Chapter 15

Minimum Spanning Trees

This chapter applies the greedy algorithm design paradigm to a famous graph problem, the *minimum spanning tree (MST)* problem. The MST problem is a uniquely great playground for the study of greedy algorithms, in which almost any greedy algorithm that you can think of turns out to be correct. After reviewing graphs and defining the problem formally (Section 15.1), we'll discuss the two best-known MST algorithms—Prim's algorithm (Section 15.2) and Kruskal's algorithm (Section 15.5). Both algorithms admit blazingly fast implementations, using the heap and union-find data structures, respectively. Section 15.8 outlines an application of Kruskal's algorithm in machine learning, to single-link clustering.

15.1 Problem Definition

The minimum spanning tree problem is about connecting a bunch of objects as cheaply as possible. The objects and connections could represent something physical, like computer servers and communication links between them. Or maybe each object is a representation of a document (say, as a vector of word frequencies), with connections corresponding to pairs of "similar" documents. The problem arises naturally in several application domains, including computer networking (try a Web search for "spanning tree protocol") and machine learning (see Section 15.8).

15.1.1 Graphs

Objects and connections between them are most naturally modeled with graphs. A *graph* $G = (V, E)$ has two ingredients: a set V of *vertices* and a set E of *edges* (Figure 15.1). This chapter considers only *undirected* graphs, in which each edge e is an unordered pair $\{v, w\}$

of vertices (written as $e = (v, w)$ or $e = (w, v)$), which are called the *endpoints* of the edge.[1] The numbers $|V|$ and $|E|$ of vertices and edges are usually denoted by n and m, respectively.

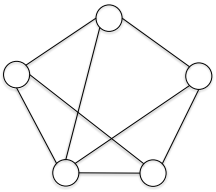


Figure 15.1: An undirected graph with five vertices and eight edges.

Graphs can be encoded in different ways for use in an algorithm. This chapter assumes that the input graph is represented using adjacency lists, with an array of vertices, an array of edges, pointers from each edge to its two endpoints, and pointers from each vertex to its incident edges.[2]

15.1.2 Spanning Trees

The input in the minimum spanning tree problem is an undirected graph $G = (V, E)$ in which each edge e has a real-valued cost c_e. (For example, c_e could indicate the cost of connecting two computer servers.) The goal is to compute a spanning tree of the graph with the minimum-possible sum of edge costs. By a *spanning tree* of G, we mean a subset $T \subseteq E$ of edges that satisfies two properties. First, T should not contain a cycle (this is the "tree" part).[3] Second, for every pair $v, w \in V$ of vertices, T should include a path between v and w (this is the "spanning" part).[4]

[1] There is an analog of the MST problem for directed graphs, which is known as both the *minimum-cost arborescence problem* and the *optimum branching problem*. There are also fast algorithms for this problem, but they lie a bit beyond the scope of this book series.

[2] For more details on graphs and their representations, see Chapter 7 of *Part 2*.

[3] A *cycle* in a graph $G = (V, E)$ is a path that loops back to where it began—an edge sequence $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_k = (v_{k-1}, v_k)$ with $v_k = v_0$.

[4] For convenience, we typically allow a path $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ in a graph to include repeated vertices or, equivalently, to contain one or more cycles. Don't let this bother you: You can always convert such a path into a cycle-free path with the same endpoints v_0 and v_k by repeatedly splicing out subpaths between different visits to the same vertex (see Figure 15.2 below).

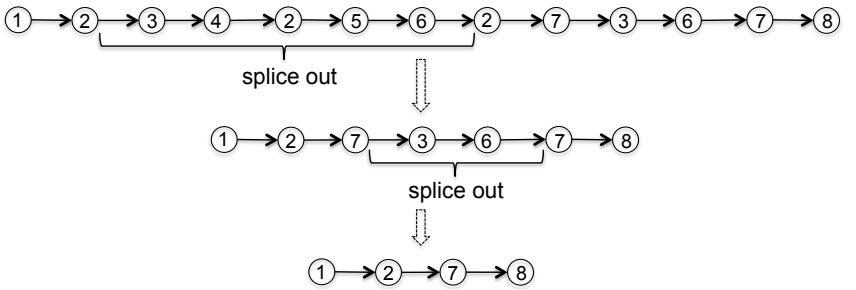
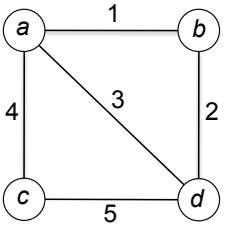


Figure 15.2: A path with repeated vertices can be converted into a path with no repeated vertices and the same endpoints.

Quiz 15.1

What is the minimum sum of edge costs of a spanning tree of the following graph? (Each edge is labeled with its cost.)



a) 6

b) 7

c) 8

d) 9

(See Section 15.1.3 for the solution and discussion.)

It makes sense only to talk about spanning trees of *connected* graphs $G = (V, E)$, in which it's possible to travel from any vertex $v \in V$ to any other vertex $w \in V$ using a path of edges in E.[5] (If there

[5]For example, the graph in Figure 15.1 is connected, while the graph in Figure 15.3 is not.

is no path in E between the vertices v and w, there certainly isn't one in any subset $T \subseteq E$ of edges, either.) For this reason, throughout this chapter we assume that the input graph is a connected graph.

