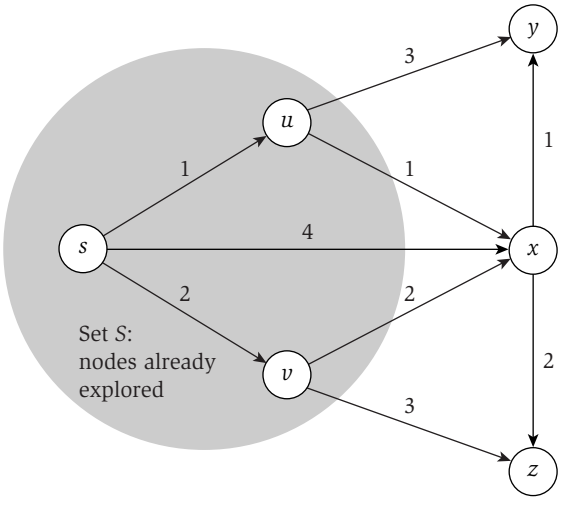

Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

the sense that we always form the shortest new s-v path we can make from a path in S followed by a single edge. We prove its correctness using a variant of our first style of analysis: we show that it "stays ahead" of all other solutions by establishing, inductively, that each time it selects a path to a node v, that path is shorter than every other possible path to v.

(4.14) *Consider the set S at any point in the algorithm's execution. For each $u \in S$, the path P_u is a shortest s-u path.*

Note that this fact immediately establishes the correctness of Dijkstra's Algorithm, since we can apply it when the algorithm terminates, at which point S includes all nodes.

Proof. We prove this by induction on the size of S. The case $|S| = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = 0$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; we now grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge on our s-v path P_v.

By induction hypothesis, P_u is the shortest s-u path for each $u \in S$. Now consider any other s-v path P; we wish to show that it is at least as long as P_v. In order to reach v, this path P must leave the set S *somewhere*; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y.

The situation is now as depicted in Figure 4.8, and the crux of the proof is very simple: P cannot be shorter than P_v because it is already at least as

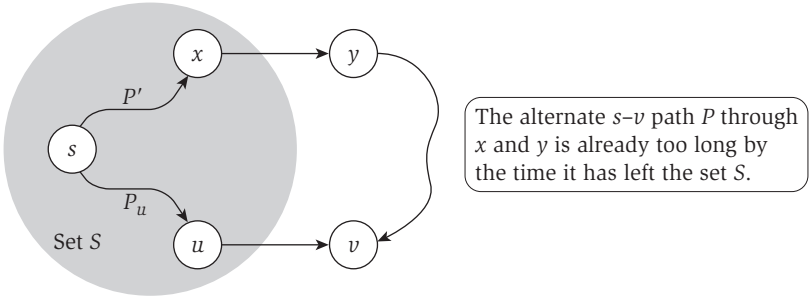

Figure 4.8 The shortest path P_v and an alternate s-v path P through the node y.

long as P_v by the time it has left the set S. Indeed, in iteration $k+1$, Dijkstra's Algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favor of adding v. This means that there is no path from s to y through x that is shorter than P_v. But the subpath of P up to y is such a path, and so this subpath is at least as long as P_v. Since edge lengths are nonnegative, the full path P is at least as long as P_v as well.

This is a complete proof; one can also spell out the argument in the previous paragraph using the following inequalities. Let P' be the subpath of P from s to x. Since $x \in S$, we know by the induction hypothesis that P_x is a shortest s-x path (of length $d(x)$), and so $\ell(P') \geq \ell(P_x) = d(x)$. Thus the subpath of P out to node y has length $\ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq d'(y)$, and the full path P is at least as long as this subpath. Finally, since Dijkstra's Algorithm selected v in this iteration, we know that $d'(y) \geq d'(v) = \ell(P_v)$. Combining these inequalities shows that $\ell(P) \geq \ell(P') + \ell(x, y) \geq \ell(P_v)$. ■

Here are two observations about Dijkstra's Algorithm and its analysis. First, the algorithm does not always find shortest paths if some of the edges can have negative lengths. (Do you see where the proof breaks?) Many shortest-path applications involve negative edge lengths, and a more complex algorithm—due to Bellman and Ford—is required for this case. We will see this algorithm when we consider the topic of dynamic programming.

