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.

7

Minimum Spanning Trees

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$ that assigns a real *weight* $w(e)$ to each edge e, which may be positive, negative, or zero. This chapter describes several algorithms to find the ***minimum spanning tree*** of G, that is, the spanning tree T that minimizes the function

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

See Figure 7.1 for an example.

7.1 Distinct Edge Weights

An annoying subtlety in the problem statement is that weighted graphs can have more than one spanning tree with the same minimum weight; in particular, if every edge in G has weight 1, then *every* spanning tree of G is a minimum

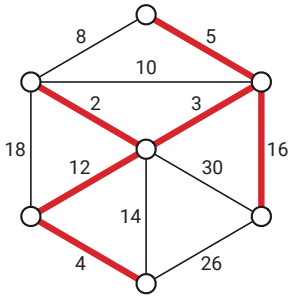


Figure 7.1. A weighted graph and its minimum spanning tree.

spanning tree, with weight $V - 1$. This ambiguity complicates the development of our algorithms; everything would be much simpler if we could simply *assume* that minimum spanning trees are unique.

Fortunately, there is an easy condition that implies the uniqueness we want.

Lemma 7.1. *If all edge weights in a connected graph G are distinct, then G has a unique minimum spanning tree.*[1]

Proof: Let G be an arbitrary connected graph with two minimum spanning trees T and T'; we need to prove that some pair of edges in G have the same weight. The proof is essentially a greedy exchange argument.

Each of our spanning trees must contain an edge that the other tree omits. Let e be a minimum-weight edge in $T \setminus T'$, and let e' be a minimum-weight edge in $T' \setminus T$ (breaking ties arbitrarily). Without loss of generality, suppose $w(e) \le w(e')$.

The subgraph $T' \cup \{e\}$ contains exactly one cycle C, which passes through the edge e. Let e'' be *any* edge of this cycle that is *not* in T. At least one such edge must exist, because T is a tree. (We may or may not have $e'' = e'$.) Because $e \in T$, we immediately have $e'' \neq e$ and therefore $e'' \in T' \setminus T$. It follows that $w(e'') \ge w(e') \ge w(e)$.

Now consider the spanning tree $T'' = T' + e - e''$. (This new tree T'' *might* be equal to T.) We immediately have $w(T'') = w(T') + w(e) - w(e'') \le w(T')$. But T' is a *minimum* spanning tree, so we must have $w(T'') = w(T')$; in other words, T'' is also a minimum spanning tree. We conclude that $w(e) = w(e'')$, which completes the proof. □

If we already have an algorithm that assumes distinct edge weights, we can still run it on graphs where some edges have equal weights, as long as we have a consistent method for breaking ties. One such method uses the following

[1] The converse of this lemma is false; a connected graph with repeated edge weights can still have a unique minimum spanning tree. As a trivial example, suppose G is a tree!

algorithm in place of simple weight comparisons. ShorterEdge takes as input four integers i, j, k, l, representing four (not necessarily distinct) vertices, and decides which of the two edges (i, j) and (k, l) has "smaller" weight. (Because the input graph undirected, the pairs (i, j) and (j, i) represent the same edge.)

```
ShorterEdge(i, j, k, l)
  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)
```

In light of Lemma 7.1 and this tie-breaking rule, we will safely assume for the rest of this chapter that edge weights are *always* distinct, and therefore minimum spanning trees are *always* unique. In particular, we can freely discuss ***the*** minimum spanning tree with no confusion.

7.2 The Only Minimum Spanning Tree Algorithm

There are many algorithms to compute minimum spanning trees, but almost all of them are instances of the following generic strategy. The situation is similar to graph traversal, where several different algorithms are all variants of the generic traversal algorithm whatever-first search.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, which we will call the *intermediate spanning forest*. At all times, F satisfies the following invariant:

> F is a subgraph of the minimum spanning tree of G.

Initially, F consists of V one-vertex trees. The generic algorithm connects trees in F by adding certain edges between them. When the algorithm halts, F consists of a single spanning tree; our invariant implies that this must be the minimum spanning tree of G. Obviously, we have to be careful about *which* edges we add to the evolving forest, because not every edge is in the minimum spanning tree.

At any stage of its evolution, the intermediate spanning forest F induces two special types of edges in the rest of the graph.