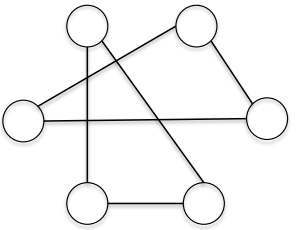


Figure 15.3: A graph that is not connected.

MST Assumption

The input graph $G = (V, E)$ is connected, with at least one path between each pair of vertices.

It's easy enough to compute the minimum spanning tree of a four-vertex graph like the one in Quiz 15.1; what about in general?

Problem: Minimum Spanning Tree (MST)

Input: A connected undirected graph $G = (V, E)$ and a real-valued cost c_e for each edge $e \in E$.

Output: A spanning tree $T \subseteq E$ of G with the minimum-possible sum $\sum_{e \in T} c_e$ of edge costs.[6]

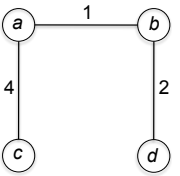
We can assume that the input graph has at most one edge between each pair of vertices; all but the cheapest of a set of parallel edges can be thrown out without changing the problem.

[6] For graphs that are not connected, we could instead consider the minimum spanning *forest* problem, in which the goal is to find a maximal acyclic subgraph with the minimum-possible sum of edge costs. This problem can be solved by first computing the connected components of the input graph in linear time using breadth- or depth-first search (see Chapter 8 of *Part 2*), and then applying an algorithm for the MST problem to each component separately.

Like minimizing the sum of weighted completion times (Chapter 13) or the optimal prefix-free code problem (Chapter 14), the number of possible solutions can be exponential in the size of the problem.[7] Could there be an algorithm that magically homes in on the minimum-cost needle in the haystack of spanning trees?

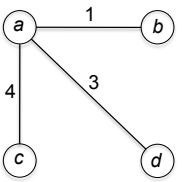
15.1.3 Solution to Quiz 15.1

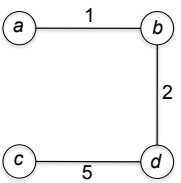
Correct answer: (b). The minimum spanning tree comprises the edges (a, b), (b, d), and (a, c):



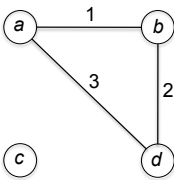
The sum of the edges' costs is 7. The edges do not include a cycle, and they can be used to travel from any vertex to any other vertex.

Here are two spanning trees with an inferior total cost of 8:





The three edges (a, b), (b, d), and (a, d) have a smaller total cost of 6:



but these edges do not form a spanning tree. In fact, they fail on both counts: They form a cycle and there is no way to use them to travel from c to any other vertex.

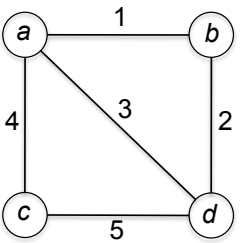
[7] For example, *Cayley's formula* is a famous result from combinatorics stating that the n-vertex complete graph (in which all the $\binom{n}{2}$ possible edges are present) has exactly n^{n-2} different spanning trees. This is bigger than the estimated number of atoms in the known universe when $n \geq 50$.

15.2 Prim's Algorithm

Our first algorithm for the minimum spanning tree problem is *Prim's algorithm*, which is named after Robert C. Prim, who discovered the algorithm in 1957. The algorithm closely resembles Dijkstra's shortest-path algorithm (covered in Chapter 9 of *Part 2*), so it shouldn't surprise you that Edsger W. Dijkstra independently arrived at the same algorithm shortly thereafter, in 1959. Only later was it realized that the algorithm had been discovered over 25 years earlier, by Vojtěch Jarník in 1930. For this reason, the algorithm is also called *Jarník's algorithm* and the *Prim-Jarník algorithm*.[8]

15.2.1 Example

Next we'll step through Prim's algorithm on a concrete example, the same one from Quiz 15.1:



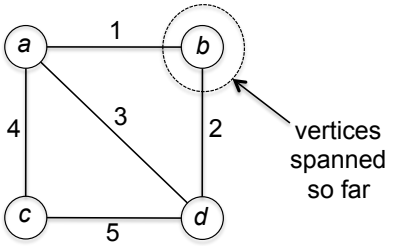
It might seem weird to go through an example of an algorithm before you've seen its code, but trust me: After you understand the example, the pseudocode will practically write itself.[9]

Prim's algorithm begins by choosing an arbitrary vertex—let's say vertex b in our example. (In the end, it won't matter which one we pick.) The plan is to construct a tree one edge at a time, starting from b and growing like a mold until the tree spans the entire vertex set. In each iteration, we'll greedily add the cheapest edge that extends the reach of the tree-so-far.

[8] History buffs should check out the paper "On the History of the Minimum Spanning Tree Problem," by Ronald L. Graham and Pavol Hell (*Annals of the History of Computing*, 1985).

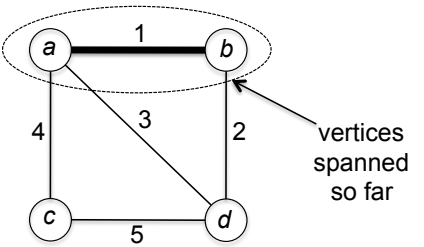
[9] Readers of *Part 2* should recognize strong similarities to Dijkstra's shortest-path algorithm.

