(4.15) *Using a priority queue, Dijkstra's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m)$ time, plus the time for n* `ExtractMin` *and m* `ChangeKey` *operations.*

Using the heap-based priority queue implementation discussed in Chapter 2, each priority queue operation can be made to run in $O(\log n)$ time. Thus the overall time for the implementation is $O(m \log n)$.

4.5 The Minimum Spanning Tree Problem

We now apply an exchange argument in the context of a second fundamental problem on graphs: the Minimum Spanning Tree Problem.

The Problem

Suppose we have a set of locations $V = \{v_1, v_2, \ldots, v_n\}$, and we want to build a communication network on top of them. The network should be connected—there should be a path between every pair of nodes—but subject to this requirement, we wish to build it as cheaply as possible.

For certain pairs (v_i, v_j), we may build a direct link between v_i and v_j for a certain cost $c(v_i, v_j) > 0$. Thus we can represent the set of possible links that may be built using a graph $G = (V, E)$, with a positive *cost* c_e associated with each edge $e = (v_i, v_j)$. The problem is to find a subset of the edges $T \subseteq E$ so that the graph (V, T) is connected, and the total cost $\sum_{e \in T} c_e$ is as small as possible. (We will assume that the full graph G is connected; otherwise, no solution is possible.)

Here is a basic observation.

(4.16) *Let T be a minimum-cost solution to the network design problem defined above. Then (V, T) is a tree.*

Proof. By definition, (V, T) must be connected; we show that it also wil contain no cycles. Indeed, suppose it contained a cycle C, and let e be an edge on C. We claim that $(V, T - \{e\})$ is still connected, since any path tha previously used the edge e can now go "the long way" around the remainde of the cycle C instead. It follows that $(V, T - \{e\})$ is also a valid solution to th problem, and it is cheaper—a contradiction. ■

If we allow some edges to have 0 cost (that is, we assume only that th costs c_e are nonnegative), then a minimum-cost solution to the network desig problem may have extra edges—edges that have 0 cost and could optional be deleted. But even in this case, there is always a minimum-cost solution th is a tree. Starting from any optimal solution, we could keep deleting edges

cycles until we had a tree; with nonnegative edges, the cost would not increase during this process.

We will call a subset $T \subseteq E$ a *spanning tree* of G if (V, T) is a tree. Statement (4.16) says that the goal of our network design problem can be rephrased as that of finding the cheapest spanning tree of the graph; for this reason, it is generally called the *Minimum Spanning Tree Problem*. Unless G is a very simple graph, it will have exponentially many different spanning trees, whose structures may look very different from one another. So it is not at all clear how to efficiently find the cheapest tree from among all these options.

Designing Algorithms

As with the previous problems we've seen, it is easy to come up with a number of natural greedy algorithms for the problem. But curiously, and fortunately, this is a case where *many* of the first greedy algorithms one tries turn out to be correct: they each solve the problem optimally. We will review a few of these algorithms now and then discover, via a nice pair of exchange arguments, some of the underlying reasons for this plethora of simple, optimal algorithms.

Here are three greedy algorithms, each of which correctly finds a minimum spanning tree.

- One simple algorithm starts without any edges at all and builds a spanning tree by successively inserting edges from E in order of increasing cost. As we move through the edges in this order, we insert each edge e as long as it does not create a cycle when added to the edges we've already inserted. If, on the other hand, inserting e would result in a cycle, then we simply discard e and continue. This approach is called *Kruskal's Algorithm*.
- Another simple greedy algorithm can be designed by analogy with Dijkstra's Algorithm for paths, although, in fact, it is even simpler to specify than Dijkstra's Algorithm. We start with a root node s and try to greedily grow a tree from s outward. At each step, we simply add the node that can be attached as cheaply as possibly to the partial tree we already have.

  More concretely, we maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far. Initially, $S = \{s\}$. In each iteration, we grow S by one node, adding the node v that minimizes the "attachment cost" $\min_{e=(u,v):u \in S} c_e$, and including the edge $e = (u, v)$ that achieves this minimum in the spanning tree. This approach is called *Prim's Algorithm*.
- Finally, we can design a greedy algorithm by running sort of a "backward" version of Kruskal's Algorithm. Specifically, we start with the full graph (V, E) and begin deleting edges in order of decreasing cost. As we get to each edge e (starting from the most expensive), we delete it as

