9

GRAPHS

The origins of graph theory are humble, even frivolous.
—Norman L. Biggs (1941–)

To many, the word *graph* implies the *x* versus *y* kind. This is not, as you'll learn in this chapter, what computer scientists mean when they use the term.

Graph theory, the formal name for this chapter's topic, has a long history in mathematics stretching as far back as Euler. Modern graph applications are legion because of a graph's ability to represent relational knowledge between entities. For example, a road map showing the distances between cities is a graph, as are social networks showing who knows who. The internet itself is a gigantic graph indicating how each computer is connected to every other computer.

In this chapter, we focus on practical algorithms that a professional software engineer will likely encounter. All functions and variables referenced are in the `graphs` module, which you'll find on the book's GitHub site.

We begin with essential graph concepts and terminology, before investigating how graphs are represented in code. Elementary graph algorithms come next, beginning with breadth-first and depth-first searches. We then explore algorithms for finding the shortest path between two graph nodes.

We end the chapter by contemplating directed acyclic graphs (DAGs). These popular graphs are usually known by another name: neural networks,

the backbone of modern AI systems. Because this use is sadly beyond our scope, we conclude with a less society-altering use for DAGs: topological sorting. This technique tells us the order in which tasks must be completed to ensure that task outputs are available as inputs when needed. The scheduling application is obvious.

Basic Graph Concepts

A *graph*, G, is a set of *vertices*, V, connected by a set of *edges*, E; each edge links a pair of vertices. A *vertex* (the singular of *vertices*) is also called a *node*, and I'll use both terms interchangeably throughout the chapter, favoring *node* because it's most often used in programming. Collectively, a graph is denoted $G(V, E)$. Small graphs are often visualized as in Figure 9-1.

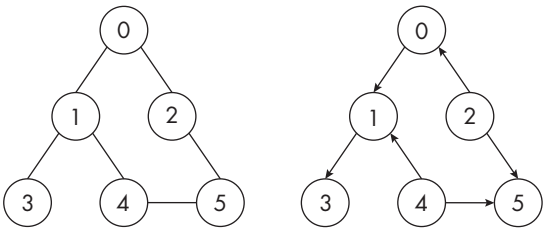

Figure 9-1: A simple graph with six nodes (left) and a directed version (right)

Each of these two graphs has six nodes, labeled 0 through 5. Edges are the line segments between nodes. If two nodes have an edge between them, they are *adjacent*.

Graphs are generic representations of relationships among entities. For example, on the left-hand graph in Figure 9-1, the entity represented by node 0 shares a relationship with nodes 1 and 2, as the edge between them indicates. The right-hand graph shows that node 0 leads to node 1, while node 2 leads to node 0 (follow the arrows).

The caption for Figure 9-1 states that the graph on the left is simple. A *simple graph* has no self-loops, meaning no node is connected to itself. Additionally, only one edge is between any pair of nodes, and the edges have no direction; that is, a simple graph is an *undirected graph*. In a *directed graph*, like the one on the right in Figure 9-1, edges have a direction indicated by arrows.

Graphs are *sets* of nodes and edges. What are the sets for the graphs in Figure 9-1? Let's begin with the nodes common to both graphs:

$$V = \{0, 1, 2, 3, 4, 5\}$$

To represent the edges, we need a set of pairs that tells us which nodes go together. For the undirected graph on the left, we write this:

$$E = \{\{0, 1\}, \{0, 2\}, \{1, 3\}, \{1, 4\}, \{2, 5\}, \{4, 5\}\}$$

Notice that E is a set of sets. Edges in an undirected graph link vertices without direction, so we write $\{0, 1\}$ to indicate the edge between nodes 0 and 1.

Directed graphs use a different notation. For the right-hand side of Figure 9-1, we get the following:

$$E = \{(0, 1), (1, 3), (2, 0), (2, 5), (4, 1), (4, 5)\}$$

Here we have a set of tuples, pairs showing the originating node followed by the destination node.

The following are additional graph concepts that are worth remembering:

- The *degree* of an undirected vertex is the number of edges connected to it. For example, vertex 1 on the left in Figure 9-1 has degree 3.
- For directed graphs, the degree can be *in-degree* (for arrows ending at the vertex) or *out-degree* (for arrows originating with the vertex). Vertex 1 on the right in Figure 9-1 has in-degree 2 and out-degree 1.
- A *path* is a sequence of nodes telling us how to get from node A to node B, assuming such a sequence is possible.
- A *cycle* is a path where node A and node B are the same; that is, the path starts and ends at the same node.
- Undirected graphs are labeled *connected* if a path leads from any node to any other node.
- A directed graph is *strongly connected* if, for every pair, each of its vertices is reachable from the other. It is possible to go from vertex A to vertex B and from B to A for any pair of vertices in the graph.
- Undirected graphs are labeled *complete* if each node is connected to every other node.
- A *subgraph* of graph G is a graph formed from a subset of the vertices of G and the edges from G that connect only those vertices. For example, $V = \{1, 2, 4, 5\}$ and $E = \{\{1, 4\}, \{2, 5\}, \{4, 5\}\}$ is a subgraph of the leftmost graph in Figure 9-1.
- A graph with no vertices or edges is a *null graph*.

The vertices of the graphs in Figure 9-1 have numeric labels as stand-ins for more meaningful collections of data that the nodes would, in practice, represent. The edges, however, are unlabeled. This need not be the case. If a graph's edges are labeled, typically with a number, this means we're working with a *weighted graph*, which may be undirected or directed.

Figure 9-2 adds numbers to the edges of Figure 9-1. These are the weights.

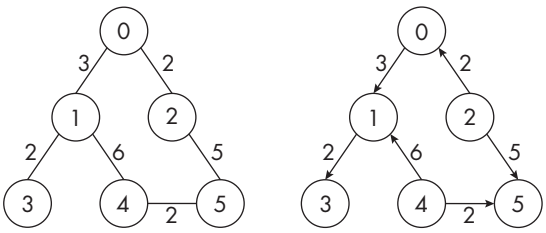

Figure 9-2: A weighted version of Figure 9-1

What the weights represent is problem specific. For nodes that indicate tasks, the weight might be the cost of a task or the time to finish it. Weights could also represent distances between cities or the strength of connections between nodes in a neural network.

Complete Graphs

A *complete graph* includes every possible edge between the vertices. When the vertices are arranged equidistantly around a circle, the resulting graph takes on the appearance of a regular polygon with a pleasing set of lines between the points. We call such graphs *mystic roses*.

The Python code in *mystic.py* first finds the coordinates of a user-supplied number of points on the unit circle, then plots them along with every possible edge. The code isn't difficult to follow, so I won't list it here, but when run with the number of points on the command line and an optional output base filename for the plot, the result is similar to the graphs in Figure 9-3.

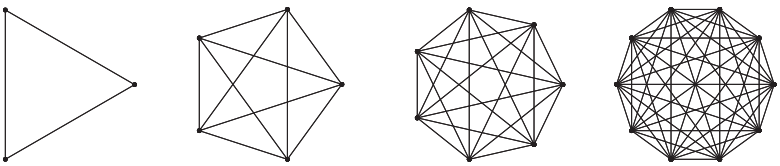

Figure 9-3: The complete graphs for 3, 5, 7, and 10 vertices

If a complete graph has n vertices, we denote it as K_n. Therefore, Figure 9-3 shows K_3, K_5, K_7, and K_{10}. A complete graph with n vertices has degree $n - 1$ and $n(n-1)/2$ edges. I can write that the graph has a degree in this case because every vertex has the same degree. It is connected to the $n - 1$ vertices that are not itself.

A *clique*, C, of an undirected graph, G, is a subset of vertices, $C \subseteq V$, such that every vertex in C is adjacent to every other vertex in C. In other words, C is a complete subgraph of G. The K_n graphs are complete for all vertices; therefore, they form a clique, specifically, an n-vertex clique. Deciding whether a given graph has a clique of a given size is a difficult problem.

