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)

I remember seeking advice from someone—who could it have been?—about whether this work was worth submitting for publication; the reasoning it uses is so very simple.... Fortunately he advised me to go ahead, and many years passed before another of my publications became as well-known as this very simple one.

— Joseph Kruskal, describing his shortest-spanning-subtree algorithm (1997)

Clean ALL the things!

— Allie Brosh, "This is Why I'll Never be an Adult", *Hyperbole and a Half*, June 17, 2010.

CHAPTER 7

Minimum Spanning Trees

Status: Beta.

7.1 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 real *weight* $w(e)$ to each edge e, which may be positive, negative, or zero. Our task is to find the ***minimum spanning tree*** of G, that is, the spanning tree T that minimizes 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 $V - 1$.

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

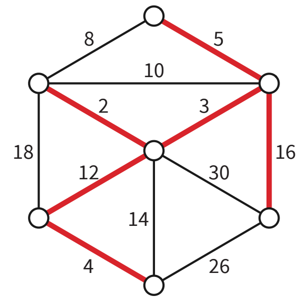


Figure 7.1. A weighted graph and its minimum spanning tree.

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.

<u>ShorterEdge(i, j, k, l)</u>	
if $w(i, j) < w(k, l)$	then return (i, j)
if $w(i, j) > w(k, l)$	then return (k, l)
if $\min(i, j) < \min(k, l)$	then return (i, j)
if $\min(i, j) > \min(k, l)$	then return (k, l)
if $\max(i, j) < \max(k, l)$	then return (i, j)
⟨⟨*if max(i,j) < max(k,l)*⟩⟩	return (k, l)

7.2 The Only Minimum Spanning Tree Algorithm

There are several different methods for computing minimum spanning trees, but almost all of them are 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. The first observation was proved by Robert Prim in 1957, and the second is immediate.

Lemma 1 (Prim). *The minimum spanning tree contains every safe edge.*

Proof: In fact we prove the following stronger statement: For *any* subset S of the vertices of G, the minimum spanning tree of G contains the minimum-weight edge with exactly one endpoint in S. We prove this claim using a greedy exchange argument.

Let S be an arbitrary subset of vertices of G; let e be the lightest edge with exactly one endpoint in S; and let T be an arbitrary spanning tree that does *not* contain e. Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S; let e' be *any* such edge. Because T is acyclic, removing e' from T yields a spanning *forest* with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree $T' = T - e' + e$. The definition of e implies $w(e') > w(e)$, which implies that T' has smaller total weight than T. We conclude that T is not the minimum spanning tree, which completes the proof. □

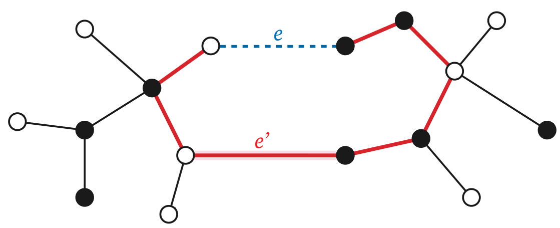


Figure 7.2. Proving that every safe edge is in the minimum spanning tree. Black vertices are in the subset S.

Lemma 2. *The minimum spanning tree contains no useless edge.*

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

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 we must describe how to identify new safe and new useless edges, at each iteration of our generic template.

7.3 Borůvka's Algorithm

The oldest and arguably simplest minimum spanning tree algorithm was discovered by the Czech mathematician Otakar Borůvka in 1926, about a year after Jindřich Saxel asked him how to construct an electrical network connecting several cities using the least amount of wire.[1] The algorithm was rediscovered by Gustav Choquet in 1938, rediscovered again by a team of Polish mathematicians led by Józef Łukaszewicz in 1951, and rediscovered again by George Sollin in 1961. Although Sollin never published his rediscovery, it was carefully described and credited in one of the first textbooks on graph algorithms; as a result, this is sometimes called “Sollin’s algorithm”.

