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
Chapter 4  Greedy Algorithms

\[ d'(v) = \min_{e=(u,v) \in E} d(u) + \ell_e. \]  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 (\( G, \ell \))

Let \( S \) be the set of explored nodes

For each \( u \in S \), we store a distance \( d(u) \)

Initially \( S = \{s\} \) and \( d(s) = 0 \)

While \( S \neq V \)

Select a node \( v \notin S \) with at least one edge from \( S \) for which

\[ d'(v) = \min_{e=(u,v) \in E} d(u) + \ell_e \]  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 \( \min_{e=(u,v) \in E} d(u) + \ell_e \). 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 \( d'(x) = d(u) + \ell_{ux} = 2 \). 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