Graph Isomorphisms

Consider the three graphs in Figure 9-4, the leftmost of which is copied from Figure 9-1. A few moments of contemplation using the labels should convince you that all three represent the same graph; each has the same set of vertices and edges. The three graphs are structurally the same, meaning they are *isomorphic* and that there is an *isomorphism* mapping one to the other. The word *isomorphic* comes from the Greek *iso-*, meaning *equal*, and *morphic*, meaning *shape*.

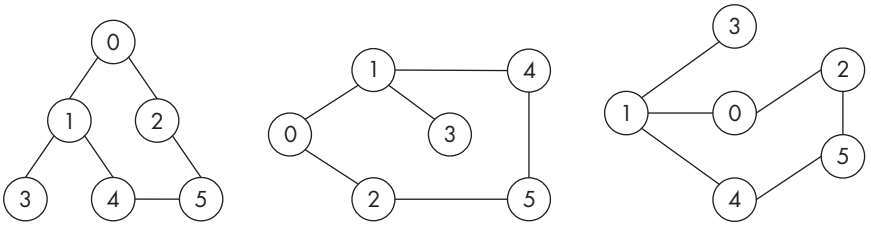

Figure 9-4: Three isomorphic graphs

We can quickly determine that the graphs of Figure 9-4 are isomorphic—doubly so because the nodes share the same labels. However, determining whether two graphs are isomorphic is typically a hard problem, though perhaps not as hard as deciding whether a specific-sized clique is present. Fortunately, certain graph types have efficient, though nontrivial, isomorphism algorithms. Two such graph types are trees, the subject of Chapter 10, and planar graphs. A *planar graph* can be drawn in the 2D plane such that edges do not cross. All the graphs in Figure 9-4 are planar. Graph K_3 on the left of Figure 9-3 is planar, but the other complete graphs are not.

Graph isomorphism separates the set of all graphs into equivalence classes. We explored equivalence classes in Chapter 4 and again in Chapter 7 when discussing congruence classes. Recall the definition of an equivalence class:

$$\overline{a} = \{b \in \mathbb{Z} \mid a \equiv b \pmod{n}\}$$

Graph isomorphism places all graphs into disjoint sets where the graphs of each set are isomorphic. Therefore, all the graphs in Figure 9-4 are in the same equivalence class and are functionally identical.

We can demonstrate that two graphs are isomorphic by a brute-force relabeling of the nodes. Our graphs use integer labels, so if we can find a permutation of the labels of one graph that match another, we know that the graphs are isomorphic. Chapter 8 taught you that the number of permutations of n things is $n!$, meaning a brute-force isomorphism algorithm for graphs A and B must check up to $n!$ label permutations to find one that matches. A small graph with only six nodes requires up to 6! = 720 permutations, but expand the graph to 40 nodes, and the number of permutations explodes:

$$40! = 815{,}915{,}283{,}247{,}897{,}734{,}345{,}611{,}269{,}596{,}115{,}894{,}272{,}000{,}000{,}000$$

Clearly, brute force isn't the way to go. The `graphs` module includes `Isomorphic`, a brute-force function to determine whether two graphs are isomorphic. I leave experimenting with the code as an exercise, but you'll likely want to review the following section before you do.

Graph theory's long and rich history leaves many other terms and concepts to learn, but we have what we need for our purposes. We'll therefore press on to consider representing graphs in code so we can implement and understand fundamental graph algorithms. Graphs as images may look nice, but computers can't efficiently work with them in that form.

Representing Graphs in Code

We often represent graphs in code as *adjacency lists* or *adjacency matrices*. We'll explore both in Python, beginning with adjacency lists.

Adjacency Lists

Look again at the leftmost graph in Figure 9-1. It is an undirected graph that we can represent as a list of sets:

```
[{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1, 5}, {2, 4}]
```

The graph has six vertices labeled 0 through 5, so the list has six elements, and order matters. The first element represents the nodes that are connected to the first node (node 0), the second element represents the nodes that are connected to the second node (node 1), and so forth. For example, node 0 connects to nodes 1 and 2; therefore, the first element of the adjacency list is the set `{1,2}`. Similarly, node 1 connects to nodes 0, 3, and 4, making the second element of the adjacency list `{0,3,4}`. The pattern continues for all the vertices. Notice that node 0 lists node 1 as adjacent, while node 1 lists node 0 as adjacent—both directions appear in the adjacency list for an undirected graph.

The directed version on the right in Figure 9-1 is represented in much the same way:

```
[{1}, {3}, {0, 5}, set(), {1, 5}, set()]
```

Again, we use a list of sets to tell us that node 0 is connected to node 1, node 1 is connected to node 3, node 2 is connected to nodes 0 and 5, and node 4 leads to nodes 1 and 5. Nodes 3 and 5 lead nowhere, as indicated by the empty set, `set()`. We must use `set()` because Python interprets `{}` as an empty dictionary. Dictionaries are far more common in Python than sets, so the concession is appropriate for us, if a little clunky.

Weighted graphs require us to include the edge weight. Therefore, a weighted undirected graph, like the one on the left in Figure 9-2, becomes the following:

```
[{(1, 3), (2, 2)}, {(0, 3), (3, 2), (4, 6)}, {(0, 2), (5, 5)}, {(1, 2)},
 {(1, 6), (5, 2)}, {(2, 5), (4, 2)}]
```

The elements of a weighted graph use tuples to represent the node to which the current node is connected, followed by the weight of that edge. Therefore, node 0 is connected to node 1 by an edge with a weight of 3, and node 2 by an edge with weight 2, and so on.

Weighted directed graphs follow the natural extension of an unweighted directed graph:

```
[{(1, 3)}, {(3, 2)}, {(0, 2), (5, 5)}, set(), {(1, 6), (5, 2)}, set()]
```

Nodes leading nowhere are represented by the empty set, `set()`.

Adjacency Matrices

The algorithms we'll discuss later in the chapter expect graphs as adjacency lists, but storing a graph as a matrix can be helpful. For example, to know whether node i is connected to node j, simply check whether the (i, j) matrix element is nonzero.

The matrix has as many rows as columns, one each for each node; therefore, adjacency matrices are square. If we have n nodes, the adjacency matrix is $n \times n$. For example, we represent the leftmost graph in Figure 9-1 as an adjacency matrix like so:

```
[[0, 1, 1, 0, 0, 0],
 [1, 0, 0, 1, 1, 0],
 [1, 0, 0, 0, 0, 1],
 [0, 1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 1],
 [0, 0, 1, 0, 1, 0]]
```

I'm showing the matrix as a list of lists arranged in 2D because we're discussing graph representation in code.

The first row indicates the nodes to which node 0 is connected. We know that node 0 connects to nodes 1 and 2, and we see this in the matrix as a `1` in columns 1 and 2. (Recall that we index matrices from zero in this book, following typical programming convention.) Similarly, according to the matrix, node 2 is connected to nodes 0 and 5, and so on.

If `1` indicates a connection and `0` indicates no connection, we might represent a weighted graph by using the weight as the value, reserving `0` for no connection. This convention implies positive weights only, which works for us in this chapter. For example, the unweighted and weighted versions of the left-hand side of Figure 9-1 are as follows:

```
[[0, 1, 1, 0, 0, 0],          [[0, 3, 2, 0, 0, 0],
 [1, 0, 0, 1, 1, 0],           [3, 0, 0, 2, 6, 0],
 [1, 0, 0, 0, 0, 1],    ==>    [2, 0, 0, 0, 0, 5],
 [0, 1, 0, 0, 0, 0],           [0, 3, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 1],           [0, 6, 0, 0, 0, 2],
 [0, 0, 1, 0, 1, 0]]           [0, 0, 5, 0, 2, 0]]
```

For directed graphs, the right-hand side of Figure 9-1 becomes the following:

```
[[0, 1, 0, 0, 0, 0],          [[0, 3, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 0],           [0, 0, 0, 2, 0, 0],
 [1, 0, 0, 0, 0, 1],    ==>    [2, 0, 0, 0, 0, 5],
 [0, 0, 0, 0, 0, 0],           [0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 1],           [0, 6, 0, 0, 0, 2],
 [0, 0, 0, 0, 0, 0]]           [0, 0, 0, 0, 0, 0]]
```

A few details are worth noting about these adjacency matrices. First, for undirected graphs, the matrices are symmetric along the main diagonal.