The algorithm's initial (empty) tree spans only the starting vertex b. There are two options for expanding its reach: the edge (a, b) and the edge (b, d).



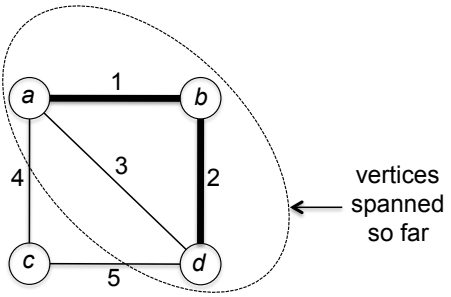
The former is cheaper, so the algorithm chooses it. The tree-so-far spans the vertices a and b.

In the second iteration, three edges would expand the tree's reach: (a, c), (a, d), and (b, d).

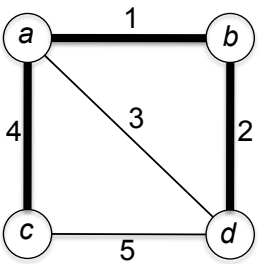


The cheapest of these is (b, d). After its addition, the tree-so-far spans a, b, and d. Both endpoints of the edge (a, d) have been sucked into the set of vertices spanned so far; adding this edge in the future would create a cycle, so the algorithm does not consider it further.

In the final iteration, there are two options for expanding the tree's reach to c, the edges (a, c) and (c, d):



Prim's algorithm chooses the cheaper edge (a, c), resulting in the same minimum spanning tree identified in Quiz 15.1:



15.2.2 Pseudocode

In general, Prim's algorithm grows a spanning tree from a starting vertex one edge at a time, with each iteration extending the reach of the tree-so-far by one additional vertex. As a greedy algorithm, the algorithm always chooses the cheapest edge that does the job.

Prim

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

```
// Initialization
X := {s}    // s is an arbitrarily chosen vertex
T := ∅      // invariant:  the edges in T span X
// Main loop
while there is an edge (v, w) with v ∈ X, w ∉ X do
    (v*, w*) := a minimum-cost such edge
    add vertex w* to X
    add edge (v*, w*) to T
return T
```

The sets T and X keep track of the edges chosen and the vertices spanned so far. The algorithm seeds X with an arbitrarily chosen starting vertex s; as we'll see, the algorithm is correct no matter

which vertex it chooses.[10] Each iteration is responsible for adding one new edge to T. To avoid redundant edges and ensure that the edge addition extends the reach of T, the algorithm considers only the edges that "cross the frontier," with one endpoint in each of X and $V - X$ (Figure 15.4). If there are many such edges, the algorithm greedily chooses the cheapest one. After $n - 1$ iterations (where n is the number of vertices), X contains all the vertices and the algorithm halts. Under our assumption that the input graph G is connected, there's no way for the algorithm to get stuck; if there were ever an iteration with no edges of G crossing between X and $V - X$, we could conclude that G is not connected (because it contains no path from any vertex in X to any vertex in $V - X$).

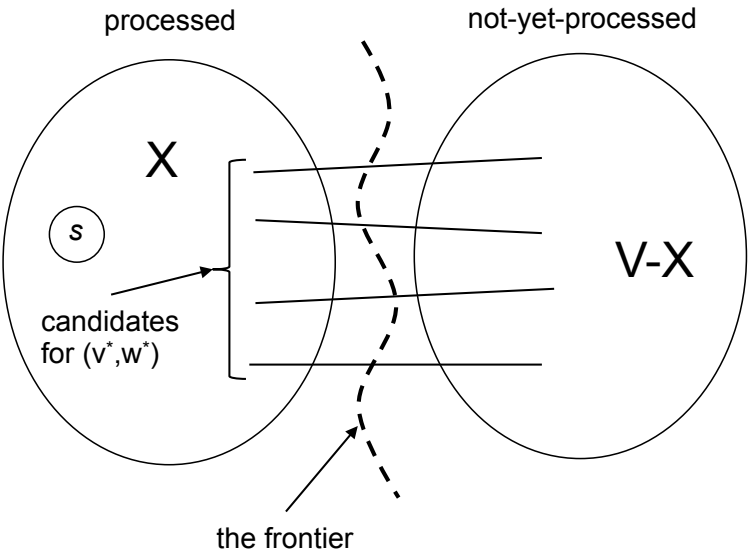


Figure 15.4: Every iteration of Prim's algorithm chooses one new edge that crosses from X to $V - X$.

[10]The MST problem definition makes no reference to a starting vertex, so it might seem weird to artificially introduce one here. One big benefit is that a starting vertex allows us to closely mimic Dijkstra's shortest-path algorithm (which is saddled with a starting vertex by the problem it solves, the single-source shortest path problem). And it doesn't really change the problem: Connecting every pair of vertices is the same thing as connecting some vertex s to every other vertex. (To get a v-w path, paste together paths from v to s and from s to w.)

The algorithm `Prim` computes the minimum spanning tree in the four-vertex five-edge graph of Quiz 15.1, which means approximately... nothing. The fact that an algorithm works correctly on a specific example does *not* imply that it is correct in general![11] You should be initially skeptical of the `Prim` algorithm and demand a proof of correctness.

