CHAPTER 7

Graphs: Only Connect!

In this chapter, you will learn:

- The *stack* abstract data type.
- The *indexed min priority queue* data type, which is the final data type included in this book.
- How to model a *graph* using nodes and edges. In a directed graph, the edges have an orientation. In a weighted graph, edges have an associated numeric value.
- How Depth First Search uses a stack to organize searching in a graph.
- How Breadth First Search uses a queue to search through a graph. If a path exists between a source node and a target node, Breadth First Search will return the shortest path that exists.
- How to detect whether a directed graph contains a *cycle*: a sequence of edges starting from, and ending with, a specific node.
- How to use Topological Sort in a directed graph to produce a linear ordering of nodes compatible with all dependencies in the directed graph.
- How to determine the shortest accumulated path in a weighted graph from one node to all other nodes.
- How to determine the shortest accumulated path in a weighted graph between any two nodes.

Graphs Efficiently Store Useful Information

I've covered algorithms for solving common problems in information systems regarding storing and processing data. These algorithms can solve countless real-world problems if only we could properly model these problems. Here are three such problems that I will solve by using *graphs*:

- A maze consists of rooms with doorways leading to other rooms. Find the shortest path from an entrance to the exit.
- A project is defined by a collection of tasks, but some tasks require other tasks to complete before they can begin. Assemble a linear schedule that describes the order in which the tasks can be performed to complete the project.
- A map contains a collection of highway segments, including their length in miles. Find the shortest traveling distance between any two locations in the map.

Each of these problems can be modeled effectively using *graphs*, a fundamental concept studied by mathematicians for centuries. Modeling the relationships *between* data is often as important as the data values themselves. A *graph* models information as *nodes* connected by *edges*. Any number of edges, e = (u, v), can exist to represent some relationship between nodes u and v. As you can see in Figure 7-1, graphs can model concepts from a variety of application domains. An *undirected graph* can model the structural relationship between the carbon and hydrogen atoms in the propane molecule. A mobile app can provide driving directions in New York City by representing the orientation of one-way streets as a *directed graph*. A driver's road atlas can represent the driving distances between New England state capitals as a *weighted graph*. With a bit of computation, you can see that the shortest driving distance from Hartford, Connecticut, to Bangor, Maine, is 278 miles.

A *graph* is a data type that contains a collection of N distinct nodes, each with a unique label to identify the node.[1] You can add an *edge* to a graph to connect two different nodes, u and v, with each other. An edge is represented as (u, v), and u and v are called its *endpoints*. Each edge (u, v) joins u and v together so u is adjacent to v (and, vice versa, v is adjacent to u).

1 A node is often called a vertex, but for this chapter I use the term *node* to be consistent with `networkx`.

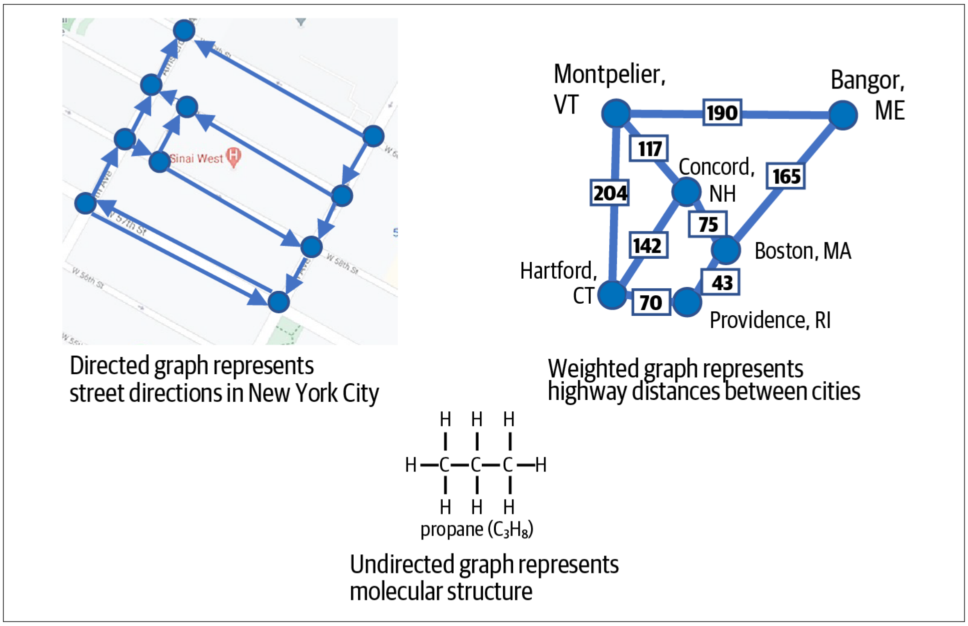

Directed graph represents street directions in New York City
Montpelier, VT
Bangor, ME
Concord, NH
Boston, MA
Hartford, CT
Providence, RI
190
117
204
165
75
142
43
70
Weighted graph represents highway distances between cities
propane (C_3H_8)
Undirected graph represents molecular structure