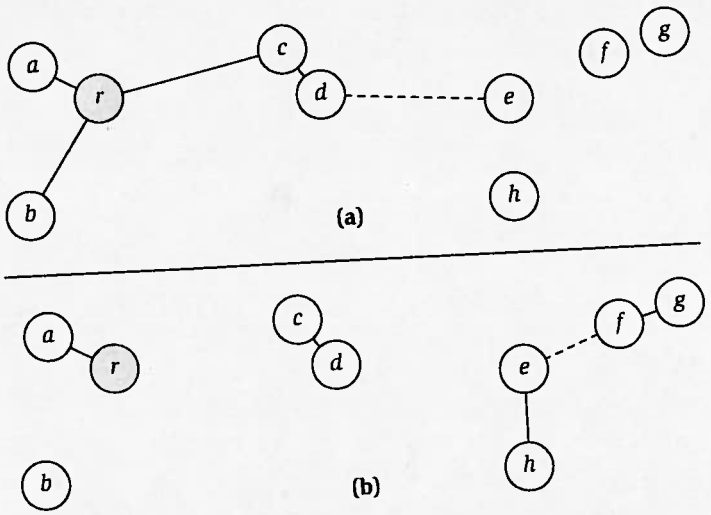

Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

long as doing so would not actually disconnect the graph we currently have. For want of a better name, this approach is generally called the *Reverse-Delete Algorithm* (as far as we can tell, it's never been named after a specific person).

For example, Figure 4.9 shows the first four edges added by Prim's and Kruskal's Algorithms respectively, on a geometric instance of the Minimum Spanning Tree Problem in which the cost of each edge is proportional to the geometric distance in the plane.

The fact that each of these algorithms is guaranteed to produce an optimal solution suggests a certain "robustness" to the Minimum Spanning Tree Problem—there are many ways to get to the answer. Next we explore some of the underlying reasons why so many different algorithms produce minimum-cost spanning trees.

Analyzing the Algorithms

All these algorithms work by repeatedly inserting or deleting edges from a partial solution. So, to analyze them, it would be useful to have in hand some basic facts saying when it is "safe" to include an edge in the minimum spanning tree, and, correspondingly, when it is safe to eliminate an edge on the grounds that it couldn't possibly be in the minimum spanning tree. For purposes of the analysis, we will make the simplifying assumption that all edge costs are distinct from one another (i.e., no two are equal). This assumption makes it

easier to express the arguments that follow, and we will show later in this section how this assumption can be easily eliminated.

When Is It Safe to Include an Edge in the Minimum Spanning Tree? The crucial fact about edge insertion is the following statement, which we will refer to as the *Cut Property*.

(4.17) *Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the minimum-cost edge with one end in S and the other in $V - S$. Then every minimum spanning tree contains the edge e.*

Proof. Let T be a spanning tree that does not contain e; we need to show that T does not have the minimum possible cost. We'll do this using an exchange argument: we'll identify an edge e' in T that is more expensive than e, and with the property exchanging e for e' results in another spanning tree. This resulting spanning tree will then be cheaper than T, as desired.

The crux is therefore to find an edge that can be successfully exchanged with e. Recall that the ends of e are v and w. T is a spanning tree, so there must be a path P in T from v to w. Starting at v, suppose we follow the nodes of P in sequence; there is a first node w' on P that is in $V - S$. Let $v' \in S$ be the node just before w' on P, and let $e' = (v', w')$ be the edge joining them. Thus, e' is an edge of T with one end in S and the other in $V - S$. See Figure 4.10 for the situation at this stage in the proof.

If we exchange e for e', we get a set of edges $T' = T - \{e'\} \cup \{e\}$. We claim that T' is a spanning tree. Clearly (V, T') is connected, since (V, T) is connected, and any path in (V, T) that used the edge $e' = (v', w')$ can now be "rerouted" in (V, T') to follow the portion of P from v' to v, then the edge e, and then the portion of P from w to w'. To see that (V, T') is also acyclic, note that the only cycle in $(V, T' \cup \{e'\})$ is the one composed of e and the path P, and this cycle is not present in (V, T') due to the deletion of e'.

We noted above that the edge e' has one end in S and the other in $V - S$. But e is the cheapest edge with this property, and so $c_e < c_{e'}$. (The inequality is strict since no two edges have the same cost.) Thus the total cost of T' is less than that of T, as desired. ■

The proof of (4.17) is a bit more subtle than it may first appear. To appreciate this subtlety, consider the following shorter but incorrect argument for (4.17). Let T be a spanning tree that does not contain e. Since T is a spanning tree, it must contain an edge f with one end in S and the other in $V - S$. Since e is the cheapest edge with this property, we have $c_e < c_f$, and hence $T - \{f\} \cup \{e\}$ is a spanning tree that is cheaper than T.