Imagine a line running along the diagonal from the upper left to the lower right. If you fold the matrix along this line, the corresponding elements of the matrix on either side will line up. Alternatively, imagine flipping the matrix so that row 0 is now column 0. This is a *matrix transpose*, which we'll encounter again in Chapter 13. The transpose of a symmetric matrix is the same as the original matrix.

It makes sense that the adjacency matrix of an undirected graph is symmetric. After all, I emphasized that the adjacency list contains the edges twice, from node A to node B and again from B to A. This requirement makes the adjacency matrix symmetric. Finally, notice that the weighted and unweighted versions of the matrix are the same in shape and in which elements are nonzero, with the only difference being that the weights are used in place of `1` to indicate an edge between two nodes.

Now consider the adjacency matrices for the directed graphs. The weights are handled in the same manner as for an undirected graph, but the matrices are no longer symmetric. This also makes sense, because while the matrix might indicate a connection of weight 3 between nodes 0 and 1, the arrow doesn't go the other way, so node 1 isn't connected to node 0. Therefore, the matrix element at $(0, 1)$ is 3 and the element at $(1, 0)$ is 0.

The `graphs` module contains several example graphs as global variables. For example, `A` is the undirected graph on the left in Figure 9-1:

```
#  Three isomorphic graphs (first is left of Figure 9-1)
A = [{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1, 5}, {2, 4}]
B = [{3}, {4, 5}, {3, 4}, {0, 2, 5}, {1, 2}, {1, 3}]
C = [{1, 2, 5}, {0, 3}, {0}, {1, 4}, {3, 5}, {0, 4}]

#  A graph with the same number of nodes that is not isomorphic to A, B, C
D = [{1}, {0, 4}, {4, 5}, {4}, {1, 2, 3}, {2}]

#  A directed graph (right of Figure 9-1)
E = [{1}, {3}, {0, 5}, set(), {1, 5}, set()]

#  Weighted undirected graph (weighted version of A)
F = [{(1, 3), (2, 2)}, {(0, 3), (3, 2), (4, 6)}, {(0, 2), (5, 5)}, {(1, 3)},
     {(1, 6), (5, 2)}, {(2, 5), (4, 2)}]

#  Weighted directed graph (weighted version of E)
G = [{(1, 3)}, {(3, 2)}, {(0, 2), (5, 5)}, set(), {(1, 6), (5, 2)}, set()]
```

We'll use these graphs from time to time throughout the chapter.

The `graphs` module also includes two utility functions to map between adjacency lists and adjacency matrices. For example:

```
>>> from graphs import *
>>> A
[{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1, 5}, {2, 4}]
```

```
>>> ListToMatrix(A)
[[0, 1, 1, 0, 0, 0], [1, 0, 0, 1, 1, 0], [1, 0, 0, 0, 0, 1],
 [0, 1, 0, 0, 0, 0], [0, 1, 0, 0, 0, 1], [0, 0, 1, 0, 1, 0]]
>>> MatrixToList(ListToMatrix(A))
[{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1, 5}, {2, 4}]
```

Use the `ListToMatrix` function to turn an adjacency list graph into an adjacency matrix presented as a list of lists. The `MatrixToList` function undoes the process. Both functions are straightforward passes through the respective representations to reconstruct the other with necessary checks to handle all four graph formats: undirected/directed and unweighted/weighted.

Now that you know how to represent graphs in code, let's use this knowledge to understand breadth-first and depth-first search, two foundational graph algorithms.

Breadth-First and Depth-First Traversal and Searching

Many advanced graph algorithms are enhancements or variations of one of two fundamental algorithms: breadth-first traversal or depth-first traversal. For example, minor tweaks turn the traversal algorithms into graph search algorithms, as you'll discover in this section. Let's dive in.

Breadth-First Traversal

Breadth-first traversal, often called *breadth-first search (BFS)*, regardless of whether it's searching, accepts a graph and a starting node. It then follows a simple algorithm to move through the graph until all nodes that can be reached from the starting node have been visited. We'll denote the algorithm as BFS, knowing that, at least initially, we won't be using it to search a graph. BFS is best understood in code, so we begin there. Then we'll walk through examples using the graphs of Figure 9-1.

At its simplest, BFS is only a few lines of code, assuming graphs are stored as adjacency lists (see Listing 9-1).

```
def BreadthFirst(graph, start):
    visited, queue = [start], [start]
    while queue:
        node = queue.pop(0)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
    return visited
```

Listing 9-1: Breadth-first traversal

This version of `BreadthFirst` works with undirected and directed graphs. If the graphs are weighted, the `for` line becomes

```
for neighbor, weight in graph[node]:
```

to account for the weight stored along with the edge endpoint for a node. The weight is ignored but must be read.

The code accepts the adjacency list (`graphs`) and the `starting` node, which is an integer label. The starting node label initializes two lists: `visited` and `queue`. The `visited` list contains the nodes in the order BFS visits them. It's what `BreadthFirst` returns. The `queue` list is just that, a queue, which is a first-in, first-out (FIFO) data structure.

The body of `BreadthFirst` loops until `queue` is empty. Python considers a non-empty list to be `True`. The `while` loop extracts the first element of the `queue` list with the call to `pop`. Typically, `pop` removes and returns the last element of a list, but supplying a specific index—here, zero—removes and then returns the first element.

Because `graph` is an adjacency list, `graph[node]` returns the set of nodes that are neighbors of `node`. BFS wants to look at these neighbors one by one. If the current `neighbor` isn't already in `visited`, it is added to `visited` and pushed onto the end of the `queue`.

Repeating this process until the `queue` is empty eventually reaches every node that can be accessed from the starting node. Exploring all the neighbors of a node before moving on to those neighbors that haven't been previously encountered gives the BFS algorithm its name. The current node's breadth is explored instead of diving deep along a particular path. (Diving deep implies a depth-first search.)

BFS looks first at the current node's neighbors, adding new ones to the visited list and pushing them on the queue so that the neighbors are visited in turn. BFS expands in a wave from the current node, like a ripple growing in all directions from a stone dropped in a pond.

For example, consider the following, where I apply `BreadthFirst` to the leftmost graph in Figure 9-1, which the `graphs` module stores in list `A`:

```
>>> A
[{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1, 5}, {2, 4}]
>>> BreadthFirst(A, 3)
[3, 1, 0, 4, 2, 5]
```

The returned path through the graph begins with node 3, as it must, then proceeds to nodes 1, 0, 4, 2, and ends with 5. Figure 9-5 visualizes this process beginning in the upper left and running left to right, top to bottom.