Theorem 15.1 (Correctness of `Prim`) *For every connected graph $G = (V, E)$ and real-valued edge costs, the `Prim` algorithm returns a minimum spanning tree of G.*

See Section 15.4 for a proof of Theorem 15.1.

15.2.3 Straightforward Implementation

As is typical of greedy algorithms, the running time analysis of Prim's algorithm (assuming a straightforward implementation) is far easier than its correctness proof.

Quiz 15.2

Which of the following running times best describes a straightforward implementation of Prim's minimum spanning tree algorithm for graphs in adjacency-list representation? As usual, n and m denote the number of vertices and edges, respectively, of the input graph.

a) $O(m + n)$

b) $O(m \log n)$

c) $O(n^2)$

d) $O(mn)$

(See below for the solution and discussion.)

Correct answer: (d). A straightforward implementation keeps track of which vertices are in X by associating a Boolean variable with each vertex. In each iteration, it performs an exhaustive search through

[11] Even a broken analog clock is correct two times a day...

all the edges to identify the cheapest one with one endpoint in each of X and $V - X$. After $n - 1$ iterations, the algorithm runs out of new vertices to add to its set X and halts. Because the number of iterations is $O(n)$ and each takes $O(m)$ time, the overall running time is $O(mn)$.

Proposition 15.2 (`Prim` Running Time (Straightforward)) *For every graph $G = (V, E)$ and real-valued edge costs, the straightforward implementation of `Prim` runs in $O(mn)$ time, where $m = |E|$ and $n = |V|$.*

*15.3 Speeding Up Prim's Algorithm via Heaps

15.3.1 The Quest for Near-Linear Running Time

The running time of the straightforward implementation of Prim's algorithm (Proposition 15.2) is nothing to sneeze at—it's a polynomial function of the problem size, while exhaustive search through all of a graph's spanning trees can take an exponential amount of time (see footnote 7). This implementation is fast enough to process medium-size graphs (with thousands of vertices and edges) in a reasonable amount of time, but not big graphs (with millions of vertices and edges). Remember the mantra of any algorithm designer worth their salt: Can we do better? The holy grail in algorithm design is a linear-time algorithm (or close to it), and this is what we want for the MST problem.

We don't need a better *algorithm* to achieve a near-linear-time solution to the problem, just a better *implementation* of Prim's algorithm. The key observation is that the straightforward implementation performs minimum computations, over and over, using exhaustive search. Any method for computing repeated minimum computations faster than exhaustive search would translate to a faster implementation of Prim's algorithm.

We mentioned briefly in Section 14.3.6 that there is, in fact, a data structure whose raison d'être is fast minimum computations: the *heap* data structure. Thus, a light bulb should go off in your head: Prim's algorithm calls out for a heap!

15.3.2 The Heap Data Structure

A heap maintains an evolving set of objects with keys and supports several fast operations, of which we'll need three.

Heaps: Three Supported Operations

INSERT: given a heap H and a new object x, add x to H.

EXTRACTMIN: given a heap H, remove and return from H an object with the smallest key (or a pointer to it).

DELETE: given a heap H and a pointer to an object x in H, delete x from H.

For example, if you invoke INSERT four times to add objects with keys 12, 7, 29, and 15 to an empty heap, the EXTRACTMIN operation will return the object with key 7.

Standard implementations of heaps provide the following guarantee.

Theorem 15.3 (Running Time of Three Heap Operations) *In a heap with n objects, the* INSERT*,* EXTRACTMIN*, and* DELETE *operations run in $O(\log n)$ time.*

As a bonus, in typical implementations, the constant hidden by the big-O notation and the amount of space overhead are relatively small.[12]

15.3.3 How to Use Heaps in Prim's Algorithm

Heaps enable a blazingly fast, near-linear-time implementation of Prim's algorithm.[13]

[12] For the goals of this section, it's not important to know how heaps are implemented and what they look like under the hood. We'll simply be educated clients of them, taking advantage of their logarithmic-time operations. For additional operations and implementation details, see Chapter 10 of *Part 2*.

[13] For readers of *Part 2*, all the ideas in this section will be familiar from the corresponding heap-based implementation of Dijkstra's shortest-path algorithm (Section 10.4).

Theorem 15.4 (`Prim` Running Time (Heap-Based)) *For every graph $G = (V, E)$ and real-valued edge costs, the heap-based implementation of `Prim` runs in $O((m + n) \log n)$ time, where $m = |E|$ and $n = |V|$.*[14]

The running time bound in Theorem 15.4 is only a logarithmic factor more than the time required to read the input. The minimum spanning tree problem thus qualifies as a "for-free primitive," joining the likes of sorting, computing the connected components of a graph, and the single-source shortest path problem.

For-Free Primitives

We can think of an algorithm with linear or near-linear running time as a primitive that we can use essentially "for free" because the amount of computation used is barely more than the amount required simply to read the input. When you have a primitive relevant to your problem that is so blazingly fast, why not use it? For example, you can always compute a minimum spanning tree of your undirected graph data in a preprocessing step, even if you're not quite sure how it will help later. One of the goals of this book series is to stock your algorithmic toolbox with as many for-free primitives as possible, ready to be applied at will.