The second observation is that Dijkstra's Algorithm is, in a sense, even simpler than we've described here. Dijkstra's Algorithm is really a "continuous" version of the standard breadth-first search algorithm for traversing a graph, and it can be motivated by the following physical intuition. Suppose the edges of G formed a system of pipes filled with water, joined together at the nodes; each edge e has length ℓ_e and a fixed cross-sectional area. Now suppose an extra droplet of water falls at node s and starts a wave from s. As the wave expands out of node s at a constant speed, the expanding sphere

of wavefront reaches nodes in increasing order of their distance from s. It is easy to believe (and also true) that the path taken by the wavefront to get to any node v is a shortest path. Indeed, it is easy to see that this is exactly the path to v found by Dijkstra's Algorithm, and that the nodes are discovered by the expanding water in the same order that they are discovered by Dijkstra's Algorithm.

Implementation and Running Time To conclude our discussion of Dijkstra's Algorithm, we consider its running time. There are $n-1$ iterations of the `While` loop for a graph with n nodes, as each iteration adds a new node v to S. Selecting the correct node v efficiently is a more subtle issue. One's first impression is that each iteration would have to consider each node $v \notin S$, and go through all the edges between S and v to determine the minimum $\min_{e=(u,v):u\in S} d(u) + \ell_e$, so that we can select the node v for which this minimum is smallest. For a graph with m edges, computing all these minima can take $O(m)$ time, so this would lead to an implementation that runs in $O(mn)$ time.

We can do considerably better if we use the right data structures. First, we will explicitly maintain the values of the minima $d'(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$ for each node $v \in V - S$, rather than recomputing them in each iteration. We can further improve the efficiency by keeping the nodes $V - S$ in a priority queue with $d'(v)$ as their keys. Priority queues were discussed in Chapter 2; they are data structures designed to maintain a set of n elements, each with a key. A priority queue can efficiently insert elements, delete elements, change an element's key, and extract the element with the minimum key. We will need the third and fourth of the above operations: `ChangeKey` and `ExtractMin`.

How do we implement Dijkstra's Algorithm using a priority queue? We put the nodes V in a priority queue with $d'(v)$ as the key for $v \in V$. To select the node v that should be added to the set S, we need the `ExtractMin` operation. To see how to update the keys, consider an iteration in which node v is added to S, and let $w \notin S$ be a node that remains in the priority queue. What do we have to do to update the value of $d'(w)$? If (v, w) is not an edge, then we don't have to do anything: the set of edges considered in the minimum $\min_{e=(u,w):u\in S} d(u) + \ell_e$ is exactly the same before and after adding v to S. If $e' = (v, w) \in E$, on the other hand, then the new value for the key is $\min(d'(w), d(v) + \ell_{e'})$. If $d'(w) > d(v) + \ell_{e'}$ then we need to use the `ChangeKey` operation to decrease the key of node w appropriately. This `ChangeKey` operation can occur at most once per edge, when the tail of the edge e' is added to S. In summary, we have the following result.