The Borůvka/Choquet/Florek-Łukaziewicz-Perkal-Stienhaus-Zubrzycki/Prim/Sollin/Brosh[2] algorithm can be summarized in one line:

$$\boxed{\text{Borůvka: Add } \textit{\textbf{ALL}} \text{ the safe edges and recurse.}}$$

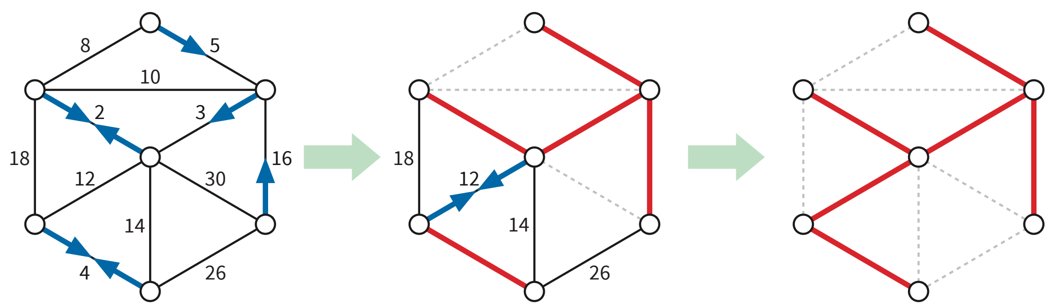


Figure 7.3. Borůvka’s algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component’s safe edge.

We can find all the safe edge in the graph in $O(E)$ time as follows. First, we count the components of F using whatever-first search, using the standard wrapper function. As we count, we label every vertex with its component number; that is, every vertex in the first traversed component gets label 1, every vertex in the second component gets label 2, and so on.

If F has only one component, we’re done. Otherwise, we compute an array $\mathit{safe}[1..V]$ of safe edges, where $\mathit{safe}[i]$ is the minimum-weight edge with one endpoint in the ith component (or a sentinel value Null if there are less than i components). To compute this array, we consider each edge uv in the input graph G by brute force. If u and v have the same label, then uv is useless; otherwise, we compare the weight of uv to the weights of $\mathit{safe}[\mathit{label}(u)]$ and $\mathit{safe}[\mathit{label}(v)]$ and update the array entries if necessary. Finally, we add each edge $\mathit{safe}[i]$ to our forest F.

[1]Saxel was an employee of the West Moravian Power Company, described by Borůvka as “very talented and hard-working”, who was later executed by the Nazis as a person of Jewish descent.

[2]Go read everything in *Hyperbole and a Half*. And then go buy the book. And an extra copy for your cat.

```
AddAllSafeEdges(E, F, count):
  for i ← 1 to count
      safe[i] ← Null
  for each edge uv ∈ E
      if label(u) ≠ label(v)
          if safe[label(u)] = Null  or  w(uv) < w(safe[label(u)])
              safe[label(u)] ← uv
          if safe[label(v)] = Null  or  w(uv) < w(safe[label(v)])
              safe[label(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```

Finally, here is the main algorithm:

```
Borůvka(V, E):
  F = (V, ∅)
  count ← CountAndLabel(F)
  while count > 1
      AddAllSafeEdges(E, F, count)
      count ← CountAndLabel(F)
  return F
```

Each call to CountAndLabel requires $O(V)$ time, because the forest F has at most $V-1$ edges. Assuming the graph is represented by an adjacency list, the rest of each iteration of the main while loop requires $O(E)$ time, because we spend constant time on each edge. Because the graph is connected, we have $V \le E+1$, so each iteration of the while loop takes $O(E)$ time.

Each iteration 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 F initially has V components, the while loop iterates at most $O(\log V)$ times. Thus, the overall running time of Borůvka's algorithm is $\boldsymbol{O(E \log V)}$.

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, Borůvka's algorithm has several distinct advantages over other classical MST algorithms.

- Borůvka's algorithm often runs faster than the $O(E \log V)$ worst-case running time. In arbitrary graphs, the number of components in F can drop by significantly more than a factor of 2 in a single iteration, reducing the number of iterations below the worst-case $\lceil \log_2 V \rceil$. A slight reformulation of Borůvka's algorithm (actually closer to Borůvka's original presentation) actually runs in $O(E)$ time for a broad class of interesting graphs, including graphs that can be drawn in the plane without edge crossings. In contrast, the time analysis for the other two algorithms applies to *all* graphs.

- Borůvka's algorithm allows for significant parallelism; in each iteration, each component of F can be handled in a separate independent thread. This implicit parallelism allows for even faster performance on multicore or distributed systems. In contrast, the other two classical MST algorithms are intrinsically serial.
- There are several more recent minimum-spanning-tree algorithms that are faster even in the worst case than the classical algorithms described here. *All* of these faster algorithms are generalizations of Borůvka's algorithm.

In short, if you ever need to implement a minimum-spanning-tree algorithm, use Borůvka. On the other hand, if you want to *prove things about* minimum spanning trees effectively, you really need to know the next two algorithms as well.

7.4 Jarník's ("Prim's") Algorithm

The next oldest minimum spanning tree algorithm was first described by the Czech mathematician Vojtěch Jarník in a 1929 letter to Borůvka; Jarník published his discovery the following year. The algorithm was independently rediscovered by Joseph Kruskal in 1956, (arguably) by Robert Prim in 1957, and finally by Edsger Dijkstra in 1958. Both Prim and Dijkstra (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 called "Prim's algorithm", or sometimes "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: Repeatedly add T's safe edge to T.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a priority queue. When we pull the minimum-weight edge out of the priority queue, 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 priority queue. In other words, Jarník's algorithm is another instance of the generic graph traversal algorithm we saw last time, using a priority queue as the "bag"! If we implement the algorithm this way, the algorithm runs in $O(E \log E) = \boldsymbol{O(E \log V)}$ time.

♥Improving Jarník's Algorithm

We can improve Jarník's algorithm using a more advanced priority queue data structure called a ***Fibonacci heap***, first described by Michael Fredman and Robert Tarjan in 1984. Fibonacci heaps support the standard priority queue operations INSERT, EXTRACTMIN, and DECREASEKEY. However, unlike standard binary heaps, which require $O(\log n)$ time

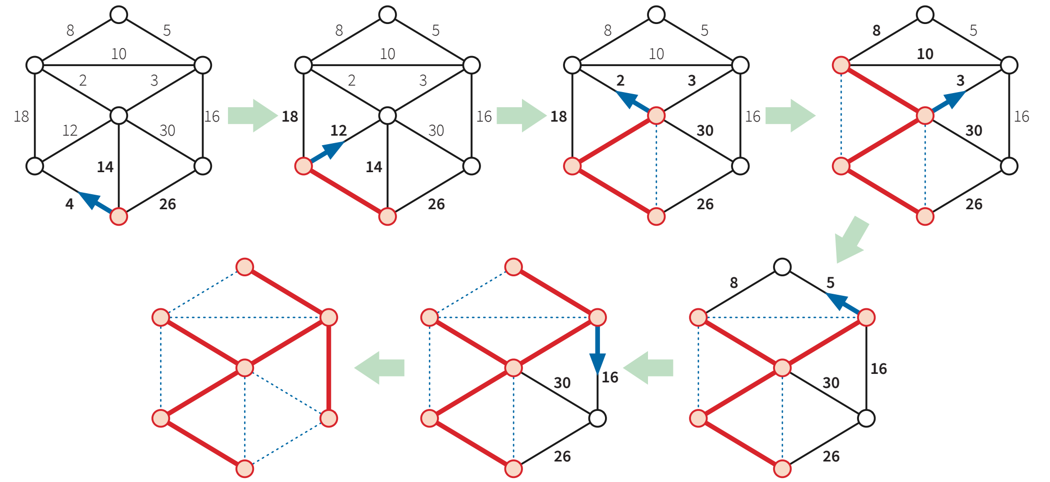


Figure 7.4. Jarník's algorithm run on the example graph, starting with the bottom vertex. At each stage, thick red edges are in T, an arrow points along T's safe edge; and dashed edges are useless.

for every operation, Fibonacci heaps support INSERT and DECREASEKEY in constant *amortized* time. The amortized cost of EXTRACTMIN is still $O(\log n)$.

To apply this faster data structure, we keep *vertices* in the priority queue instead of edge, where the key for each vertex v is either the minimum-weight edge between v and the evolving tree T, or ∞ if there is no such edge. We can INSERT all the vertices into the priority queue at the beginning of the algorithm; then, whenever we add a new edge to T, we may need to decrease the keys of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JARNÍKINIT initializes the priority queue; JARNÍKLOOP is the main algorithm. The input consists of the vertices and edges of the graph, plus the start vertex s. For each vertex v, we maintain both its key $key(v)$ and the incident edge $edge(v)$ such that $w(edge(v)) = key(v)$.

