(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 will contain no cycles. Indeed, suppose it contained a cycle C, and let e be any edge on C. We claim that $(V, T - \{e\})$ is still connected, since any path that previously used the edge e can now go "the long way" around the remainder of the cycle C instead. It follows that $(V, T - \{e\})$ is also a valid solution to the problem, and it is cheaper—a contradiction. ∎

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

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

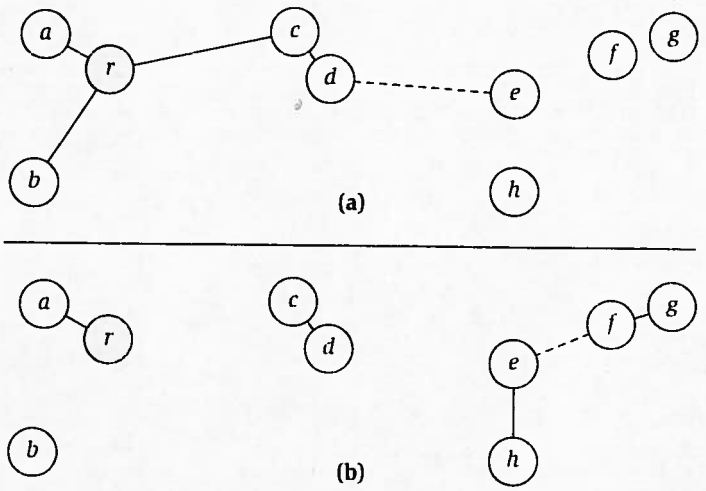

Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

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