(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

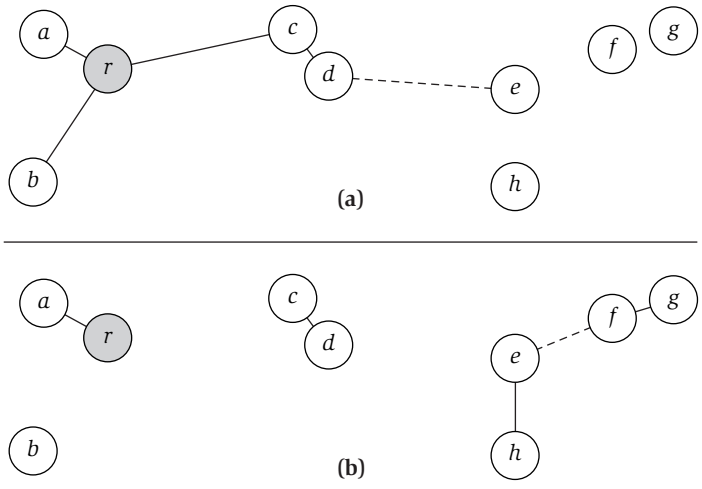

Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

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

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.

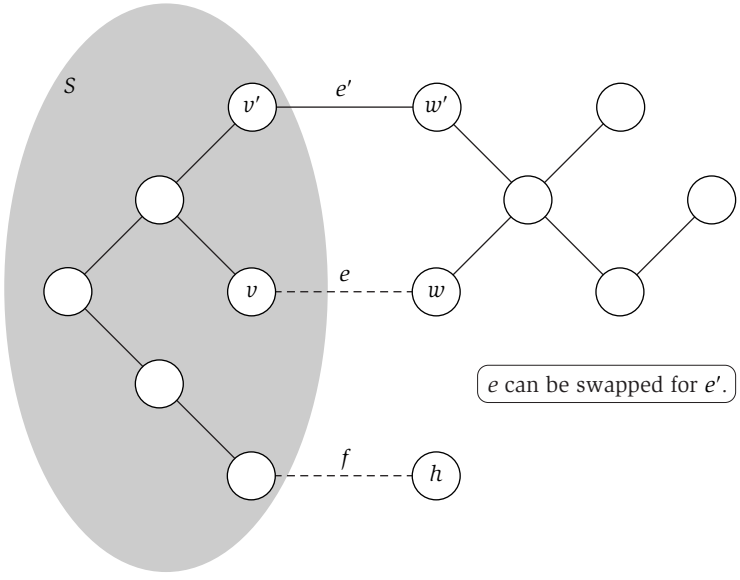

Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T, as described in the proof of (4.17).

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

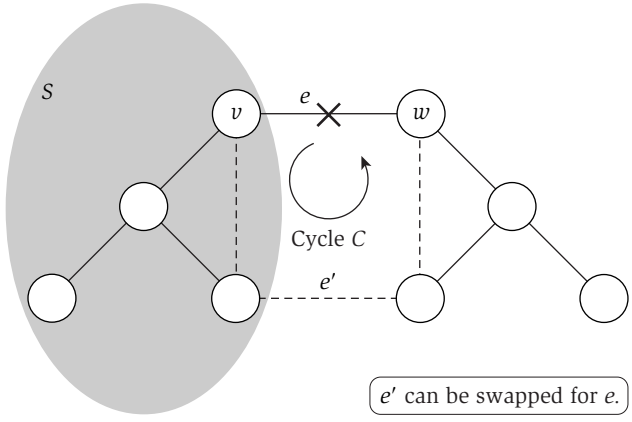

Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T, as described in the proof of (4.20).

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 v-w 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

name of the component containing u. If we add an edge (u, v) to the graph, then we first test if u and v are already in the same connected component (by testing if `Find`(u) = `Find`(v)). If they are not, then `Union`(`Find`(u),`Find`(v)) can be used to merge the two components into one. It is important to note that the `Union-Find` data structure can only be used to maintain components of a graph as we *add* edges; it is not designed to handle the effects of edge deletion, which may result in a single component being "split" into two.

To summarize, the `Union-Find` data structure will support three operations.

- `MakeUnionFind`(S) for a set S will return a `Union-Find` data structure on set S where all elements are in separate sets. This corresponds, for example, to the connected components of a graph with no edges. Our goal will be to implement `MakeUnionFind` in time $O(n)$ where $n = |S|$.
- For an element $u \in S$, the operation `Find`(u) will return the name of the set containing u. Our goal will be to implement `Find`(u) in $O(\log n)$ time. Some implementations that we discuss will in fact take only $O(1)$ time for this operation.
- For two sets A and B, the operation `Union`(A, B) will change the data structure by merging the sets A and B into a single set. Our goal will be to implement `Union` in $O(\log n)$ time.