Figure 7-1. Modeling different problems using graphs

The graph in Figure 7-2 has 12 unique nodes and 12 edges. Imagine that each node is an island, and the edges are bridges connecting the islands. A traveler can walk from island B2 to island C2 or from island C2 to island B2; however, there is no way for a traveler to *walk directly* from island B2 to B3. Instead, the traveler can cross the bridge from island B2 to island C2, then cross the bridge from island C2 to island C3, and finally cross the bridge from island C3 to B3. Based on this representation of the islands and its bridges, the traveler can find a sequence of bridges to travel between any of the "B" and "C" islands, but despite the bridges connecting the "A" islands, there is no way to travel from an "A" island to a "B" island.

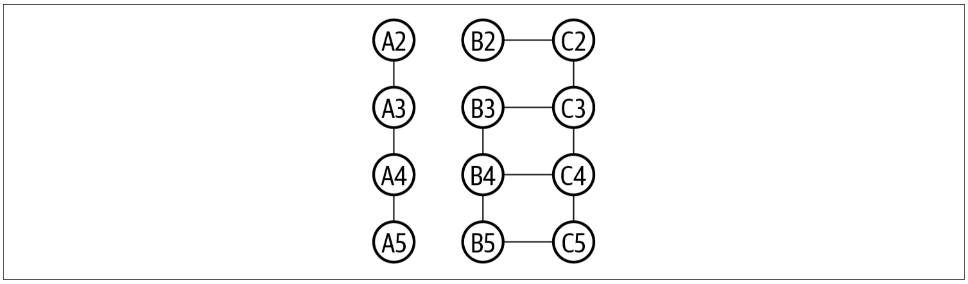


Figure 7-2. An undirected graph with 12 nodes and 12 edges

Given a graph containing nodes and edges, a common problem is to compute a path from a *source* node (such as the node representing island B2) to a *target* node (such as the node representing the island B3) using only the edges contained in the graph.

A *path* is formed by a sequence of edges starting at a source and terminating at a target. Naturally, every edge forms a path between its two endpoint nodes, but what about two nodes that do not have an edge to join them together? In Figure 7-2 there is a path from node B2 to B3 by following the sequence of edges (B2, C2), (C2, C3), and then (C3, B3).

In a path, each successive edge must start at the node in which the prior edge terminated. A path can also be represented by listing the sequence of nodes encountered along the way, such as [B2, C2, C3, B3]. Another longer path from B2 to B3 is [B2, C2, C3, C4, B4, B3]. A *cycle* is a path that starts and ends at the same node, such as [C4, C5, B5, B4, C4]. A node, *v*, is *reachable* from another node, *u*, if there is a path of edges in the graph from *u* to *v*. In some graphs, there may be no path between two nodes—for example, there is no path from A2 to B2 in the graph from Figure 7-2. When this happens, the graph is considered to be *disconnected*. In a *connected* graph, it is possible to compute a path between any two of its nodes.

In this chapter, I will present three types of graphs:

Undirected graph
: A graph that contains edges (*u*, *v*) that connect two nodes such that *u* is adjacent to *v*, and *v* is adjacent to *u*. This is like a bridge that can be traveled in either direction.

Directed graph
: A graph that contains edges (*u*, *v*), where each edge has a fixed orientation. When the edge (*u*, *v*) is in the graph, *v* is adjacent to *u*, but the opposite is not true. This is like a one-way bridge that a traveler can use to walk from island *u* to island *v* (but not the other direction).

Weighted graph
: A graph that contains edges (*u*, *v*, *weight*), where *weight* is a numeric value associated with the edge (note: the underlying graph can either be directed or undirected). This weight represents an aspect about the relationship between *u* and *v*; for example, weight could represent the physical distance in miles between the locations modeled by the nodes *u* and *v*.