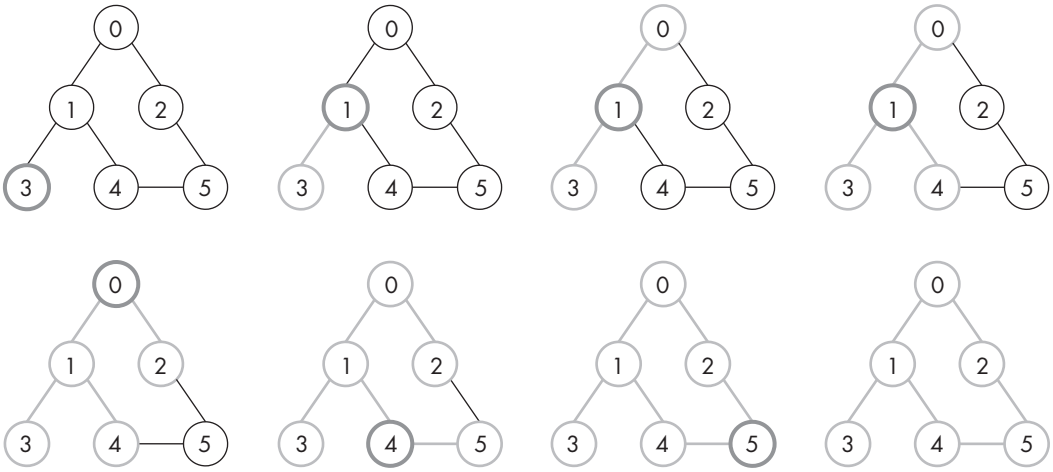

Figure 9-5: A breadth-first traversal beginning with node 3 (left to right, top to bottom)

Each graph shows the current node, whose neighbors are being explored, in thick bold. Visited nodes are bold, as are the edges connecting them to parts of the graph that have been explored.

Therefore, the upper-left graph shows node 3 in thick bold. The neighbor of node 3 is node 1, which is pushed on the queue because it hasn't yet been visited. The next graph over to the right shows node 1 as the current node. Its neighbors will be explored. This includes node 3, but node 3 is already on the visited list, so BFS moves on to nodes 0 and 4, the two rightmost graphs in the top row of Figure 9-5. The process repeats, neighbor by neighbor of the current node, until, ultimately, all nodes have been visited.

Tracing the iterations of the `while` loop (Listing 9-1) gives us another approach to BFS if we track the state of the `queue` and `visited` lists, as Table 9-1 shows.

Table 9-1: Tracing BFS

Iteration	Queue	Visited
0	3	3
1	1	3, 1
2	0	3, 1, 0
3	0, 4	3, 1, 0, 4
4	4	3, 1, 0, 4, 2
5	5	3, 1, 0, 4, 2, 5
6	—	3, 1, 0, 4, 2, 5

The structure of graph `A` is such that only node 1 has three neighbors, 0, 3, and 4. Because the test case begins with node 3, when node 1 is the current node (`graph[node]`), neither node 0 nor node 4 have been explored,

which is why iteration 3 in Table 9-1 shows both nodes in the queue waiting for their turn. Every other node has, at most, only one node that hasn't already been visited.

The BFS algorithm is just as happy to traverse directed graphs. The rightmost graph in Figure 9-1 is a directed version of `A`. It's in `E`, as this example demonstrates:

```
>>> E
[{1}, {3}, {0, 5}, set(), {1, 5}, set()]
>>> BreadthFirst(E, 3)
[3]
```

Here, `BreadthFirst` returns a single visited node, the node we started with. This makes sense because node 3 has no outgoing arrows, so BFS has nowhere to go. Let's see what happens if we start with other nodes:

```
>>> [BreadthFirst(E, i) for i in range(6)]
[[0, 1, 3], [1, 3], [2, 0, 5, 1, 3], [3], [4, 1, 5, 3], [5]]
```

Take a moment to convince yourself that these results make sense. The longest path is the one that begins on node 2 because from node 2, it's possible to get to every node except 4, as the left-hand side of Figure 9-6 illustrates.

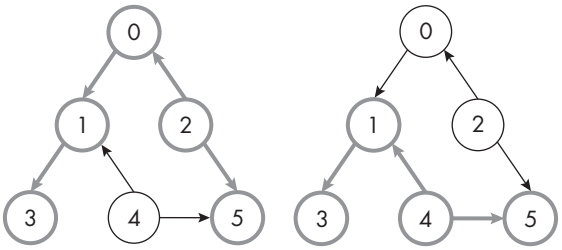

Figure 9-6: BFS for a directed graph beginning with node 2 (left) and node 4 (right)

Node 4 has no inbound arrows, meaning there's no way to get to node 4 without beginning the traversal there (the right-hand side of Figure 9-6).

Now that you have a handle on BFS, let's move on to DFS.

Depth-First Traversal

While breadth-first traversal's motto is "visit all the neighbors, then their neighbors" (Listing 9-1 does this via the loop over a current node's neighbors and the queue of who to visit next), *depth-first traversal*, or *DFS*, follows the motto of "go as deep as you can, then back up and repeat." The code illustrates the process nicely, so let's begin there with Listing 9-2.

```
def DepthFirst(graph, node, visited=None):
    if visited is None:
        visited = []
    visited.append(node)
```

```
    for neighbor in graph[node]:
        if neighbor not in visited:
            DepthFirst(graph, neighbor, visited=visited)
    return visited
```

Listing 9-2: Depth-first traversal of an undirected, unweighted graph

As with BFS, this version of DFS is simplified somewhat from that in the `graphs` module, primarily to ignore edge weights that don't alter the traversal order.

Note that `DepthFirst` is recursive, which makes sense since the intention is to dive as deeply as possible along a path. The code includes a loop over neighbors of a current node, as in BFS, but unlike BFS, DFS follows each neighbor all the way down before returning to consider the next neighbor. The recursive call to `DepthFirst` will return after it has applied the same logic to every neighbor of a current node, and their neighbors, relying on the Python call stack to manage the process. That this approach limits the complexity of the graph because of Python's finite recursion stack depth is merely an implementation detail.

Also note that `visited` is initialized if not explicitly supplied and then expanded to include the current `node` as the last element of the list, meaning each call to `DepthFirst` visits one, and only one, node. Passing `visited` as a keyword argument ensures that the same Python list is used on every recursive call because Python passes lists by reference, not by value (that is, not a copy).

Chapter 6 taught us that recursive algorithms must have a base case: some way to end the recursion. DFS's base case is that the `for` loop will exit because the current node will eventually run out of neighbors. When that happens, `visited` is returned, having been updated by the deep dives over the neighbors of the current `node`. When all nodes that are reachable by DFS, as dictated by the structure in the adjacency list, have been visited, the initial call to `DepthFirst` returns `visited`, which now contains every node in the order visited. The recursive calls to `DepthFirst` within the `for` loop discard return values.

Let's give `DepthFirst` a go using undirected graph `A`:

```
>>> A
[{1, 2}, {0, 3, 4}, {0, 5}, {1}, {1, 5}, {2, 4}]
>>> DepthFirst(A, 3)
[3, 1, 0, 2, 5, 4]
>>> BreadthFirst(A, 3)
[3, 1, 0, 4, 2, 5]
```

I included the output of `BreadthFirst` for comparison.

Both functions traversed the entire graph, which either will do if every graph node is accessible from the beginning node. What changes is the order in which the nodes are visited. The example graph is small, so the difference between BFS and DFS isn't dramatic but should be explainable from the algorithm. Figure 9-7 displays the path taken by DFS.

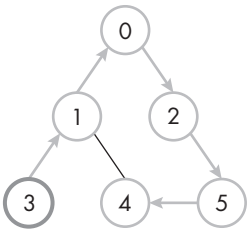

Figure 9-7: A depth-first traversal beginning with node 3

DFS goes deep, visiting a new node on every recursive call. It won't visit another neighbor of a node until it's gone as deep along the first neighbor as possible. Therefore, beginning with node 3, DFS dives down all the way to node 0. Then, it tries to visit node 1 from node 0, but node 1 is already in the visited list. DFS then moves to node 2 and dives along it, eventually reaching node 4 from node 5.