Let's briefly discuss what we mean by the *name* of a set—for example, as returned by the `Find` operation. There is a fair amount of flexibility in defining the names of the sets; they should simply be consistent in the sense that `Find`(v) and `Find`(w) should return the same name if v and w belong to the same set, and different names otherwise. In our implementations, we will name each set using one of the elements it contains.

A Simple Data Structure for Union-Find

Maybe the simplest possible way to implement a `Union-Find` data structure is to maintain an array `Component` that contains the name of the set currently containing each element. Let S be a set, and assume it has n elements denoted $\{1, \ldots, n\}$. We will set up an array `Component` of size n, where `Component`$[s]$ is the name of the set containing s. To implement `MakeUnionFind`(S), we set up the array and initialize it to `Component`$[s] = s$ for all $s \in S$. This implementation makes `Find`(v) easy: it is a simple lookup and takes only $O(1)$ time. However, `Union`(A, B) for two sets A and B can take as long as $O(n)$ time, as we have to update the values of `Component`$[s]$ for all elements in sets A and B.

To improve this bound, we will do a few simple optimizations. First, it is useful to explicitly maintain the list of elements in each set, so we don't have to look through the whole array to find the elements that need updating. Further,

we save some time by choosing the name for the union to be the name of one of the sets, say, set A: this way we only have to update the values `Component`$[s]$ for $s \in B$, but not for any $s \in A$. Of course, if set B is large, this idea by itself doesn't help very much. Thus we add one further optimization. When set B is big, we may want to keep its name and change `Component`$[s]$ for all $s \in A$ instead. More generally, we can maintain an additional array `size` of length n, where `size`$[A]$ is the size of set A, and when a `Union`(A, B) operation is performed, we use the name of the larger set for the union. This way, fewer elements need to have their `Component` values updated.

Even with these optimizations, the worst case for a `Union` operation is still $O(n)$ time; this happens if we take the union of two large sets A and B, each containing a constant fraction of all the elements. However, such bad cases for `Union` cannot happen very often, as the resulting set $A \cup B$ is even bigger. How can we make this statement more precise? Instead of bounding the worst-case running time of a single `Union` operation, we can bound the total (or average) running time of a sequence of k `Union` operations.

(4.23) *Consider the array implementation of the* `Union-Find` *data structure for some set S of size n, where unions keep the name of the larger set. The* `Find` *operation takes $O(1)$ time,* `MakeUnionFind`(S) *takes $O(n)$ time, and any sequence of k* `Union` *operations takes at most $O(k \log k)$ time.*

Proof. The claims about the `MakeUnionFind` and `Find` operations are easy to verify. Now consider a sequence of k `Union` operations. The only part of a `Union` operation that takes more than $O(1)$ time is updating the array `Component`. Instead of bounding the time spent on one `Union` operation, we will bound the total time spent updating `Component`$[v]$ for an element v throughout the sequence of k operations.

Recall that we start the data structure from a state when all n elements are in their own separate sets. A single `Union` operation can consider at most two of these original one-element sets, so after any sequence of k `Union` operations, all but at most $2k$ elements of S have been completely untouched. Now consider a particular element v. As v's set is involved in a sequence of `Union` operations, its size grows. It may be that in some of these `Union`s, the value of `Component`$[v]$ is updated, and in others it is not. But our convention is that the union uses the name of the larger set, so in every update to `Component`$[v]$ the size of the set containing v at least doubles. The size of v's set starts out at 1, and the maximum possible size it can reach is $2k$ (since we argued above that all but at most $2k$ elements are untouched by `Union` operations). Thus `Component`$[v]$ gets updated at most $\log_2(2k)$ times throughout the process. Moreover, at most $2k$ elements are involved in any `Union` operations at all, so

we get a bound of $O(k \log k)$ for the time spent updating `Component` values in a sequence of k `Union` operations. ■

While this bound on the average running time for a sequence of k operations is good enough in many applications, including implementing Kruskal's Algorithm, we will try to do better and reduce the *worst-case* time required. We'll do this at the expense of raising the time required for the `Find` operation to $O(\log n)$.