- An edge is ***useless*** if it is not an edge of F, but both its endpoints are in the same component of F.
- An edge is ***safe*** if it is the minimum-weight edge with exactly one endpoint in some component of F.

The same edge could be safe for two different components of F. Some edges of $G \setminus F$ 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 (although it is implicit in several earlier algorithms), and the second is immediate.

Lemma 7.2 (Prim). *The minimum spanning tree of G 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. Like the previous lemma, we prove this claim using a greedy exchange argument.

Let S be an arbitrary subset of vertices of G, and let e be the lightest edge with exactly one endpoint in S. (Our assumption that all edge weights are distinct implies that e is unique.) Let T be an arbitrary spanning tree that does *not* contain e; we need to prove that T is *not* the minimum spanning tree of G.

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. Thus, T is not the minimum spanning tree of G, which completes the proof. □

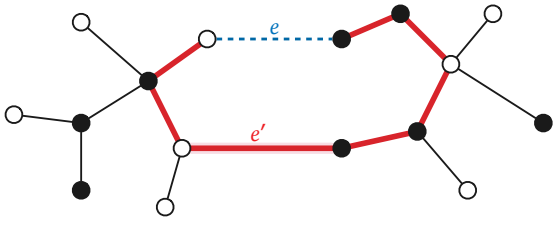


Figure 7.2. Every safe edge is in the minimum spanning tree. Black vertices are in the subset S.

Lemma 7.3. *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 *safe* edges to the evolving forest F. If F is not yet connected, there must be at least one safe edge, because the input graph G is connected. Thus, no matter which safe edges we add in each iteration, our generic algorithm eventually connects F. By induction, Lemma 7.2 implies that the resulting tree is in fact the minimum

spanning tree. Whenever we add new edges to F, some undecided edges may become safe, and other undecided edges may become useless. (Once an edge becomes useless, it stays useless forever.) To fully specify a particular algorithm, we must describe *which* safe edge(s) to add in each iteration, and how to find those edges.

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.[2] 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 algorithm is sometimes called "Sollin's algorithm".

The Borůvka / Choquet / Florek-Łukaziewicz-Perkal-Steinhaus-Zubrzycki / Prim / Sollin / Brosh[3] algorithm can be summarized in one line:

> BORŮVKA: Add ***ALL*** the safe edges and recurse.

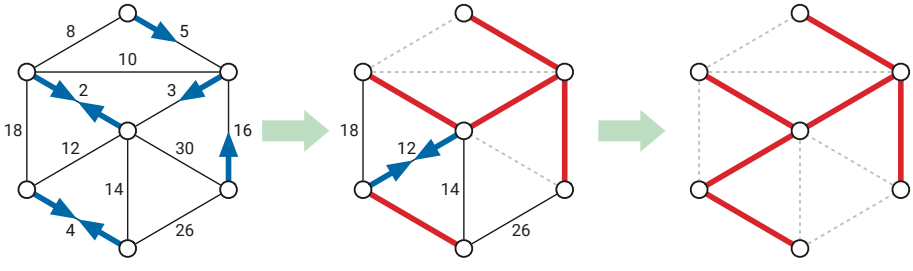


Figure 7.3. Borůvka's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

Here is Borůvka's algorithm in more detail. The algorithm calls the COUNTANDLABEL algorithm from Chapter 5 (on page 204) to count the components of F and label each vertex v with an integer $comp(v)$ indicating its component.

[2] 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.

[3] Go read everything in *Hyperbole and a Half*. And then go buy the book. And an extra copy for your cat. What's that? You don't have a cat? What kind of a monster are you? Go get a cat, and then buy it an extra copy of *Hyperbole and a Half*.

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

It remains only to describe how to identify and add all the safe edges to F. Suppose F has more than one component, since otherwise we're already done. The following subroutine computes an array $safe[1..V]$ of safe edges, where $safe[i]$ is the minimum-weight edge with one endpoint in the ith component of F, by a brute force examination of every edge in G. For each edge uv, if u and v are in the same component, then uv is either useless or already an edge in F. Otherwise, we compare the weight of uv to the weights of $safe[comp(u)]$ and $safe[comp(v)]$ and update the array entries if necessary. Once we have identified all the safe edges, we add each edge $safe[i]$ to F.

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

Each call to COUNTANDLABEL runs in $O(V)$ time, because the forest F has at most $V-1$ edges. *AddAllSafeEdges* runs in $O(V+E)$ time, because we spend constant time on each vertex, each edge of G, and each component of F. Because the input graph is connected, we have $V \le E+1$. It follows that each iteration of the while loop of BORŮVKA takes $O(E)$ time.