Node 4's neighbors are nodes 1 and 5, both already visited, so each recursive call to `DepthFirst` returns, backing up to node 1. The first neighbor of node 1 was node 0, which started the entire chain leading to node 4. Node 4 is the next neighbor of node 1, but it's already on the visited list, so the `DepthFirst` call with node 1 exits back to the initial call using node 3. Node 3 has no additional neighbors beyond node 1, so the initial call ends, and the complete visited list is returned in the order marked in Figure 9-7.

DFS is similarly happy to traverse directed graphs. Consider graph `E`:

```
>>> E
[{1}, {3}, {0, 5}, set(), {1, 5}, set()]
>>> [DepthFirst(E, i) for i in range(6)]
[[0, 1, 3], [1, 3], [2, 0, 1, 3, 5], [3], [4, 1, 3, 5], [5]]
>>> [BreadthFirst(E, i) for i in range(6)]
[[0, 1, 3], [1, 3], [2, 0, 5, 1, 3], [3], [4, 1, 5, 3], [5]]
```

As with graph `A`, I include the output of `BreadthFirst` for every node in `E`. Four of the traversals are identical between BFS and DFS, a consequence of the simplicity of graph `E`. However, the traversals for nodes 2 and 4 differ between BFS and DFS, but only in the order in which the nodes are visited. Therefore, Figure 9-6 remains relevant regarding the nodes visited by DFS, but the order is slightly different.

BFS, beginning on node 2, looks at node 0 and node 5, neighbors of node 2, then proceeds to node 1 and finally to node 3. DFS moves as far as possible along the path, beginning with node 0, before backing up to the second neighbor of node 2 to grab node 5. A similar difference in traversal leads to the ordering beginning at node 4.

The graphs we've explored so far are quite simple and don't illustrate the distinction between BFS and DFS as nicely as they could. To drive the point home, consider BFS and DFS applied to graph `S` as Figure 9-8 shows.

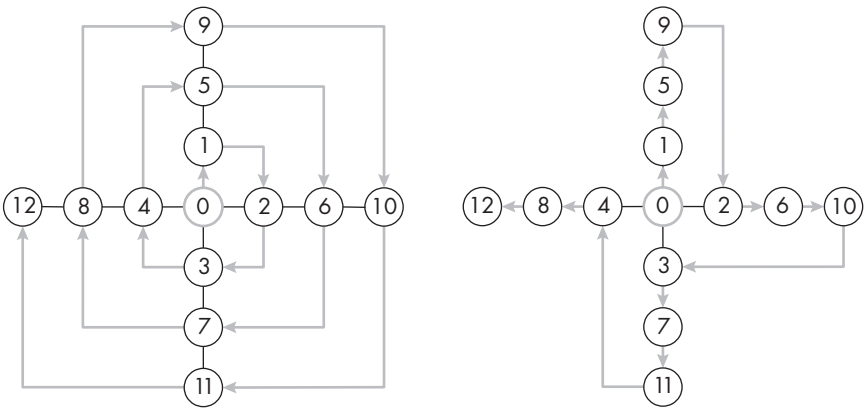

Figure 9-8: BFS (left) and DFS (right) using arrows to show the traversal path from node 0

Graph `S` is undirected and in a cross shape. The arrows in Figure 9-8 display the path followed by BFS (left) and DFS (right). The distinction between the two traversals is now plain to see. BFS first visits all nodes a distance 1 away from the beginning node, then all nodes a distance 2 away, and so on, until all nodes have been visited. This behavior isn't restricted to this example; it's fundamental to the way BFS operates.

DFS, as we expect, dives as far as possible from node 0 along the path of node 1, the first neighbor of node 0, until hitting node 9. It then backs up from node 9 to nodes 5 and 1, looking for unvisited neighbors along the way. There are none, so DFS backs up to node 0 and repeats with its next neighbor, node 2. The process continues until all nodes are visited.

Traversals as Searches

I've used the acronyms BFS and DFS to refer to breadth-first and depth-first traversals of a graph, where the *S* means *search*, though we've yet to do any searching. Let's now use the traversals to search a graph for a key—in this case, a string representing a name. If the name exists in the graph, and it can be reached from the starting node, then return the data associated with the name; otherwise, indicate that the name isn't found.

We've been using adjacency lists to represent graphs. We could alter the adjacency list structure to keep associated data with each node, but that's cumbersome and mixes node data with node relationships. Instead, we'll use an ancillary Python dictionary to store the node data. The dictionary key is the node number, and the data is a two-element list of the name followed by its associated data (here, a string identifying the type of being with that name). For example, you'll find the `people` dictionary in the `graphs` module:

```
people = {
    0: ['Drofo', 'halfing'],
    1: ['Aranorg', 'human'],
    2: ['Yowen', 'human'],
    3: ['Fangald', 'wizard'],
```

```
    4: ['Lelogas', 'elf'],
    5: ['Milgi', 'dwarf'],
}
```

There are six nodes in `people` corresponding to the six nodes in the adjacency lists for graphs `A` through `E`. In other words, graphs `A` through `E` encode relationships between the entities in the `people` dictionary.

The elegance of the BFS and DFS traversal algorithms must be slightly diminished to enable the search, but the violence is minimal. For BFS, we get Listing 9-3.

```
def BreadthFirstSearch(graph, start, name=None, data=None):
    visited, queue = [start], [start]
    while queue:
        node = queue.pop(0)
        if data[node][0] == name:
            return True, data[node][1]
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
    return False, None
```

Listing 9-3: Breadth-first search

DFS becomes Listing 9-4.

```
def DepthFirstSearch(graph, node, visited=None, name=None, data=None):
    if (visited is None):
        visited = []
    if data[node][0] == name:
        return True, data[node][1]
    visited.append(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            found, type = DepthFirstSearch(graph, neighbor, visited=visited,
                                           name=name, data=data)
            if (found):
                return found, type
    return False, None
```

Listing 9-4: Depth-first search

Both `BreadthFirstSearch` and `DepthFirstSearch` accept a `graph` and starting `node` along with the target `name` and the associated `data`.

BFS remains closest to the traversal-only algorithm, but after pulling `node` from the front of the queue, we check whether that node's name element is the one we're looking for. If so, we're done and return `True` along with the associated data. Should we ever exit the `while` loop, the graph has been traversed without locating the target, so `False` and `None` are returned.

DFS is recursive, so care is necessary. As with BFS, if the current `node`—now an argument to the function—holds the `name` we seek, we return `True` and the associated data. However, the caller isn't necessarily the initial caller. Therefore, the `for` loop, which calls `DepthFirstSearch`, receives the returned data in `found` and `type`. If `found`, immediately return to the previous caller.

Checking `found` ensures that the call stack will be traversed after the target name is located, ultimately leading to the initial caller. Should `name` never be discovered, `False` and `None` will be returned and passed up the call stack.

The search functions work with undirected and directed graphs. For convenience, and to eliminate further violence to the initially elegant implementations, weighted graphs of any kind are not supported, though you're invited to make the necessary adjustments.

Let's take these new functions out and see if they work. First, we'll use an undirected graph:

```
>>> BreadthFirstSearch(A, 1, name="Milgi", data=people)
(True, 'dwarf')
>>> BreadthFirstSearch(A, 4, name="Yowen", data=people)
(True, 'human')
>>> BreadthFirstSearch(A, 5, name="Nauros", data=people)
(False, None)
>>> DepthFirstSearch(A, 4, name="Yowen", data=people)
(True, 'human')
>>> DepthFirstSearch(A, 4, name="Fangald", data=people)
(True, 'wizard')
```

So far, so good.