A Better Data Structure for Union-Find

The data structure for this alternate implementation uses pointers. Each node $v \in S$ will be contained in a record with an associated pointer to the name of the set that contains v. As before, we will use the elements of the set S as possible set names, naming each set after one of its elements. For the `MakeUnionFind`(S) operation, we initialize a record for each element $v \in S$ with a pointer that points to itself (or is defined as a `null` pointer), to indicate that v is in its own set.

Consider a `Union` operation for two sets A and B, and assume that the name we used for set A is a node $v \in A$, while set B is named after node $u \in B$. The idea is to have either u or v be the name of the combined set; assume we select v as the name. To indicate that we took the union of the two sets, and that the name of the union set is v, we simply update u's pointer to point to v. We do not update the pointers at the other nodes of set B.

As a result, for elements $w \in B$ other than u, the name of the set they belong to must be computed by following a sequence of pointers, first leading them to the "old name" u and then via the pointer from u to the "new name" v. See Figure 4.12 for what such a representation looks like. For example, the two sets in Figure 4.12 could be the outcome of the following sequence of `Union` operations: `Union`(w, u), `Union`(s, u), `Union`(t, v), `Union`(z, v), `Union`(i, x), `Union`(y, j), `Union`(x, j), and `Union`(u, v).

This pointer-based data structure implements `Union` in $O(1)$ time: all we have to do is to update one pointer. But a `Find` operation is no longer constant time, as we have to follow a sequence of pointers through a history of old names the set had, in order to get to the current name. How long can a `Find`(u) operation take? The number of steps needed is exactly the number of times the set containing node u had to change its name, that is, the number of times the `Component`[u] array position would have been updated in our previous array representation. This can be as large as $O(n)$ if we are not careful with choosing set names. To reduce the time required for a `Find` operation, we will use the same optimization we used before: keep the name of the larger set as the name of the union. The sequence of `Unions` that produced the data

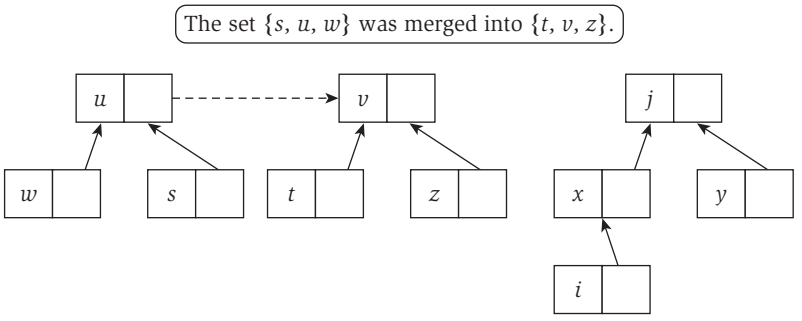

Figure 4.12 A `Union-Find` data structure using pointers. The data structure has only two sets at the moment, named after nodes v and j. The dashed arrow from u to v is the result of the last `Union` operation. To answer a `Find` query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query `Find`(i) would involve following the arrows i to x, and then x to j.

structure in Figure 4.12 followed this convention. To implement this choice efficiently, we will maintain an additional field with the nodes: the size of the corresponding set.