Each iteration reduces the number of components of F by at least a factor of two—in the worst case, the components of F coalesce in pairs. Because F initially has V components, the while loop iterates at most $O(\log V)$ times. We conclude that the overall running time of Borůvka's algorithm is $\boldsymbol{O(E \log V)}$.

This is the MST Algorithm You Want

Despite its relatively obscure origin, early Western algorithms researchers were aware of Borůvka's algorithm, but dismissed it as being "too complicated".

As a result, despite its simplicity and efficiency, most algorithms and data structures textbooks unfortunately do not even mention Borůvka's algorithm. This omission is a serious mistake; Borůvka's algorithm has several distinct advantages over other classical MST algorithms.

- Borůvka's algorithm often runs faster than its $O(E \log V)$ worst-case running time. 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.
- Several more recent minimum-spanning-tree algorithms 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, by Harry Loberman and Arnold Weinberger in 1957, and finally by Edsger Dijkstra in 1958. Prim, Lobermand and 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 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 intermediate forest F has only one nontrivial component T; all the other components are isolated vertices. Initially, T consists

of a single 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.

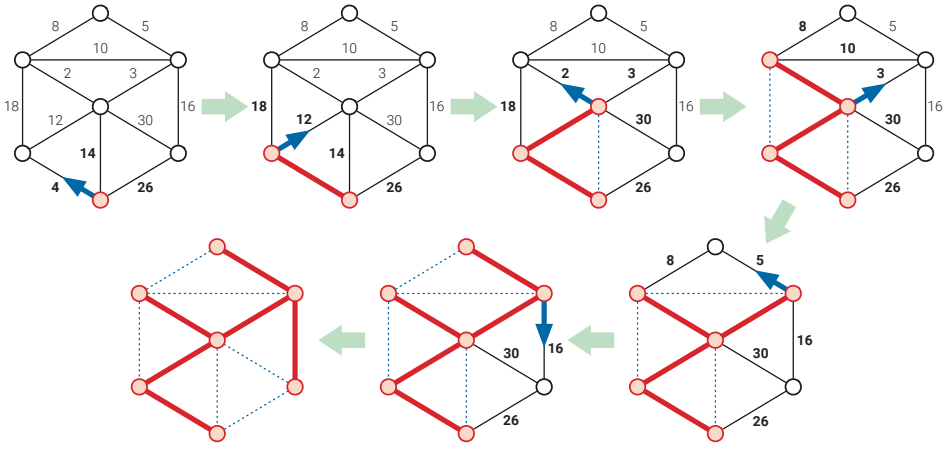


Figure 7.4. Jarník's algorithm run on the example graph, starting with the bottom vertex. At each stage, thick red edges are in T, an arrow points along T's safe edge; and dashed edges are useless.

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 a variant of "best-first search", as described at the end of Chapter 5! If we implement the underlying priority queue using a standard binary heap, Jarník's 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 complex priority queue data structure called a ***Fibonacci heap***, first described by Michael Fredman and Robert Tarjan in 1984. Just like binary heaps, Fibonacci heaps support the standard priority queue operations Insert, ExtractMin, and DecreaseKey. However, unlike standard binary heaps, which require $O(\log n)$ time for every operation, Fibonacci heaps support Insert and DecreaseKey in constant *amortized* time. The amortized cost of ExtractMin is still $O(\log n)$.[4]

[4]Amortized time is an accounting trick that allows us to ignore infrequent fluctuations in the time for a single data structure operation. A Fibonacci heap can execute any intermixed sequence of I Inserts, D DecreaseKeys, and X ExtractMins in $O(I + D + X \log n)$ time, in the worst case. So the *average* Insert and the *average* DecreaseKey each take constant time, and the *average* ExtractMin takes $O(\log n)$ time; however, some individual operations may take

To apply this faster data structure, we keep the *vertices* of G in the priority queue instead of edges, where the priority of 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 priorities 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, along with the start vertex s. For each vertex v, we maintain both its priority $priority(v)$ and the incident edge $edge(v)$ such that $w(edge(v)) = priority(v)$.