Now consider a directed graph, that of Figure 9-6, which is in `E`. Notice that the traversals in Figure 9-6 are not all the same. A traversal beginning with node 2 will exclude node 4, while nodes 0 and 2 are missed by traversals beginning with node 4. This affects the search results like so:

```
>>> BreadthFirstSearch(E, 2, name="Drofo", data=people)
(True, 'halfling')
>>> BreadthFirstSearch(E, 2, name="Lelogas", data=people)
(False, None)
>>> DepthFirstSearch(E, 4, name="Drofo", data=people)
(False, None)
>>> DepthFirstSearch(E, 4, name="Lelogas", data=people)
(True, 'elf')
```

The calls to `BreadthFirstSearch` begin with node 2. The search for Drofo succeeds because node 0 is in the list of nodes accessible from node 2, but the search for Lelogas fails because node 4 is not. Beginning instead with node 4 flips the results so that Drofo is now missing, but Lelogas isn't.

It's often helpful to know the shortest path between two nodes in a graph. We're almost there with our basic algorithms; let's understand what it takes to get all the way.

The Shortest Path Between Nodes

Locating the shortest path between two points is a common problem that we often take for granted because of our ever-present phones. The mapping software on your phone implements a glorified version of the topic we'll explore in this section: finding the shortest path between two nodes, including scenarios where we can't get from node *A* to node *B*.

The previous section showed that BFS first examines all nodes a distance 1 away from the starting node, then all a distance 2 away, and so on. This insight will lead us from stock BFS to the shortest-path algorithm for unweighted graphs. Processing weighted graphs takes more effort; we'll get to them next.

Unweighted Shortest Path

For a shortest-path algorithm for unweighted graphs, we need BFS as well as a way to track the path so that when *B* is located, we have the sequence of nodes from *A* to *B*. Because no path might exist between *A* and *B*, our algorithm will account for that possibility. Finally, because we're using adjacency lists, the shortest-path algorithm is equally content with directed and undirected graphs.

Let's begin with the code in Listing 9-5.

```
def ShortestPath(graph, start, end):
    visited, queue = [start], [[start]]
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if (node == end):
            return path
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(path + [neighbor])
    return []
```

```
def BreadthFirst(graph, start):
    visited, queue = [start], [start]
    while queue:

        node = queue.pop(0)


        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
    return visited
```

Listing 9-5: BFS modified to locate the shortest path between two nodes

Listing 9-5 shows `ShortestPath` on the left and `BreadthFirst` from Listing 9-1 on the right. I paired like lines of code to accentuate differences.

BFS uses a `queue` to hold the next node to investigate. The `ShortestPath` function uses a queue as well, but in place of the node number, it holds a list, the first of which is `[start]`; that is, the list holding a single number, the starting node.

BFS wants to examine all the neighbors of the current `node` with the `for` loop. Because the queue holds lists (paths), `ShortestPath` extracts the last node of the path just pulled from the front of the queue (`path[-1]`) to use as the current node. If this `node` happens to be the endpoint, the path from `start` to `end` is in `path`, which is returned.

If `node` isn't the final destination, its neighbors are examined, and if not already visited, are *appended* to the current `path`, *then* pushed on the `queue`. This tracks the path from `start` to the neighbors of `node`. Should the `end` node never appear during the breadth-first traversal, the `while` loop will exit when the `queue` is empty, and `[]` is returned to indicate no path to `end` was found.

Carefully review Listing 9-5 to convince yourself that `ShortestPath` does what I claim. The fundamental takeaways include how BFS moves from the starting node and the trick of pushing the path to the current node on the queue instead of the node itself.

Let's continue with a few more examples. Figure 9-9 presents four graphs that match graphs `U`, `W`, `T`, and `V` in the `graphs` module.

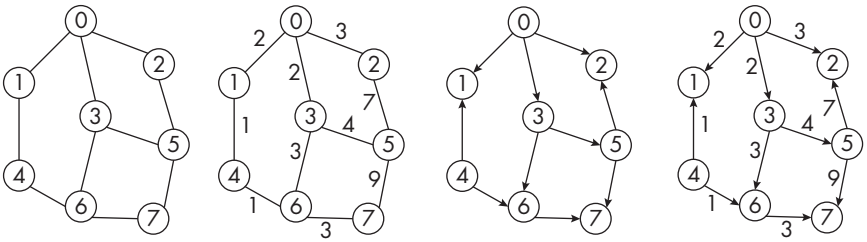

Figure 9-9: Graphs `U`, `W`, `T`, and `V`

The `ShortestPath` function expects unweighted graphs, so let's explore what it makes of `U` (undirected) and `T` (directed):

```
>>> ShortestPath(U, 0, 7)
[0, 2, 5, 7]
>>> ShortestPath(T, 0, 7)
[0, 3, 5, 7]
>>> ShortestPath(U, 7, 0)
[7, 5, 2, 0]
>>> ShortestPath(T, 7, 0)
[]
```

Multiple paths exist between node 0 and node 7. The `U` graph has three of equal length: `[0, 2, 5, 7]`, `[0, 3, 5, 7]`, and `[0, 3, 6, 7]`. The `ShortestPath` function stops on the first one found, which is `[0, 2, 5, 7]` because node 5 is a neighbor of node 2 and node 5's neighbor is node 7.

Switching to `T`, a directed graph, restricts the number of paths. Now we have only two paths from node 0 to node 7 of any length: `[0, 3, 5, 7]` and `[0, 3, 6, 7]`. The neighbors of node 5 are examined first, meaning the function will locate `[0, 3, 5, 7]` before the path involving node 6.

Undirected graphs are symmetric in that if an edge is present between node *A* and *B*, we can move from *A* to *B* and from *B* to *A*. Therefore, asking for the shortest path from node 7 to node 0 returns the reverse of the path from node 0 to node 7, `[7, 5, 2, 0]`.

Finally, in `T`, there's no way to move from node 7, as it is an absorbing node with no outbound arrows. Therefore, no path exists from node 7 to node 0, which explains the empty list returned by `ShortestPath`.

Dijkstra's Algorithm for Weighted Graphs

It's possible to alter the code in `ShortestPath` to ignore weights so that we might pass in graphs like those in `W` and `V` of Figure 9-9, but the path returned won't be satisfactory. This is because the very reason we place weights on the edges of a graph is to use the information they provide. So how do we find the shortest path that respects the edge weights?

The standard answer to that question is *Dijkstra's algorithm*, developed in 1956 by Dutch computer scientist Edsger Dijkstra (1930–2002). It's implemented in the `graphs` module as `Dijkstra`, and in 2024 it was proven to be universally optimal (that is, the best approach). Let's see the algorithm in action before working with it:

```
>>> Dijkstra(W, 0, 7)
([0, 1, 4, 6, 7], 7)
>>> Dijkstra(W, 7, 0)
([7, 6, 4, 1, 0], 7)
>>> Dijkstra(V, 0, 7)
([0, 3, 6, 7], 8)
>>> Dijkstra(V, 7, 0)
([], 0)
```

The function returns the path as a list, as well as the total distance along the path. The weights must be positive values, though floating-point is just fine.

The results differ from those returned by `ShortestPath` for graphs `U` and `T`. Graph `W` is a weighted version of `U`. Because of the weights, the "shortest" path is now the longer path from node 0 to node 7 by way of nodes 1, 4, and 6. The total weight of this path is 2 + 1 + 1 + 3 = 7. The next "lightest" path is `[0, 3, 6, 7]`, with a weight of 2 + 3 + 3 = 8. The "heaviest" path is `[0, 2, 5, 7]`, weighing in at 3 + 7 + 9 = 19.

For the directed weighted graph in `V`, which mirrors `T`, the least heavy (or least expensive) path is `[0, 3, 6, 7]` with a weight of 2 + 3 + 3 = 8. For `T`, the shortest path was `[0, 3, 5, 7]`, but with weights, that path totals 2 + 4 + 9 = 15, making the other path the winner.

The `Dijkstra` function produces the expected output. An overview of the algorithm followed by an analysis of the code will help in understanding how the function works.

An Overview

We want the shortest path between a `start` node and an `end` node, where *shortest* means the smallest edge sum. Interpreting the edge weights as distances is easiest, but the weights could represent anything we deem important, like time, relationship strength, or difficulty.