In the heap-based implementation of Prim's algorithm, the objects in the heap correspond to the as-yet-unprocessed vertices ($V-X$ in the `Prim` pseudocode).[15,16] The key of a vertex $w \in V-X$ is defined as the minimum cost of an incident crossing edge (Figure 15.5).

[14] Under our standing assumption that the input graph is connected, m is at least $n - 1$ and we can therefore simplify $O((m + n) \log n)$ to $O(m \log n)$ in the running time bound.

[15] We refer to vertices of the input graph and the corresponding objects in the heap interchangeably.

[16] Your first thought might be to store the *edges* of the input graph in a heap, with an eye toward replacing the minimum computations (over edges) in the straightforward implementation with calls to ExtractMin. This idea can be made to work, but the slicker and quicker implementation stores vertices in a heap.

Invariant

The key of a vertex $w \in V - X$ is the minimum cost of an edge (v, w) with $v \in X$, or $+\infty$ if no such edge exists.

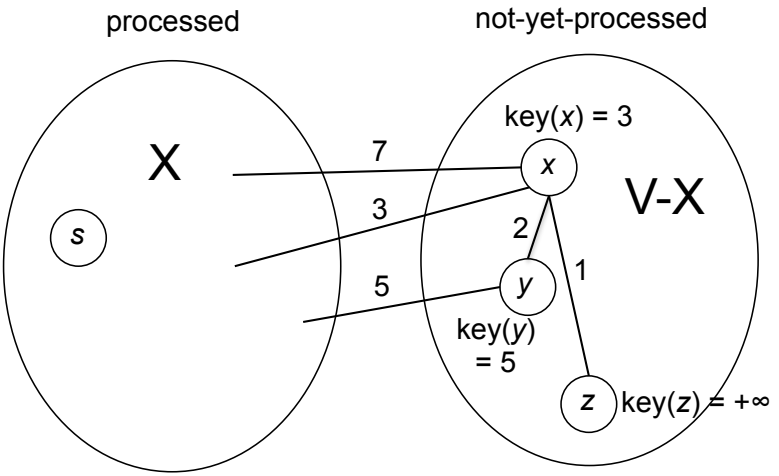


Figure 15.5: The key of a vertex $w \in V - X$ is defined as the minimum cost of an edge (v, w) with $v \in X$ (or $+\infty$, if no such edge exists).

To interpret these keys, imagine using a two-round knockout tournament to identify the minimum-cost edge (v, w) with $v \in X$ and $w \notin X$. The first round comprises a local tournament for each vertex $w \in V - X$, where the participants are the edges (v, w) with $v \in X$ and the first-round winner is the cheapest participant (or $+\infty$, if there are no such edges). The first-round winners (at most one per vertex $w \in V - X$) proceed to the second round, and the final champion is the cheapest first-round winner. Thus, the key of a vertex $w \in V - X$ is exactly the winning edge cost in the local tournament at w. Extracting the vertex with the minimum key then implements the second round of the tournament and returns on a silver platter the next addition to the solution-so-far. As long as we pay the piper and maintain the invariant, keeping objects' keys up to date, we can implement each iteration of Prim's algorithm with a single heap operation.

15.3.4 Pseudocode

The pseudocode then looks like this:

`Prim` (Heap-Based)

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

```
   // Initialization
 1 X := {s}, T = ∅, H := empty heap
 2 for every v ≠ s do
 3     if there is an edge (s, v) ∈ E then
 4         key(v) := c_sv, winner(v) := (s, v)
 5     else      // v has no crossing incident edges
 6         key(v) := +∞, winner(v) := NULL
 7     INSERT v into H
   // Main loop
 8 while H is non-empty do
 9     w* := EXTRACTMIN(H)
10     add w* to X
11     add winner(w*) to T
       // update keys to maintain invariant
12     for every edge (w*, y) with y ∈ V − X do
13         if c_w*y < key(y) then
14             DELETE y from H
15             key(y) := c_w*y, winner(y) := (w*, y)
16             INSERT y into H
17 return T
```

Each not-yet-processed vertex w records in its *winner* and *key* fields the identity and cost of the winner of its local tournament—the cheapest edge incident to w that crosses the frontier (i.e., edges (v, w) with $v \in X$). Lines 2–7 initialize these values for all the vertices other than s so that the invariant is satisfied and insert these vertices into a heap. Lines 9–11 implement one iteration of the main loop of Prim's algorithm. The invariant ensures that the local winner of

the extracted vertex is the cheapest edge crossing the frontier, which is the correct edge to add next to the tree-so-far T. The next quiz illustrates how an extraction can change the frontier, necessitating updates to the keys of vertices still in $V - X$ to maintain the invariant.

Quiz 15.3

In Figure 15.5, suppose the vertex x is extracted and moved to the set X. What should be the new values of y and z's keys, respectively?

a) 1 and 2

b) 2 and 1

c) 5 and $+\infty$

d) $+\infty$ and $+\infty$

(See Section 15.3.6 for the solution and discussion.)

