an algorithm to compute a spanning tree whose maximum-weight edge is minimal, in $O(V + E)$ time. [Hint: Start by computing the median of the edge weights.]
9. Consider an algorithm that first performs k passes of Borůvka's algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.
- (a) What is the running time of this hybrid algorithm, as a function of V , E , and k ?
- (b) For which value of k is this running time minimized? What is the resulting running time?
10. Describe an algorithm to compute the minimum spanning tree of an n -vertex *planar* graph in $O(n)$ time. [Hint: Contracting an edge in a planar graph yields another planar graph. Any planar graph with n vertices has at most $3n - 6$ edges.]