(4.24) *Consider the above pointer-based implementation of the* `Union-Find` *data structure for some set S of size n, where unions keep the name of the larger set. A* `Union` *operation takes $O(1)$ time,* `MakeUnionFind`(S) *takes $O(n)$ time, and a* `Find` *operation takes $O(\log n)$ time.*

Proof. The statements about `Union` and `MakeUnionFind` are easy to verify. The time to evaluate `Find`(v) for a node v is the number of times the set containing node v changes its name during the process. By the convention that the union keeps the name of the larger set, it follows that every time the name of the set containing node v changes, the size of this set at least doubles. Since the set containing v starts at size 1 and is never larger than n, its size can double at most $\log_2 n$ times, and so there can be at most $\log_2 n$ name changes. ■

Further Improvements

Next we will briefly discuss a natural optimization in the pointer-based `Union-Find` data structure that has the effect of speeding up the `Find` operations. Strictly speaking, this improvement will not be necessary for our purposes in this book: for all the applications of `Union-Find` data structures that we consider, the $O(\log n)$ time per operation is good enough in the sense that further improvement in the time for operations would not translate to improvements

in the overall running time of the algorithms where we use them. (The `Union-Find` operations will not be the only computational bottleneck in the running time of these algorithms.)

To motivate the improved version of the data structure, let us first discuss a bad case for the running time of the pointer-based `Union-Find` data structure. First we build up a structure where one of the `Find` operations takes about $\log n$ time. To do this, we can repeatedly take `Union`s of equal-sized sets. Assume v is a node for which the `Find`(v) operation takes about $\log n$ time. Now we can issue `Find`(v) repeatedly, and it takes $\log n$ for each such call. Having to follow the same sequence of $\log n$ pointers every time for finding the name of the set containing v is quite redundant: after the first request for `Find`(v), we already "know" the name x of the set containing v, and we also know that all other nodes that we touched during our path from v to the current name also are all contained in the set x. So in the improved implementation, we will *compress* the path we followed after every `Find` operation by resetting all pointers along the path to point to the current name of the set. No information is lost by doing this, and it makes subsequent `Find` operations run more quickly. See Figure 4.13 for a `Union-Find` data structure and the result of `Find`(v) using path compression.

Now consider the running time of the operations in the resulting implementation. As before, a `Union` operation takes $O(1)$ time and `MakeUnion-Find`(S) takes $O(n)$ time to set up a data structure for a set of size n. How did the time required for a `Find`(v) operation change? Some `Find` operations can still take up to $\log n$ time; and for some `Find` operations we actually increase

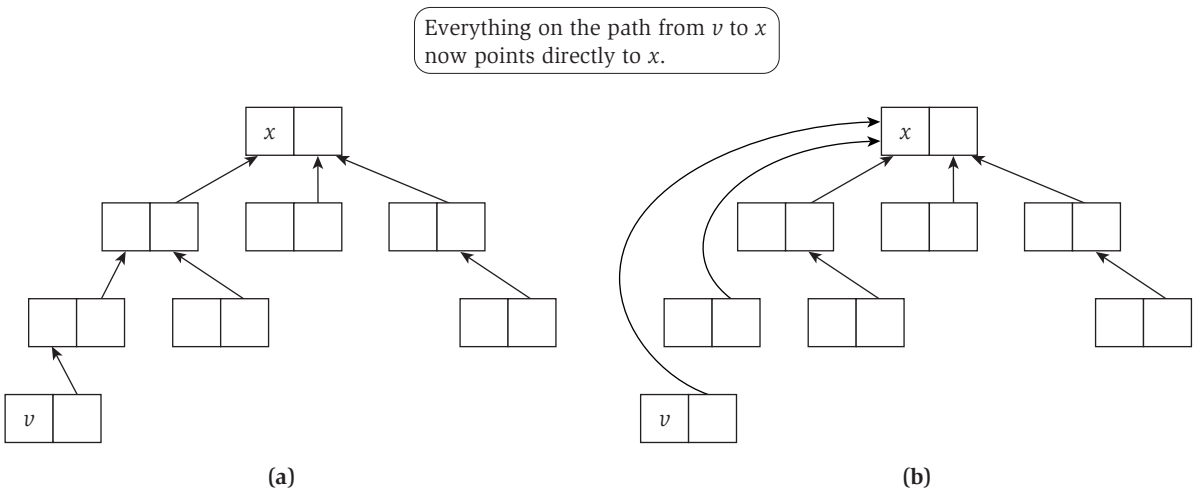

Figure 4.13 (a) An instance of a `Union-Find` data structure; and (b) the result of the operation `Find`(v) on this structure, using path compression.