Lines 12–16 of the pseudocode pay the piper and perform the necessary updates to the keys of the vertices remaining in $V - X$. When w^* is moved from $V - X$ to X, edges of the form (w^*, y) with $y \in V - X$ cross the frontier for the first time; these are the new contestants in the local tournaments at the vertices of $V - X$. (We can ignore the fact that edges of the form (u, w^*) with $u \in X$ get sucked into X and no longer cross the frontier, as we're not responsible for maintaining keys for vertices in X.) For a vertex $y \in V - X$, the new winner of its local tournament is either the old winner (stored in $winner(y)$) or the new contestant (w^*, y). Line 12 iterates through the new contestants.[17] Line 13 checks whether an edge (w^*, y) is the new winner in y's local tournament; if it is, lines 14–16 update y's *key* and *winner* fields and the heap H accordingly.[18]

[17] This is the main step in which it's so convenient to have the input graph represented via adjacency lists—the edges of the form (w^*, y) can be accessed directly via w^*'s array of incident edges.

[18] Some heap implementations export a DecreaseKey operation, in which case lines 14–16 can be implemented with one heap operation rather than two.

15.3.5 Running Time Analysis

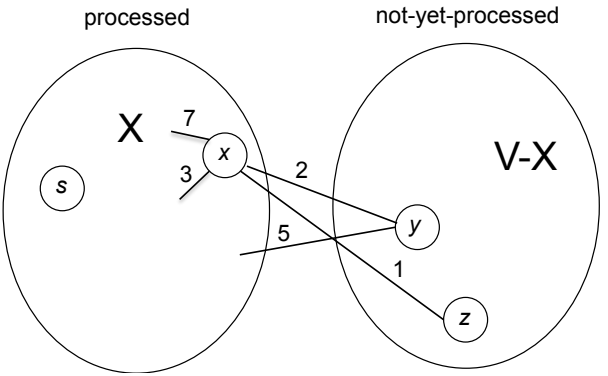
The initialization phase (lines 1–7) performs $n-1$ heap operations (one INSERT per vertex other than s) and $O(m)$ additional work, where n and m denote the number of vertices and edges, respectively. There are $n-1$ iterations of the main while loop (lines 8–16), so lines 9–11 contribute $O(n)$ heap iterations and $O(n)$ additional work to the overall running time. Bounding the total time spent in lines 12–16 is the tricky part; the key observation is that *each edge of G is examined in line 12 exactly once*, in the iteration in which the first of its endpoints gets sucked into X (i.e., plays the role of w^*). When an edge is examined, the algorithm performs two heap operations (in lines 14 and 16) and $O(1)$ additional work, so the total contribution of lines 12–16 to the running time (over all while loop iterations) is $O(m)$ heap operations plus $O(m)$ additional work. Tallying up, the final scorecard reads

$$O(m+n) \text{ heap operations} + O(m+n) \text{ additional work.}$$

The heap never stores more than $n-1$ objects, so each heap operation runs in $O(\log n)$ time (Theorem 15.3). The overall running time is $O((m+n)\log n)$, as promised by Theorem 15.4. $\mathscr{QED}$

15.3.6 Solution to Quiz 15.3

Correct answer: (b). After the vertex x is moved from $V-X$ to X, the new picture is:



Edges of the form (v, x) with $v \in X$ get sucked into X and no longer cross the frontier (as with the edges with costs 3 and 7). The other

edges incident to x, (x, y) and (x, z), get partially yanked out of $V - X$ and now cross the frontier. For both y and z, these new incident crossing edges are cheaper than all their old ones. To maintain the invariant, both of their keys must be updated accordingly: y's key from 5 to 2, and z's key from $+\infty$ to 1.

*15.4 Prim's Algorithm: Proof of Correctness

Proving the correctness of Prim's algorithm (Theorem 15.1) is a bit easier when all the edge costs are distinct. Among friends, let's adopt this assumption for this section. With a little more work, Theorem 15.1 can be proved in its full generality (see Problem 15.5).

The proof breaks down into two steps. The first step identifies a property, called the "minimum bottleneck property," possessed by the output of Prim's algorithm. The second step shows that, in a graph with distinct edge costs, a spanning tree with this property must be a minimum spanning tree.[19]

15.4.1 The Minimum Bottleneck Property

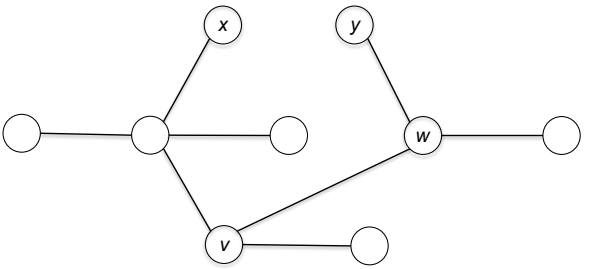
We can motivate the minimum bottleneck property by analogy with Dijkstra's shortest-path algorithm. The only major difference between Prim's and Dijkstra's algorithms is the criterion used to choose a crossing edge in each iteration. Dijkstra's algorithm greedily chooses the eligible edge that minimizes the distance (i.e., the *sum* of edge lengths) from the starting vertex s and, for this reason, computes shortest paths from s to every other vertex (provided edge lengths are nonnegative). Prim's algorithm, by always choosing the eligible edge with minimum individual cost, is effectively striving to minimize the *maximum* edge cost along every path.[20]

[19] A popular if more abstract approach to proving the correctness of Prim's (and Kruskal's) algorithm is to use what's known as the "Cut Property" of MSTs; see Problem 15.7 for details.