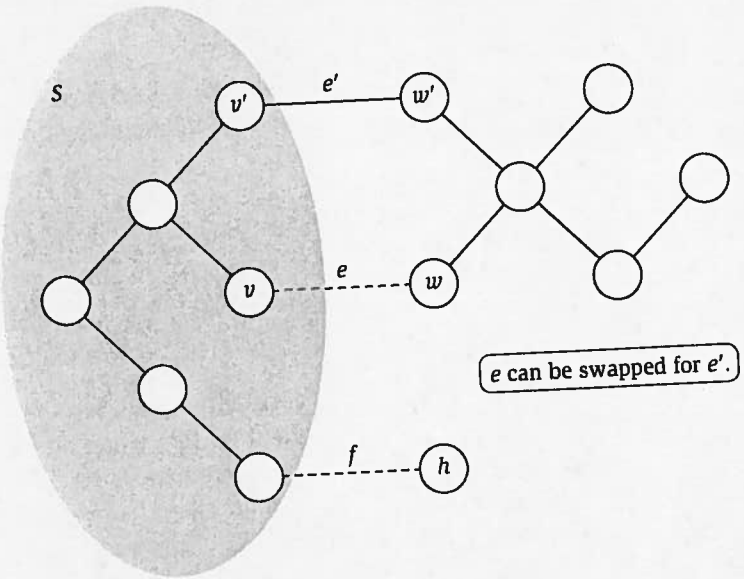

Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T, as described in the proof of (4.17).

The problem with this argument is not in the claim that f exists, or that $T - \{f\} \cup \{e\}$ is cheaper than T. The difficulty is that $T - \{f\} \cup \{e\}$ may not be a spanning tree, as shown by the example of the edge f in Figure 4.10. The point is that we can't prove (4.17) by simply picking *any* edge in T that crosses from S to $V - S$; some care must be taken to find the right one.

The Optimality of Kruskal's and Prim's Algorithms We can now easily prove the optimality of both Kruskal's Algorithm and Prim's Algorithm. The point is that both algorithms only include an edge when it is justified by the Cut Property (4.17).

(4.18) *Kruskal's Algorithm produces a minimum spanning tree of G.*

Proof. Consider any edge $e = (v, w)$ added by Kruskal's Algorithm, and let S be the set of all nodes to which v has a path at the moment just before e is added. Clearly $v \in S$, but $w \notin S$, since adding e does not create a cycle. Moreover, no edge from S to $V - S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have been added by Kruskal's Algorithm. Thus e is the cheapest edge with one end in S and the other in $V - S$, and so by (4.17) it belongs to every minimum spanning tree.

So if we can show that the output (V, T) of Kruskal's Algorithm is in fact a spanning tree of G, then we will be done. Clearly (V, T) contains no cycles, since the algorithm is explicitly designed to avoid creating cycles. Further, if (V, T) were not connected, then there would exist a nonempty subset of nodes S (not equal to all of V) such that there is no edge from S to $V - S$. But this contradicts the behavior of the algorithm: we know that since G is connected, there is at least one edge between S and $V - S$, and the algorithm will add the first of these that it encounters. ■

(4.19) *Prim's Algorithm produces a minimum spanning tree of G.*

Proof. For Prim's Algorithm, it is also very easy to show that it only adds edges belonging to every minimum spanning tree. Indeed, in each iteration of the algorithm, there is a set $S \subseteq V$ on which a partial spanning tree has been constructed, and a node v and edge e are added that minimize the quantity $\min_{e=(u,v):u \in S} c_e$. By definition, e is the cheapest edge with one end in S and the other end in $V - S$, and so by the Cut Property (4.17) it is in every minimum spanning tree.

It is also straightforward to show that Prim's Algorithm produces a spanning tree of G, and hence it produces a minimum spanning tree. ■

When Can We Guarantee an Edge Is Not in the Minimum Spanning Tree? The crucial fact about edge deletion is the following statement, which we will refer to as the *Cycle Property*.

(4.20) *Assume that all edge costs are distinct. Let C be any cycle in G, and let edge $e = (v, w)$ be the most expensive edge belonging to C. Then e does not belong to any minimum spanning tree of G.*

Proof. Let T be a spanning tree that contains e; we need to show that T does not have the minimum possible cost. By analogy with the proof of the Cut Property (4.17), we'll do this with an exchange argument, swapping e for a cheaper edge in such a way that we still have a spanning tree.