```
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 neighbor u of v
          if u ∉ T and key(u) > w(uv)
              edge(u) ← uv
              DECREASEKEY(u, w(uv))
```

The operations 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.

Thus, if we use a Fibonacci heap, the improved algorithm runs in $\boldsymbol{O(E + V \log V)}$ ***time***, which is faster than Borůvka's algorithm unless $E = O(V)$.

In practice, however, this improvement is rarely faster than the naive implementation using a binary heap, unless the graph is extremely large and dense. The Fibonacci heap algorithms are quite complex, and the hidden constants in both the running time and space are significant—not outrageous, but certainly bigger than the hidden constant 1 in the $O(\log n)$ time bound for binary heap operations.

7.5 Kruskal's Algorithm

The last minimum spanning tree algorithm we'll consider was first described by Joseph 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 that had been "floating around" the Princeton math department. Kruskal found Borůvka's algorithm "unnecessarily elaborate".[3] The same algorithm was 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.

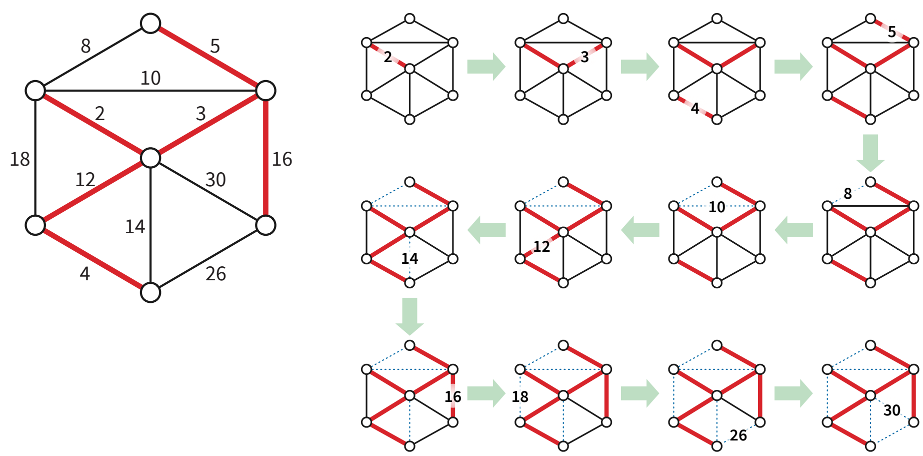


Figure 7.5. Kruskal's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless.

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

[3]To be fair, Borůvka's first paper *was* unnecessarily elaborate, in part because it was written for mathematicians in the formal language of (linear) algebra, rather than in the language of graphs. Borůvka's followup paper, also published in 1927 but in an electrotechnical journal, was written in plain language for a much broader audience, essentially its current modern form. Kruskal was apparently unaware of Borůvka's second paper. Stupid Iron Curtain.

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 increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add uv 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 $\boldsymbol{O(E \log V)}$, exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Most classical minimum-spanning-tree algorithms use the notions of "safe" and "useless" edges described in the text, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an

[4] Live at the Assembly Hall! Only \$49.95 on Pay-Per-View![5]

[5] Is Pay-Per-View still a thing?

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 following algorithm, first described by Kruskal in the same 1956 paper where he proposed "Kruskal's 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 usual 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 disjoint subsets, the minimum spanning tree of G includes the minimum-weight edge with one endpoint in each subset.

   (b) Prove that for any cycle in G, the minimum spanning tree of G *excludes* the maximum-weight edge in that cycle.

   (c) Prove or disprove: The minimum spanning tree of G includes the minimum-weight edge in *every* cycle in G.

3. Throughout this chapter, 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. (a) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.

   (b) A *feedback edge set* of an undirected graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of of a given edge-weighted graph.

5. Suppose we are given both an undirected 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.]*

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. Consider the following variant of Borůvka's algorithm. Instead of counting and labeling components of F to find safe edges, we use a standard disjoint set data structure. Each component of F is represented by an up-tree; each vertex v stores a pointer *parent*(v) to its parent in the up-tree containing v. Each leader vertex $\bar{v}$ also maintains an edge *safe*$(\bar{v})$, which is (eventually) the lightest edge with one endpoint in $\bar{v}$'s component of F.

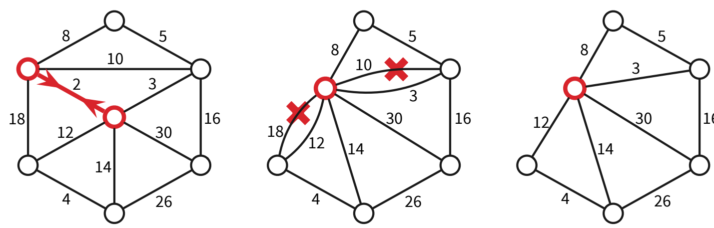
```
BORŮVKA(V, E):
  F = ∅
  for each vertex v ∈ V
      parent(v) ← v
  while FINDSAFEEDGES(V, E)
      ADDSAFEEDGES(V, E, F)
  return F
```