[20] This observation is related to a mystery that might be troubling readers of *Part 2*: Why is Dijkstra's algorithm correct only with nonnegative edge lengths, while Prim's algorithm is correct with arbitrary (positive or negative) edge costs? A key ingredient in the correctness proof for Dijkstra's algorithm is "path monotonicity," meaning that tacking on additional edges at the end of a path can only make it worse. Tacking a negative-length edge onto a path would decrease its overall length, so nonnegative edge lengths are necessary for path monotonicity.

our assumption that edges' costs are distinct, the cost of e_2 must be strictly larger: $c_{xy} > c_{vw}$.

Now derive T' from $T^* \cup \{e_1\}$ by removing the edge e_2:



Because T^* has $n-1$ edges, so does T'. Because T^* is connected, so is T'. (Removing an edge from a cycle undoes a type-C edge addition, which by Lemma 15.7(a) has no effect on the number of connected components.) Corollary 15.9 then implies that T' is also acyclic and hence a spanning tree. Because the cost of e_2 is larger than that of e_1, T' has a lower total cost than T^*; this contradicts the supposed optimality of T^* and completes the proof. *QED*

15.4.4 Putting It All Together

We now have the ingredients to immediately deduce the correctness of Prim's algorithm in graphs with distinct edge costs.

Proof of Theorem 15.1: Corollary 15.10 proves that the output of Prim's algorithm is a spanning tree. Lemma 15.5 implies that every edge of this spanning tree satisfies the MBP. Theorem 15.6 guarantees that this spanning tree is a minimum spanning tree. *QED*

15.5 Kruskal's Algorithm

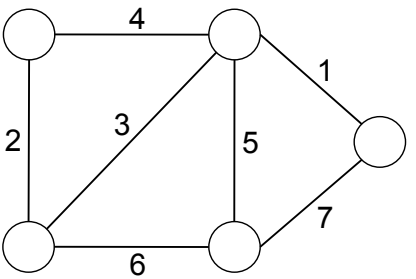
This section describes a second algorithm for the minimum spanning tree problem, *Kruskal's algorithm*.[25] With our blazingly fast heap-based implementation of Prim's algorithm, why should we care about

[25] Discovered by Joseph B. Kruskal in the mid-1950s—roughly the same time that Prim and Dijkstra were rediscovering what is now called Prim's algorithm.

Kruskal's algorithm? Three reasons. One, it's a first-ballot hall-of-fame algorithm, so every seasoned programmer and computer scientist should know about it. Properly implemented, it is competitive with Prim's algorithm in both theory and practice. Two, it provides an opportunity to study a new and useful data structure, the *disjoint-set* or *union-find* data structure. Three, there are some very cool connections between Kruskal's algorithm and widely-used clustering algorithms (see Section 15.8).

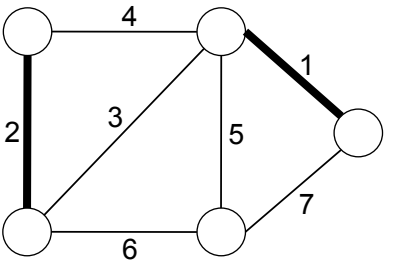
15.5.1 Example

As with Prim's algorithm, it's helpful to see an example of Kruskal's algorithm in action before proceeding to its pseudocode. Here's the input graph:

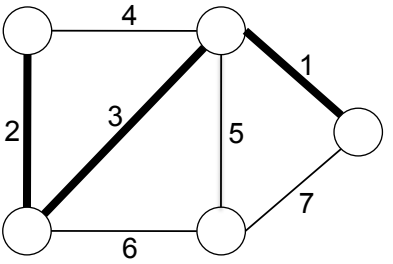


Kruskal's algorithm, like Prim's algorithm, greedily constructs a spanning tree one edge at a time. But rather than growing a single tree from a starting vertex, Kruskal's algorithm can grow multiple trees in parallel, content for them to coalesce into a single tree only at the end of the algorithm. So, while Prim's algorithm was constrained to choose the cheapest edge crossing the current frontier, Kruskal's algorithm is free to choose the cheapest remaining edge in the entire graph. Well, not quite: Cycles are a no-no, so it chooses the cheapest edge that doesn't create a cycle.

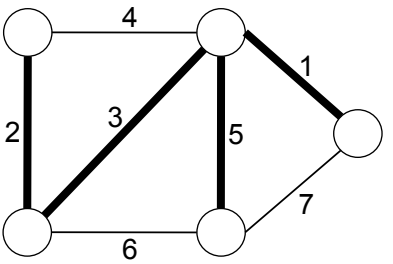
In our example, Kruskal's algorithm starts with an empty edge set T and, in its first iteration, greedily considers the cheapest edge (the edge of cost 1) and adds it to T. The second iteration follows suit with the next-cheapest edge (the edge of cost 2). At this point, the solution-so-far T looks like:



The two edges chosen so far are disjoint, so the algorithm is effectively growing two trees in parallel. The next iteration considers the edge with cost 3. Its inclusion does not create a cycle and also happens to fuse the two trees-so-far into one:



The algorithm next considers the edge of cost 4. Adding this edge to T would create a cycle (with the edges of cost 2 and 3), so the algorithm is forced to skip it. The next-best option is the edge of cost 5; its inclusion does not create a cycle and, in fact, results in a spanning tree:



The algorithm skips the edge of cost 6 (which would create a triangle with the edges of cost 3 and 5) as well as the final edge, of cost 7 (which would create a triangle with the edges of cost 1 and 5). The final output above is the minimum spanning tree of the graph (as you should check).

