Long after the adoption of LRU in practice, Sleator and Tarjan showed that one could actually provide some theoretical analysis of the performance of LRU, bounding the number of misses it incurs relative to Farthest-in-Future. We will discuss this analysis, as well as the analysis of a randomized variant on LRU, when we return to the caching problem in Chapter 13.

## 4.4 Shortest Paths in a Graph

Some of the basic algorithms for graphs are based on greedy design principles. Here we apply a greedy algorithm to the problem of finding shortest paths, and in the next section we look at the construction of minimum-cost spanning trees.

### The Problem

As we've seen, graphs are often used to model networks in which one travels from one point to another—traversing a sequence of highways through interchanges, or traversing a sequence of communication links through intermediate routers. As a result, a basic algorithmic problem is to determine the shortest path between nodes in a graph. We may ask this as a point-to-point question: Given nodes $u$ and $v$, what is the shortest $u$-$v$ path? Or we may ask for more information: Given a *start node* $s$, what is the shortest path from $s$ to each other node?

The concrete setup of the shortest paths problem is as follows. We are given a directed graph $G = (V, E)$, with a designated start node $s$. We assume that $s$ has a path to every other node in $G$. Each edge $e$ has a length $\ell_e \geq 0$, indicating the time (or distance, or cost) it takes to traverse $e$. For a path $P$, the *length of* $P$—denoted $\ell(P)$—is the sum of the lengths of all edges in $P$. Our goal is to determine the shortest path from $s$ to every other node in the graph. We should mention that although the problem is specified for a directed graph, we can handle the case of an undirected graph by simply replacing each undirected edge $e = (u, v)$ of length $\ell_e$ by two directed edges $(u, v)$ and $(v, u)$, each of length $\ell_e$.

### Designing the Algorithm

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest-paths problem. We begin by describing an algorithm that just determines the *length* of the shortest path from $s$ to each other node in the graph; it is then easy to produce the paths as well. The algorithm maintains a set $S$ of vertices $u$ for which we have determined a shortest-path distance $d(u)$ from $s$; this is the "explored" part of the graph. Initially $S = \{s\}$, and $d(s) = 0$. Now, for each node $v \in V - S$, we determine the shortest path that can be constructed by traveling along a path through the explored part $S$ to some $u \in S$, followed by the single edge $(u, v)$. That is, we consider the quantity

$d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$. We choose the node $v \in V - S$ for which this quantity is minimized, add $v$ to $S$, and define $d(v)$ to be the value $d'(v)$.

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

It is simple to produce the $s$-$u$ paths corresponding to the distances found by Dijkstra's Algorithm. As each node $v$ is added to the set $S$, we simply record the edge $(u, v)$ on which it achieved the value $\min_{e=(u,v):u \in S} d(u) + \ell_e$. The path $P_v$ is implicitly represented by these edges: if $(u, v)$ is the edge we have stored for $v$, then $P_v$ is just (recursively) the path $P_u$ followed by the single edge $(u, v)$. In other words, to construct $P_v$, we simply start at $v$; follow the edge we have stored for $v$ in the reverse direction to $u$; then follow the edge we have stored for $u$ in the reverse direction to its predecessor; and so on until we reach $s$. Note that $s$ must be reached, since our backward walk from $v$ visits nodes that were added to $S$ earlier and earlier.

To get a better sense of what the algorithm is doing, consider the snapshot of its execution depicted in Figure 4.7. At the point the picture is drawn, two iterations have been performed: the first added node $u$, and the second added node $v$. In the iteration that is about to be performed, the node $x$ will be added because it achieves the smallest value of $d'(x)$; thanks to the edge $(u, x)$, we have $d'(x) = d(u) + l_{ux} = 2$. Note that attempting to add $y$ or $z$ to the set $S$ at this point would lead to an incorrect value for their shortest-path distances; ultimately, they will be added because of their edges from $x$.

## Analyzing the Algorithm

We see in this example that Dijkstra's Algorithm is doing the right thing and avoiding recurring pitfalls: growing the set $S$ by the wrong node can lead to an overestimate of the shortest-path distance to that node. The question becomes: Is it always true that when Dijkstra's Algorithm adds a node $v$, we get the true shortest-path distance to $v$?

We now answer this by proving the correctness of the algorithm, showing that the paths $P_u$ really are shortest paths. Dijkstra's Algorithm is greedy in

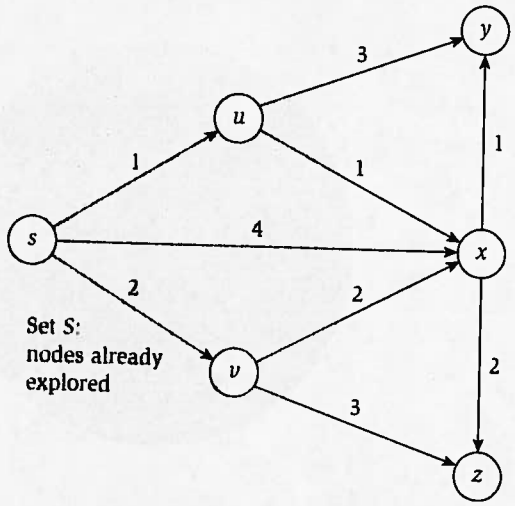


**Figure 4.7** A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set $S$ is $x$, due to the path through $u$.

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

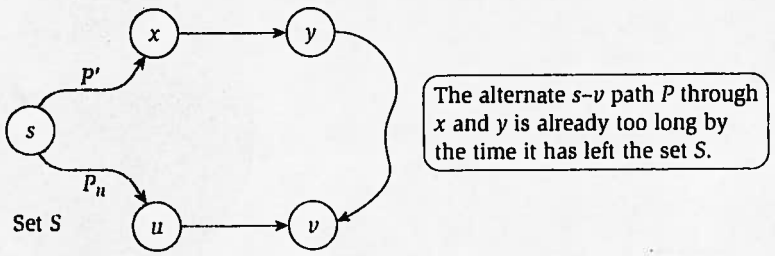


**Figure 4.8** The shortest path $P_v$ and an alternate $s$-$v$ path $P$ through the node $y$.

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 $\ell_e$ 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 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 desi 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