```
FindSafeEdges(V, E):
  for each vertex v ∈ V
      safe(v) ← Null
  found ← False
  for each edge uv ∈ E
      ū ← Find(u); v̄ ← Find(v)
      if ū ≠ v̄
          if w(uv) < w(safe(ū))
              safe(ū) ← uv
          if w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← True
  return done
```

```
AddSafeEdges(V, E, F):
  for each vertex v ∈ V
      if safe(v) ≠ Null
          xy ← safe(v)
          if Find(x) ≠ Find(y)
              Union(x, y)
              add xy to F
```

Prove that if Find uses path compression, then each call to FindSafeEdges and AddSafeEdges requires only $O(V+E)$ time. *[Hint: It doesn't matter how Union is implemented! What is the depth of the up-trees when FindSafeEdges ends?]*

9. Minimum-spanning tree algorithms are often formulated using an operation called *edge contraction*. To contract the edge uv, we insert a new node, redirect any edge incident to u or v (except uv) to this new node, and then delete u and v. After contraction, there may be multiple parallel edges between the new node and other nodes in the graph; we remove all but the lightest edge between any two nodes.



Contracting an edge and removing redundant parallel edges.

The three classical minimum-spanning tree algorithms can be expressed cleanly in terms of contraction as follows. All three algorithms start by making a clean copy G' of the input graph G and then repeatedly contract safe edges in G'; the minimum spanning tree consists of the contracted edges.

- Borůvka: Mark the lightest edge leaving each vertex, contract all marked edges, and recurse.
- Jarník: Repeatedly contract the lightest edge incident to some fixed root vertex.
- Kruskal: Repeatedly contract the lightest edge in the graph.

(a) Describe an algorithm to execute a single pass of Borůvka's contraction algorithm in $O(V+E)$ time. The input graph is represented in an adjacency list.

(b) Consider an algorithm that first performs k passes of Borůvka's contraction algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.

i. What is the running time of this hybrid algorithm, as a function of V, E, and k?

ii. For which value of k is this running time minimized? What is the resulting running time?

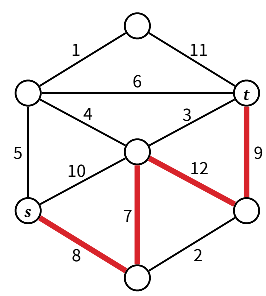
(c) Call a family of graphs *nice* if it has the following properties:

- A nice graph with n vertices has only $O(n)$ edges.
- Contracting an edge of a nice graph yields another nice graph.

For example, planar graphs—graphs that can be drawn in the plane with no crossing edges—are nice. Euler's formula implies that any planar graph with n vertices has at most $3n-6$ edges, and contracting any edge of a planar graph leaves a smaller planar graph.

Prove that Borüvka's contraction algorithm computes the minimum spanning tree of any nice n-vertex graph in $O(n)$ time.

10. Consider a path between two vertices s and t in a undirected weighted graph G. The *width* of this path is the minimum weight of any edge in the path. The *bottleneck distance* between s and t is the width of the widest path from s to t. (If there are no paths from s to t, the bottleneck distance is $-\infty$; on the other hand, the bottleneck distance from s to itself is ∞.)



The bottleneck distance between s and t is 7.

(a) Prove that the *maximum* spanning tree of G contains widest paths between *every* pair of vertices.

(b) Describe an algorithm to solve the following problem in $O(V+E)$ time: Given a undirected weighted graph G, two vertices s and t, and a weight W, is the bottleneck distance between s and t at most W?

(c) Suppose B is the bottleneck distance between s and t.

i. Prove that deleting any edge with weight less than B does not change the bottleneck distance between s and t.

ii. Prove that *contracting* any edge with weight *greater* than B does not change the bottleneck distance between s and t. (If contraction creates parallel edges, delete all but the *heaviest* edge between each pair of nodes.)

(d) Describe an algorithm to compute a minimum-bottleneck path between s and t in $O(V+E)$ time. *[Hint: Start by finding the median-weight edge in G.]*