So again the question is: How do we find a cheaper edge that can be exchanged in this way with e? Let's begin by deleting e from T; this partitions the nodes into two components: S, containing node v; and $V - S$, containing node w. Now, the edge we use in place of e should have one end in S and the other in $V - S$, so as to stitch the tree back together.

We can find such an edge by following the cycle C. The edges of C other than e form, by definition, a path P with one end at v and the other at w. If we follow P from v to w, we begin in S and end up in $V - S$, so there is some

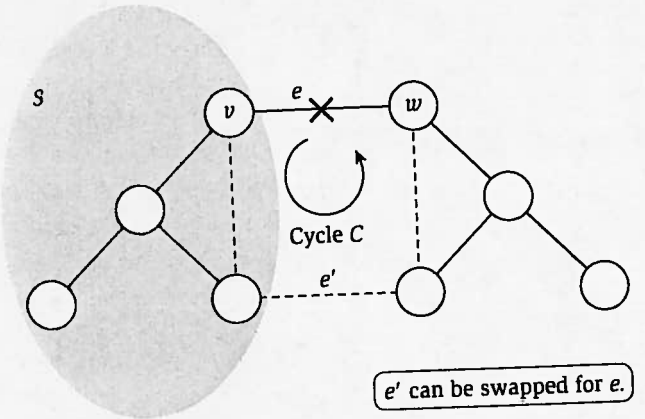

Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T, as described in the proof of (4.20).

edge e' on P that crosses from S to $V - S$. See Figure 4.11 for an illustration of this.

Now consider the set of edges $T' = T - \{e\} \cup \{e'\}$. Arguing just as in the proof of the Cut Property (4.17), the graph (V, T') is connected and has no cycles, so T' is a spanning tree of G. Moreover, since e is the most expensive edge on the cycle C, and e' belongs to C, it must be that e' is cheaper than e, and hence T' is cheaper than T, as desired. ■

The Optimality of the Reverse-Delete Algorithm Now that we have the Cycle Property (4.20), it is easy to prove that the Reverse-Delete Algorithm produces a minimum spanning tree. The basic idea is analogous to the optimality proofs for the previous two algorithms: Reverse-Delete only adds an edge when it is justified by (4.20).

(4.21) *The Reverse-Delete Algorithm produces a minimum spanning tree of G.*

Proof. Consider any edge $e = (v, w)$ removed by Reverse-Delete. At the time that e is removed, it lies on a cycle C; and since it is the first edge encountered by the algorithm in decreasing order of edge costs, it must be the most expensive edge on C. Thus by (4.20), e does not belong to any minimum spanning tree.

So if we show that the output (V, T) of Reverse-Delete is a spanning tree of G, we will be done. Clearly (V, T) is connected, since the algorithm never removes an edge when this will disconnect the graph. Now, suppose by way of

contradiction that (V, T) contains a cycle C. Consider the most expensive edge e on C, which would be the first one encountered by the algorithm. This edge should have been removed, since its removal would not have disconnected the graph, and this contradicts the behavior of Reverse-Delete. ■

While we will not explore this further here, the combination of the Cut Property (4.17) and the Cycle Property (4.20) implies that something even more general is going on. *Any* algorithm that builds a spanning tree by repeatedly including edges when justified by the Cut Property and deleting edges when justified by the Cycle Property—in any order at all—will end up with a minimum spanning tree. This principle allows one to design natural greedy algorithms for this problem beyond the three we have considered here, and it provides an explanation for why so many greedy algorithms produce optimal solutions for this problem.

Eliminating the Assumption that All Edge Costs Are Distinct Thus far, we have assumed that all edge costs are distinct, and this assumption has made the analysis cleaner in a number of places. Now, suppose we are given an instance of the Minimum Spanning Tree Problem in which certain edges have the same cost – how can we conclude that the algorithms we have been discussing still provide optimal solutions?

There turns out to be an easy way to do this: we simply take the instance and perturb all edge costs by different, extremely small numbers, so that they all become distinct. Now, any two costs that differed originally will still have the same relative order, since the perturbations are so small; and since all of our algorithms are based on just comparing edge costs, the perturbations effectively serve simply as "tie-breakers" to resolve comparisons among costs that used to be equal.

Moreover, we claim that any minimum spanning tree T for the new, perturbed instance must have also been a minimum spanning tree for the original instance. To see this, we note that if T cost more than some tree T^* in the original instance, then for small enough perturbations, the change in the cost of T cannot be enough to make it better than T^* under the new costs. Thus, if we run any of our minimum spanning tree algorithms, using the perturbed costs for comparing edges, we will produce a minimum spanning tree T that is also optimal for the original instance.