Dijkstra's algorithm tracks three sets of information as it runs: a set of unvisited nodes, a set of shortest distances from the start node to every other node, and, for each node, the neighbor that the shortest path from the starting node comes from. As we walk through the code, the purpose of this last collection will become less nebulous.

At its core, Dijkstra's algorithm is a modified version of BFS in that the neighbors of a current node are examined. The difference is that the next node to examine, the next node to label as the current node, is the unvisited node with the smallest shortest distance.

The algorithm consists of three sections that I'm naming *initialization*, *the loop*, and *denouement*.

Initialization configures the search by defining the set of unvisited nodes, the distances, and the shortest path. The first is a set in Python, and the others are lists. The distances list, one element for each node in the graph, is initialized to "infinity." As distances to that node from the start node are uncovered, they are updated appropriately. If a node is never visited, marking its distance as infinite is appropriate. For us, "infinity" is a googol, 10^{100}. The distance to the starting node is zero, so `distance[start]` is assigned `0`.

The shortest-path list, also one element per graph node in length, is initially all `None`, which is the Python way to represent null. As the loop runs, the elements of the shortest-path list are set to the current node when it becomes clear that the shortest path to that node passes through the current node. The initial current node is the starting node.

The loop runs until the end node has been made the current node or all nodes have been visited. While it runs, the loop examines the neighbors of the current node and, when a shorter distance—the distance to the current node as currently known plus the weight from the current node to the neighbor—is uncovered, the shortest path for the neighbor updates to indicate that the shortest path comes from the current node.

The denouement constructs the list of nodes representing the shortest path from start to end by walking backward through the shortest-path list, beginning with the end node, then reversing the resulting list. The path and the total distance (sum of the edge weights) are then returned. If no path is found, the empty list and zero are returned.

The Code

I invite you to reread the preceding paragraphs, nebulous as they may still be, then read the `Dijkstra` function code in Listing 9-6. When you're ready, we'll walk through it together.

```
def Dijkstra(graph, start, end):
    googol = 1E100
    n = len(graph)
    distances = [googol] * n
    distances[start] = 0
    shortest_path = [None] * n
    unvisited = {i for i in range(n)}
    current_node = start

    while True:
        for neighbor, weight in graph[current_node]:
            new_distance = distances[current_node] + weight
```

```
            if (new_distance < distances[neighbor]):
                distances[neighbor] = new_distance
                shortest_path[neighbor] = current_node

        unvisited.remove(current_node)
        if (not unvisited) or (current_node == end):
            break

        k = [i for i in unvisited]
        d = [distances[i] for i in unvisited]
        current_node = k[d.index(min(d))]

    path = []
    while end is not None:
        path.append(end)
        end = shortest_path[end]
    path.reverse()
    if distances[path[-1]] < googol:
        return path, distances[path[-1]]
    return [], 0
```

Listing 9-6: Dijkstra's algorithm in Python

Listing 9-6 reflects the three sections of the algorithm. The first code block is the initialization that defines `googol`, sets `n` to the number of nodes in the graph, and configures `distances`, `shortest_path`, and `unvisited`. Notice that `current_node` is `start` and that `distances[start]` is `0`.

The `while` loop performs the search. It runs until `break` is hit, which happens if the `unvisited` set is empty (`not unvisited`) or the desired `end` node has been reached.

The first code block of the `while` loop iterates over the neighbors of the current node. Note the similarity to BFS. For each neighbor, a new distance is calculated as the distance to the current node from `start` (`distances[current_node]`) plus the `weight` from the current node to the neighbor. If this new distance is less than the current distance from `start` to the neighbor (`distances[neighbor]`), we know that the shortest-path distance from `start` to `neighbor` is `new_distance`. Moreover, we know that the shortest path from `start` to `neighbor` passes through `current_node`, which is why `shortest_path[neighbor]` is updated.

The second `while` loop code block removes the current node from the set of unvisited nodes, then asks if all nodes have been visited (`not unvisited`) or if the current node is the desired `end` node. If either condition holds, `break` from the `while` loop.

The third `while` loop code block sets `current_node` to the unvisited node with the smallest distance from `start` to the current node.

The denouement code block builds the shortest-path node list (`path`) by beginning at the `end` and using the links in `shortest_path` to move to `start`. The loop updates `end` as it goes and stops when `end` is `None` because `shortest_path[start]` is never updated from its initial value of `None`. We want

the path from `start` to `end`, but the loop constructed it in reverse order, so we use `reverse` to flip the `path` list.

Finally, if the distance to `end` is something other than infinity, `end` was reached from `start`, so the `path` and distance along the path are returned. Otherwise, there is no path from `start` to `end`.

The Shortest Path to All Nodes

The `Dijkstra` function in the `graphs` module is slightly more complex than Listing 9-6. The additional complexity exists for good reason. If we ignore the end node and instead run the algorithm to visit all nodes, we're left with the information we need in order to know the minimum path from the starting node to every other node in the graph. Calling `Dijkstra` with only a starting node does this for us.

Visiting all nodes in the graph leaves `distances` holding the total distance along the shortest path from the `start` node to every other node in the graph (index `distances` by the desired node number). Further, the `shortest_path` list contains, for each node, the previous node in the path from `start` to that node. Therefore, building the shortest path from `start` to every other node is possible. I'll let you read the extra code yourself. Here's what it gives us:

```
>>> Dijkstra(W, 0)
([[0], [0, 1], [0, 2], [0, 3], [0, 1, 4], [0, 3, 5],
 [0, 1, 4, 6], [0, 1, 4, 6, 7]], [0, 2, 3, 2, 3, 6, 4, 7])
>>> Dijkstra(V, 0)
([[0], [0, 1], [0, 2], [0, 3], [4], [0, 3, 5], [0, 3, 6],
 [0, 3, 6, 7]], [0, 2, 3, 2, 1e+100, 6, 5, 8])
```

The `Dijkstra` function now returns a list of the shortest paths to every node from the starting node and the total distance along that path. For `V`, a directed graph, the start node, 0, isn't in the path to node 4 because node 4 is not accessible from node 0. The distance for node 4, then, is infinite (`1e+100`).

You now know how to find the shortest paths for any graph, weighted or unweighted, directed or undirected. Let's continue to explore a particularly useful type of directed graph and the concept of a topological sort on a graph.

Directed Acyclic Graphs and Topological Sort

A particularly common graph application involves *directed acyclic graphs (DAGs)*. A cycle exists in a directed graph if a path exists from node *A* back to node *A*. In a DAG, no such path exists; the arrows impose paths leading essentially in one direction only.

For example, consider the left-hand top and bottom portions of Figure 9-10.

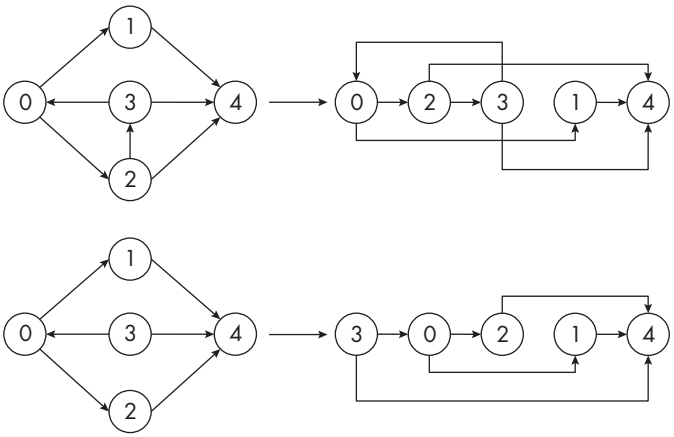

Figure 9-10: A directed graph with a cycle (top) and with the cycle removed (bottom)

The top graph contains a cycle providing a path from node 0 through node 2, then node 3, back to node 0. The bottom graph removes the edge between nodes 2 and 3 to turn the graph into a DAG.