15.5.2 Pseudocode

With our intuition solidly in place, the following pseudocode won't surprise you.

```
Kruskal

Input: connected undirected graph G = (V, E) in
  adjacency-list representation and a cost c_e for each
  edge e ∈ E.
Output: the edges of a minimum spanning tree of G.

---

// Preprocessing
T := ∅
sort edges of E by cost  // e.g., using MergeSort[26]
// Main loop
for each e ∈ E, in nondecreasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

Kruskal's algorithm considers the edges of the input graph one by one, from cheapest to most expensive, so it makes sense to sort them in nondecreasing order of cost in a preprocessing step (using your favorite sorting algorithm; see footnote 3 in Chapter 13). Ties between edges can be broken arbitrarily. The main loop zips through the edges in this order, adding an edge to the solution-so-far provided it doesn't create a cycle.[27]

It's not obvious that the `Kruskal` algorithm returns a spanning tree, let alone a minimum one. But it does!

Theorem 15.11 (Correctness of `Kruskal`) *For every connected graph $G = (V, E)$ and real-valued edge costs, the `Kruskal` algorithm returns a minimum spanning tree of G.*

[26] The abbreviation "e.g." stands for *exempli gratia* and means "for example."

[27] One easy optimization: You can stop the algorithm early once $|V| - 1$ edges have been added to T, as at this point T is already a spanning tree (by Corollary 15.9).

We've already done most of the heavy lifting in our correctness proof for Prim's algorithm (Theorem 15.1). Section 15.7 supplies the remaining details of the proof of Theorem 15.11.

15.5.3 Straightforward Implementation

How would you actually implement Kruskal's algorithm and, in particular, the cycle-checking required in each iteration?

Quiz 15.4

Which of the following running times best describes a straightforward implementation of Kruskal's MST algorithm for graphs in adjacency-list representation? As usual, n and m denote the number of vertices and edges, respectively, of the input graph.

a) $O(m \log n)$

b) $O(n^2)$

c) $O(mn)$

d) $O(m^2)$

(See below for the solution and discussion.)

Correct answer: (c). In the preprocessing step, the algorithm sorts the edge array of the input graph, which has m entries. With a good sorting algorithm (like MergeSort), this step contributes $O(m \log n)$ work to the overall running time.[28] This work will be dominated by that done by the main loop of the algorithm, which we analyze next.

The main loop has m iterations. Each iteration is responsible for checking whether the edge $e = (v, w)$ under examination can be added to the solution-so-far T without creating a cycle. By Lemma 15.7,

[28] Why $O(m \log n)$ instead of $O(m \log m)$? Because there's no difference between the two expressions. The number of edges of an n-vertex connected graph with no parallel edges is at least $n - 1$ (achieved by a tree) and at most $\binom{n}{2} = \frac{n(n-1)}{2}$ (achieved by a complete graph). Thus $\log m$ lies between $\log(n-1)$ and $2 \log n$ for every connected graph with no parallel edges, which justifies using $\log m$ and $\log n$ interchangeably inside a big-O expression.

adding e to T creates a cycle if and only if T already contains a v-w path. The latter condition can be checked in linear time using any reasonable graph search algorithm, like breadth- or depth-first search starting from v (see Chapter 8 of *Part 2*). And by "linear time," we mean linear in the size of the graph (V, T) which, as an acyclic graph with n vertices, has at most $n - 1$ edges. The per-iteration running time is therefore $O(n)$, for an overall running time of $O(mn)$.

Proposition 15.12 (`Kruskal` Run Time (Straightforward)) *For every graph $G = (V, E)$ and real-valued edge costs, the straightforward implementation of `Kruskal` runs in $O(mn)$ time, where $m = |E|$ and $n = |V|$.*

*15.6 Speeding Up Kruskal's Algorithm via Union-Find

As with Prim's algorithm, we can reduce the running time of Kruskal's algorithm from the reasonable polynomial bound of $O(mn)$ (Proposition 15.12) to the blazingly fast near-linear bound of $O(m \log n)$ through the deft use of a data structure. None of the data structures discussed previously in this book series are right for the job; we'll need a new one, called the *union-find* data structure.[29]

Theorem 15.13 (`Kruskal` Run Time (Union-Find-Based)) *For every graph $G = (V, E)$ and real-valued edge costs, the union-find-based implementation of `Kruskal` runs in $O((m + n) \log n)$ time, where $m = |E|$ and $n = |V|$.*[30]

15.6.1 The Union-Find Data Structure

Whenever a program does a significant computation over and over again, it's a clarion call for a data structure to speed up those computations. Prim's algorithm performs minimum computations in each iteration of its main loop, so the heap data structure is an obvious match. Each iteration of Kruskal's algorithm performs a cycle check or, equivalently, a path check. (Adding an edge (v, w) to the solution-so-far T creates a cycle if and only if T already contains a v-w path.)

[29] Also known as the *disjoint-set* data structure.

[30] Again, under our standing assumption that the input graph is connected, we can simplify the $O((m + n) \log n)$ bound to $O(m \log n)$.