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

```
JARNÍKINIT(V, E, s):
  for each vertex v ∈ V \ {s}
      if vs ∈ E
          edge(v) ← vs
          priority(v) ← w(vs)
      else
          edge(v) ← NULL
          priority(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 priority(u) > w(uv)
              edge(u) ← uv
              DECREASEKEY(u, w(uv))
```

Figure 7.5. Jarník's minimum spanning tree algorithm, ready to be used with a Fibonacci heap

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 algo-

longer in the worst case. Amortization uses *statistical* averaging over the sequence of operations; there is no assumption of randomness here, either in the input data or in the algorithm.

rithm. 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".[5] The same algorithm was rediscovered in 1957 by Harry Loberman and Arnold Weinberger, but somehow avoided being renamed after them.

Like our earlier minimum-spanning tree algorithms, Kruskal's algorithm has a memorable one-line description:

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

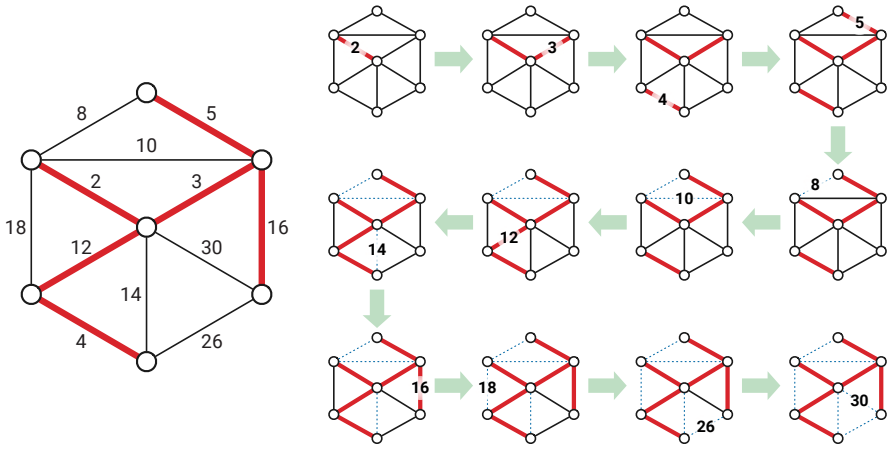


Figure 7.6. Kruskal's algorithm run on the example graph. Thick red edges are in F; thin dashed edges are useless.

The simplest method to scan the edges in increasing weight order is to *sort* the edges by weight, in $O(E \log E)$ time, and then use a simple for-loop over the sorted edge list. As we will see shortly, this preliminary sorting dominates the running time of the algorithm.

Because 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. Suppose we encounter an edge e that joins two components A and B but is not safe. Then there must be a lighter edge e' with exactly one endpoint in A. But this is impossible, because (inductively) every previously examined edge has both endpoints in the same component of F.

Just as in Borůvka's algorithm, each vertex of F needs to "know" which component of F contains it. Unlike Borůvka's algorithm, however, we do

[5] 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 in its current modern form. Kruskal was apparently unaware of Borůvka's second paper. Stupid Iron Curtain.

not recompute all component labels from scratch every time we add an edge. Instead, when two components are joined by an edge, the smaller component inherits the label of the larger component; that is, we traverse the smaller component (via whatever-first search). This traversal requires $O(1)$ time for each vertex in the smaller component. Each time the component label of a vertex changes, the component of F containing that vertex grows by at least a factor of 2; thus, each vertex label changes at most $O(\log V)$ times. It follows that the *total* time spent updating vertex labels is only $O(V \log V)$.

More generally, Kruskal's algorithm maintains a partition of the vertices of G into disjoint subsets (in our case, the components of F), using a data structure that supports the following operations:

- MakeSet(v) — Create a set containing only the vertex v.
- Find(v) — Return an identifier unique to the set containing v.
- Union(u, v) — Replace the sets containing u and v with their union. (This operation decreases the number of sets.)