the time, since after finding the name x of the set containing v, we have to go back through the same path of pointers from v to x, and reset each of these pointers to point to x directly. But this additional work can at most double the time required, and so does not change the fact that a `Find` takes at most $O(\log n)$ time. The real gain from compression is in making subsequent calls to `Find` cheaper, and this can be made precise by the same type of argument we used in (4.23): bounding the total time for a sequence of n `Find` operations, rather than the worst-case time for any one of them. Although we do not go into the details here, a sequence of n `Find` operations employing compression requires an amount of time that is extremely close to linear in n; the actual upper bound is $O(n\alpha(n))$, where $\alpha(n)$ is an extremely slow-growing function of n called the *inverse Ackermann function*. (In particular, $\alpha(n) \leq 4$ for any value of n that could be encountered in practice.)

Implementing Kruskal's Algorithm

Now we'll use the `Union-Find` data structure to implement Kruskal's Algorithm. First we need to sort the edges by cost. This takes time $O(m \log m)$. Since we have at most one edge between any pair of nodes, we have $m \leq n^2$ and hence this running time is also $O(m \log n)$.

After the sorting operation, we use the `Union-Find` data structure to maintain the connected components of (V, T) as edges are added. As each edge $e = (v, w)$ is considered, we compute `Find`(u) and `Find`(v) and test if they are equal to see if v and w belong to different components. We use `Union(Find`(u),`Find`$(v))$ to merge the two components, if the algorithm decides to include edge e in the tree T.

We are doing a total of at most $2m$ `Find` and $n - 1$ `Union` operations over the course of Kruskal's Algorithm. We can use either (4.23) for the array-based implementation of `Union-Find`, or (4.24) for the pointer-based implementation, to conclude that this is a total of $O(m \log n)$ time. (While more efficient implementations of the `Union-Find` data structure are possible, this would not help the running time of Kruskal's Algorithm, which has an unavoidable $O(m \log n)$ term due to the initial sorting of the edges by cost.)

To sum up, we have

(4.25) *Kruskal's Algorithm can be implemented on a graph with n nodes and m edges to run in $O(m \log n)$ time.*

4.7 Clustering

We motivated the construction of minimum spanning trees through the problem of finding a low-cost network connecting a set of sites. But minimum

spanning trees arise in a range of different settings, several of which appear on the surface to be quite different from one another. An appealing example is the role that minimum spanning trees play in the area of *clustering*.

The Problem

Clustering arises whenever one has a collection of objects—say, a set of photographs, documents, or microorganisms—that one is trying to classify or organize into coherent groups. Faced with such a situation, it is natural to look first for measures of how similar or dissimilar each pair of objects is. One common approach is to define a *distance function* on the objects, with the interpretation that objects at a larger distance from one another are less similar to each other. For points in the physical world, distance may actually be related to their physical distance; but in many applications, distance takes on a much more abstract meaning. For example, we could define the distance between two species to be the number of years since they diverged in the course of evolution; we could define the distance between two images in a video stream as the number of corresponding pixels at which their intensity values differ by at least some threshold.

Now, given a distance function on the objects, the clustering problem seeks to divide them into groups so that, intuitively, objects within the same group are "close," and objects in different groups are "far apart." Starting from this vague set of goals, the field of clustering branches into a vast number of technically different approaches, each seeking to formalize this general notion of what a good set of groups might look like.