Implementing Prim's Algorithm

We next discuss how to implement the algorithms we have been considering so as to obtain good running-time bounds. We will see that both Prim's and Kruskal's Algorithms can be implemented, with the right choice of data structures, to run in $O(m \log n)$ time. We will see how to do this for Prim's Algorithm

here, and defer discussing the implementation of Kruskal's Algorithm to the next section. Obtaining a running time close to this for the Reverse-Delete Algorithm is difficult, so we do not focus on Reverse-Delete in this discussion.

For Prim's Algorithm, while the proof of correctness was quite different from the proof for Dijkstra's Algorithm for the Shortest-Path Algorithm, the implementations of Prim and Dijkstra are almost identical. By analogy with Dijkstra's Algorithm, we need to be able to decide which node v to add next to the growing set S, by maintaining the attachment costs $a(v) = \min_{e=(u,v):u \in S} c_e$ for each node $v \in V - S$. As before, we keep the nodes in a priority queue with these attachment costs $a(v)$ as the keys; we select a node with an `ExtractMin` operation, and update the attachment costs using `ChangeKey` operations. There are $n - 1$ iterations in which we perform `ExtractMin`, and we perform `ChangeKey` at most once for each edge. Thus we have

(4.22) *Using a priority queue, Prim's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m)$ time, plus the time for n* `ExtractMin`*, and m* `ChangeKey` *operations.*

As with Dijkstra's Algorithm, if we use a heap-based priority queue we can implement both `ExtractMin` and `ChangeKey` in $O(\log n)$ time, and so get an overall running time of $O(m \log n)$.

Extensions

The minimum spanning tree problem emerged as a particular formulation of a broader *network design* goal—finding a good way to connect a set of sites by installing edges between them. A minimum spanning tree optimizes a particular goal, achieving connectedness with minimum total edge cost. But there are a range of further goals one might consider as well.

We may, for example, be concerned about point-to-point distances in the spanning tree we build, and be willing to reduce these even if we pay more for the set of edges. This raises new issues, since it is not hard to construct examples where the minimum spanning tree does not minimize point-to-point distances, suggesting some tension between these goals.

Alternately, we may care more about the *congestion* on the edges. Given traffic that needs to be routed between pairs of nodes, one could seek a spanning tree in which no single edge carries more than a certain amount of this traffic. Here too, it is easy to find cases in which the minimum spanning tree ends up concentrating a lot of traffic on a single edge.

More generally, it is reasonable to ask whether a spanning tree is even the right kind of solution to our network design problem. A tree has the property that destroying any one edge disconnects it, which means that trees are not at

all robust against failures. One could instead make resilience an explicit goal, for example seeking the cheapest connected network on the set of sites that remains connected after the deletion of any one edge.

All of these extensions lead to problems that are computationally much harder than the basic Minimum Spanning Tree problem, though due to their importance in practice there has been research on good heuristics for them.

4.6 Implementing Kruskal's Algorithm: The Union-Find Data Structure

One of the most basic graph problems is to find the set of connected components. In Chapter 3 we discussed linear-time algorithms using BFS or DFS for finding the connected components of a graph.

In this section, we consider the scenario in which a graph evolves through the addition of edges. That is, the graph has a fixed population of nodes, but it grows over time by having edges appear between certain pairs of nodes. Our goal is to maintain the set of connected components of such a graph throughout this evolution process. When an edge is added to the graph, we don't want to have to recompute the connected components from scratch. Rather, we will develop a data structure that we call the `Union-Find` structure, which will store a representation of the components in a way that supports rapid searching and updating.

This is exactly the data structure needed to implement Kruskal's Algorithm efficiently. As each edge $e = (v, w)$ is considered, we need to efficiently find the identities of the connected components containing v and w. If these components are different, then there is no path from v and w, and hence edge e should be included; but if the components are the same, then there is a path on the edges already included, and so e should be omitted. In the event that e is included, the data structure should also support the efficient merging of the components of v and w into a single new component.

The Problem

The `Union-Find` data structure allows us to maintain disjoint sets (such as the components of a graph) in the following sense. Given a node u, the operation `Find(`u`)` will return the name of the set containing u. This operation can be used to test if two nodes u and v are in the same set, by simply checking if `Find(`u`) = Find(`v`)`. The data structure will also implement an operation `Union(`A, B`)` to take two sets A and B and merge them to a single set.

These operations can be used to maintain connected components of an evolving graph $G = (V, E)$ as edges are added. The sets will be the connected components of the graph. For a node u, the operation `Find(`u`)` will return the