Here's a complete description of Kruskal's algorithm in terms of these operations:

```
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
```

After the initial sort, the algorithm performs exactly V MakeSet operations (one for each vertex), $2E$ Find operations (two for each edge), and $V-1$ Union operations (one for each edge in the minimum spanning tree). We just described a disjoint-set data structure for which MakeSet and Find require $O(1)$ time, and Union runs in $O(\log V)$ *amortized* time. Using this implementation, the total time spent maintaining the set partition is $O(E + V \log V)$.[6]

But recall that we already need $O(E \log E) = O(E \log V)$ time just to sort the edges. Because this is larger than the time spent maintaining the Union-Find data structure, the overall running time of Kruskal's algorithm is $\boldsymbol{O(E \log V)}$,

[6] A different disjoint-set data structure, which uses a strategy called *union-by-rank with path compression*, performs each Union or Find in $O(\alpha(V))$ amortized time, where α is the almost-but-not-quite-constant *inverse Ackerman function*. If you don't feel like consulting Wikipedia, just think of $\alpha(V)$ as 4. Using this implementation, the total time spent maintaining the set partition is $O(E\alpha(V))$, which is slightly faster when V is large and E is very close to V.

exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.
   (a) Prove that for any cycle in G, the minimum spanning tree of G *excludes* the maximum-weight edge in that cycle.
   (b) Prove or disprove: The minimum spanning tree of G includes the minimum-weight edge in *every* cycle in G.

2. 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.

3. 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 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 Joseph 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.]*

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 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. 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 elementary data structures—in particular, *without* using Fibonacci heaps. This variant of Jarník's algorithm was first described by Edsger Dijkstra in 1958.

8. 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.

   The three classical minimum-spanning tree algorithms described in this chapter can all 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.

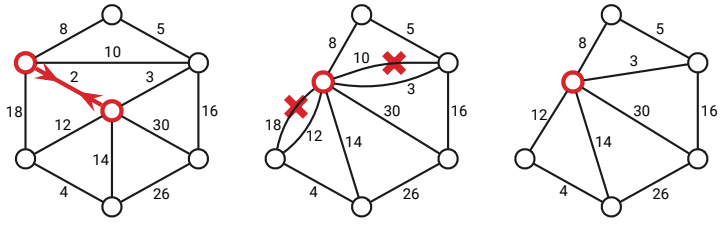


Figure 7.7. Contracting an edge and removing redundant parallel edges.

- 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:

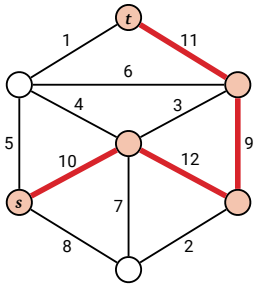
- Contracting an edge of a nice graph yields another nice graph.
- Every nice graph with V vertices has only $O(V)$ edges.

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

Prove that Borůvka's contraction algorithm computes the minimum spanning tree of any nice graph in $O(V)$ time.

9. 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 ∞.)

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



The bottleneck distance between s and t is 9.

(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.]*

10. Borůvka's algorithm can be reformulated to use a standard disjoint-set data structure to identify safe edges, just like Kruskal's algorithm, instead of explicitly counting and labeling components of the evolving spanning forest F in each iteration.

In this variant, each component of F is represented by an *up-tree*; each vertex v stores a pointer $parent(v)$ to its parent, or to v itself if v is the root of its up-tree. The subroutine FIND(v) returns the root of v's up-tree, but also applies *path compression*, reassigning all parent pointers from v to the root to point directly to the root, to speed up future FIND operations.[7] The subroutine UNION combines two up-trees into one by making one of the two root nodes the parent of the other.[8]

[7]Path compression is a form of memoization!

[8]Normally, UNION is implemented more carefully to ensure that the root of the larger or older up-tree does not change; however, those details don't matter here.

```
FIND(v):
  if parent(v) = v
      return v
  else
      v̄ ← FIND(parent(v))
      parent(v) ← v̄
      return v̄
```

```
UNION(u, v):
  ū ← FIND(u)
  v̄ ← FIND(v)
  either
      parent(ū) ← v̄
  or
      parent(v̄) ← ū
```

In the modified version of Borůvka's algorithm, in addition to the parent pointers, the root vertex $\bar{v}$ of each component of F maintains an edge $safe(\bar{v})$, which (at the end of FINDSAFEEDGES) is the lightest edge with one endpoint in that component.

```
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 safe(ū) = NULL or w(uv) < w(safe(ū))
              safe(ū) ← uv
          if safe(v̄) = NULL or w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← TRUE
  return found
```

```
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
```

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

Prove that each call to FINDSAFEEDGES and ADDSAFEEDGES requires only $O(E)$ time. *[Hint: What is the depth of the up-trees when FINDSAFEEDGES ends?]* It follows that this variant of BORŮVKA also runs in $O(E \log V)$ time.