Clusterings of Maximum Spacing Minimum spanning trees play a role in one of the most basic formalizations, which we describe here. Suppose we are given a set U of n objects, labeled $p_1, p_2, \ldots, p_n$. For each pair, p_i and p_j, we have a numerical distance $d(p_i, p_j)$. We require only that $d(p_i, p_i) = 0$; that $d(p_i, p_j) > 0$ for distinct p_i and p_j; and that distances are symmetric: $d(p_i, p_j) = d(p_j, p_i)$.

Suppose we are seeking to divide the objects in U into k groups, for a given parameter k. We say that a *k-clustering* of U is a partition of U into k nonempty sets $C_1, C_2, \ldots, C_k$. We define the *spacing* of a k-clustering to be the minimum distance between any pair of points lying in different clusters. Given that we want points in different clusters to be far apart from one another, a natural goal is to seek the k-clustering with the maximum possible spacing.

The question now becomes the following. There are exponentially many different k-clusterings of a set U; how can we efficiently find the one that has maximum spacing?

"long way" around C (avoiding e) can be viewed as an alternate route between the endpoints of e that only uses cheaper edges.

Putting these two observations together suggests that we should try proving the following statement.

(4.41) *Edge $e = (v, w)$ does not belong to a minimum spanning tree of G if and only if v and w can be joined by a path consisting entirely of edges that are cheaper than e.*

Proof. First suppose that P is a v-w path consisting entirely of edges cheaper than e. If we add e to P, we get a cycle on which e is the most expensive edge. Thus, by the Cycle Property, e does not belong to a minimum spanning tree of G.

On the other hand, suppose that v and w cannot be joined by a path consisting entirely of edges cheaper than e. We will now identify a set S for which e is the cheapest edge with one end in S and the other in $V - S$; if we can do this, the Cut Property will imply that e belongs to every minimum spanning tree. Our set S will be the set of all nodes that are reachable from v using a path consisting only of edges that are cheaper than e. By our assumption, we have $w \in V - S$. Also, by the definition of S, there cannot be an edge $f = (x, y)$ that is cheaper than e, and for which one end x lies in S and the other end y lies in $V - S$. Indeed, if there were such an edge f, then since the node x is reachable from v using only edges cheaper than e, the node y would be reachable as well. Hence e is the cheapest edge with one end in S and the other in $V - S$, as desired, and so we are done. ■

Given this fact, our algorithm is now simply the following. We form a graph G' by deleting from G all edges of weight greater than c_e (as well as deleting e itself). We then use one of the connectivity algorithms from Chapter 3 to determine whether there is a path from v to w in G'. Statement (4.41) says that e belongs to a minimum spanning tree if and only if there is no such path.

The running time of this algorithm is $O(m + n)$ to build G', and $O(m + n)$ to test for a path from v to w.

Exercises

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary connected, undirected graph with a distinct cost $c(e)$ on every edge e. Suppose e^* is the cheapest edge in G; that is, $c(e^*) < c(e)$ for every

edge $e \neq e^*$. Then there is a minimum spanning tree T of G that contains the edge e^*.

2. For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? T must still be a minimum spanning tree for this new instance.

(b) Suppose we are given an instance of the Shortest s-t Path Problem on a directed graph G. We assume that all edge costs are positive and distinct. Let P be a minimum-cost s-t path for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

True or false? P must still be a minimum-cost s-t path for this new instance.

3. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

4. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought—are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

```
buy Yahoo, buy eBay, buy Yahoo, buy Oracle
```

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

```
buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo,
buy Oracle
```

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S' of length m and S of length n, each possibly containing an event more than once—and decides in time $O(m + n)$ whether S' is a subsequence of S.

5. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

7. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be *preprocessed* on the supercomputer, and then it needs to be *finished* on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel—all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job

in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

8. Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

9. One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum *total* cost. Here we explore another type of objective: designing a spanning network for which the *most expensive* edge is as cheap as possible.

Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost.

A spanning tree T of G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

10. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.

(a) Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree T and the graph G.