A lack of cycles means that DAGs can be topologically sorted. A *topological sort* is a linear ordering of the nodes in the graph such that the arrows all lead in one direction. If the nodes represent tasks that must be completed before other tasks, the topological sort generates a sequence by which tasks can be completed so that prior tasks are always done before tasks that depend on their outputs. All DAGs possess at least one valid topological sort and often more than one.

Look again at Figure 9-10, paying attention to the graphs on the right-hand side. They show, or attempt to show, a topological sort of the graph on the left.

Let's start with the graph on the lower right, which shows a topological sort of the cycle-free (acyclic) graph on the lower left. You can see that all arrows point from left to right. If we interpret the arrows as indicators of task dependencies, we see that the topological sort tells us to do task 3 first, then task 0, followed by tasks 2, 1, and finally, 4. Performing the tasks in this order ensures that inputs required by later tasks are available when needed.

Now look at the "sort" on the upper right. I use quotation marks because the graph contains a cycle indicated by an arrow pointing from right to left, the one from node 3 back to node 0. Such a cycle poses a problem if we want to schedule tasks. Task 3 depends on task 2, task 2 depends on task 0, but task 0 depends on task 3, and now we're stuck. A topological sort can reveal cycles in a graph, which in turn may represent failures in structuring the process that produced the directed graph in the first place.

Working Through an Example

We all have the same problem every morning: getting dressed. Figure 9-11 displays a DAG representing the task dependencies of getting dressed.

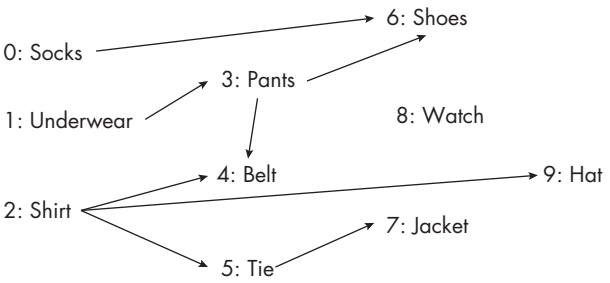

Figure 9-11: Getting dressed as a directed acyclic graph

The arrows show dependencies. For example, we must have pants and socks on before putting on shoes. Therefore, nodes 0 and 3 must happen before node 6. Likewise, we must (usually) have a shirt on before putting on a hat. Some tasks have no prerequisites, like node 8, putting on a watch. We can put on a watch at any time during the process.

Figure 9-11 is a DAG, meaning there are no cycles and at least one topological sort exists. Figure 9-12 shows one such sort.

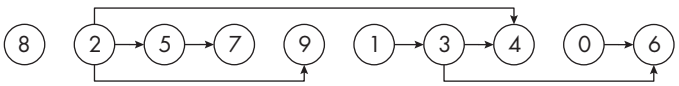

Figure 9-12: Getting dressed as a topologically sorted graph

I'll detail the sorting algorithm momentarily, but for now, consider the sort in the figure. All arrows point in the same direction because the graph has no cycles. For example, nodes 0 and 3 both occur before node 6, representing the requirement that socks and pants be on before putting on shoes.

Node 2, putting on a shirt, is a prerequisite of nodes 4 (belt), 5 (tie), and 9 (hat). The sort enforces these requirements as shown.

The sort puts node 8, the watch, first. Because it has no dependencies and nothing depends on it, we can put node 8 anywhere in the sequence. Therefore, getting dressed has many possible topological sorts.

Let's learn how to produce a topological sort for a given DAG.

Coding Topological Sort

Finding a topological sort of an arbitrary DAG might initially sound tricky. Thankfully, however, we already know what we need because, in the end, topological sorting is nothing more than a slight twist on DFS.

Listing 9-7 presents topological sorting in Python.

```
def TopologicalSort(graph):
    def DFS(graph, node, visited, result):
        visited.add(node)
        for neighbor in graph[node]:
            if (neighbor not in visited):
                DFS(graph, neighbor, visited, result)
        result.insert(0, node)
```

```
    visited = set()
    result = []
    for node in range(len(graph)):
        if (node not in visited):
            DFS(graph, node, visited, result)
    return result
```

Listing 9-7: Topological sorting of unweighted DAGs

The `TopologicalSort` function expects a DAG represented as an adjacency list of the form we've used consistently throughout the chapter. The caption reads "unweighted DAGs." As with the code for DFS earlier, Listing 9-7 is slightly simpler than the `TopologicalSort` code in the `graphs` module. The latter includes a `weighted` keyword telling the function to expect and then ignore weights on the graph's edges.

I embedded a custom `DFS` function to implement the twist: the `result .insert` line. The `DFS` function is recursive but passes a reference to a `visited` set and a `result` list. These are updated on each recursive call to mark the current node as visited and to insert the current node at the head of the `result` list. Why the head of the list and not the end will become clear momentarily.

The main part of `TopologicalSort` initializes `visited` and `result`, then iterates over all the nodes in the graph, calling `DFS` on every unvisited node. Throughout this process, the single instances of `visited` and `result` are updated by `DFS`. When all nodes have been visited, `result` is returned as the topological sort.

The `TopologicalSort` function works like so:

```
>>> TopologicalSort(cycle)
[0, 2, 3, 1, 4]
>>> TopologicalSort(nocycle)
[3, 0, 2, 1, 4]
>>> TopologicalSort(dress)
[8, 2, 5, 7, 9, 1, 3, 4, 0, 6]
```

Here, `cycle` and `nocycle` are the graphs in Figure 9-10, and `dress` is the graph in Figure 9-11:

```
[{6}, {3}, {9, 4, 5}, {4, 6}, set(), {7}, set(), set(), set(), set()]
```

As an exercise, convince yourself that Figure 9-11 is represented by `dress`. Note also that `TopologicalSort`, as implemented, always returns something, even if the resulting "sort" isn't valid because the graph isn't a DAG (for example, `cycle`).

The code in Listing 9-7 recursively performs DFS on every node. Earlier in the chapter, you learned that this translates into diving as deeply as possible through the graph from a current node before being forced to backtrack. The algorithm visits every node that occurs after the current node before backtracking to the current node. This is why the current node is placed at the head of the result list: all nodes coming later have already been placed at

the head of the list, meaning the current node comes before all of them. Arrows imply dependency, so this arrangement of DFS and placing the current node at the head of the result list ensures that the current node happens before any node that might depend on it.

Summary

This chapter introduced graph theory, a rich discipline with a long history that's taken on greater significance with the advent of computer science. We focused on basic concepts and then shifted into a more pragmatic approach to present core algorithms often encountered by practicing software engineers.

Specifically, we discussed two options for representing graphs: adjacency lists and adjacency matrices. I then introduced breadth-first and depth-first traversals, the two foundational graph theory algorithms. We implemented both in Python and then saw how to use graph traversals as searches. Breadth-first and depth-first traversals as searches are so common that most people refer to either use as BFS or DFS, respectively.

We examined finding paths through a graph from a starting node (vertex) to an ending node. Modern reliance on computers for navigation has only made solving such problems all the more necessary, even critical. We discussed two algorithms for finding the shortest path between nodes. The first works for unweighted graphs, directed or undirected. The second, Dijkstra's algorithm, applies to weighted graphs with positive weights.

We concluded the chapter by discussing directed acyclic graphs, or DAGs. DAGs are especially prevalent because they typically represent neural networks, the backbone of modern AI. DAGs are amenable to topological sorting, which sequences the nodes in order so that if the nodes represent tasks, prerequisite tasks are completed before tasks that depend on their outputs. You learned that topological sorting is little more than a tweak on depth-first graph traversal.

One type of graph is so commonly encountered by programmers that it deserves a separate chapter: trees. Let's press on and explore the forest, or at least a few of its trees.