All graphs in this chapter are *simple* graphs, which means each edge is unique (i.e., there cannot be multiple edges between the same pair of nodes), and there are no self-loops where an edge connects a node with itself. Either a graph contains all undirected edges or it has all directed edges. Similarly, either a graph has all weighted edges or none of the edges have associated weights.

For the algorithms in this chapter, you need a graph data type that can provide the following functionalities:

- Return the number of nodes, N, and the number of edges, E, in the graph.
- Generate the collection of nodes and edges.
- Generate the adjacent nodes or edges for a given node.
- Add a node or edge to a graph.
- Remove a node or edge from a graph—this functionality is not essential for the algorithms presented in this chapter, but I include it for completeness.

Python has no built-in data structure that provides this functionality. Instead of implementing code from scratch, you need to install the NetworkX open source library to create and manipulate graphs. Doing so ensures you do not waste time reinventing the wheel, plus it gives you access to an impressive number of graph algorithms already implemented by `networkx`. In addition, `networkx` seamlessly integrates with other Python libraries to visualize graphs. The program in Listing 7-1 constructs the graph shown in Figure 7-2.

Listing 7-1. A program that builds the graph in Figure 7-2

```
import networkx as nx
G = nx.Graph()                                                ❶
G.add_node('A2')                                              ❷
G.add_nodes_from(['A3', 'A4', 'A5'])                          ❸

G.add_edge('A2', 'A3')                                        ❹
G.add_edges_from([('A3', 'A4'), ('A4', 'A5')])                ❺

for i in range(2, 6):
  G.add_edge('B{}'.format(i), 'C{}'.format(i))                ❻
  if 2 < i < 5:
    G.add_edge('B{}'.format(i), 'B{}'.format(i+1))
  if i < 5:
    G.add_edge('C{}'.format(i), 'C{}'.format(i+1))

>>> print(G.number_of_nodes(), 'nodes.')                      ❼
>>> print(G.number_of_edges(), 'edges.')
>>> print('adjacent nodes to C3:', list(G['C3']))             ❽
>>> print('edges adjacent to C3:', list(G.edges('C3')))       ❾
12 nodes.
12 edges.
adjacent nodes to C3: ['C2', 'B3', 'C4']
edges adjacent to C3: [('C3', 'C2'), ('C3', 'B3'), ('C3', 'C4')]
```

❶ `nx.Graph()` constructs a new undirected graph.

❷ A node can be any hashable Python object except `None`. Strings are a good choice.

❸ Add multiple nodes from a list using `add_nodes_from()`.

❹ Add an edge between two nodes, *u* and *v*, with `add_edge(u, v)`.

❺ Add multiple edges from a list using `add_edges_from()`.

❻ If an edge is added to a graph *before its nodes are*, the corresponding nodes are automatically added to the graph.

❼ A graph can report its number of nodes and edges.

❽ Find the nodes adjacent to `v` by using the `G[v]` lookup capability.

❾ Find the edges adjacent to `v` by using the `G.edges(v)` function.

You might wonder about the order in which adjacent nodes (or edges) are returned when requested. In the subsequent code, when adjacent edges or nodes are requested, you cannot expect that they will be returned in a specific order.

Using Depth First Search to Solve a Maze

Given a rectangular maze as shown in Figure 7-3, how would you write a program that solves it? The entrance to this 3 x 5 maze consisting of 15 cells is at the top, and the desired exit is at the bottom. To move through the maze, you can only move horizontally or vertically between rooms that are not blocked by walls. The first step is to model the maze using an undirected graph consisting of 15 nodes, where each node, labeled (`row`, `column`), models a cell in the maze. The *source* of the maze, for example, is labeled `(0, 2)`, and the *target* is labeled `(2, 2)`. The second step is to add an edge between two nodes (*u*, *v*) if their corresponding cells in the maze *do not have a wall between them*. The resulting graph is shown overlayed with the maze so you can see the one-to-one correspondence between a cell in the maze and a node in the graph.

Finding a path between the source node `(0, 2)` and target node `(2, 2)` is equivalent to finding a solution to the original rectangular maze. I will show you a technique for solving any such maze, regardless of its size. If you try to solve this maze on your own, you will explore different paths, discarding those that lead to "dead ends," until you eventually find a solution. Though you might not realize it, you have a significant advantage because *you can see the whole maze at a glance* and can make decisions on which paths to explore based on your own sense of *how close you are to the final* *target*. Imagine, instead, that you are stuck inside the maze,[2] and you can only see the cells that directly connect to the cell in which you stand—these restrictions completely change your approach.

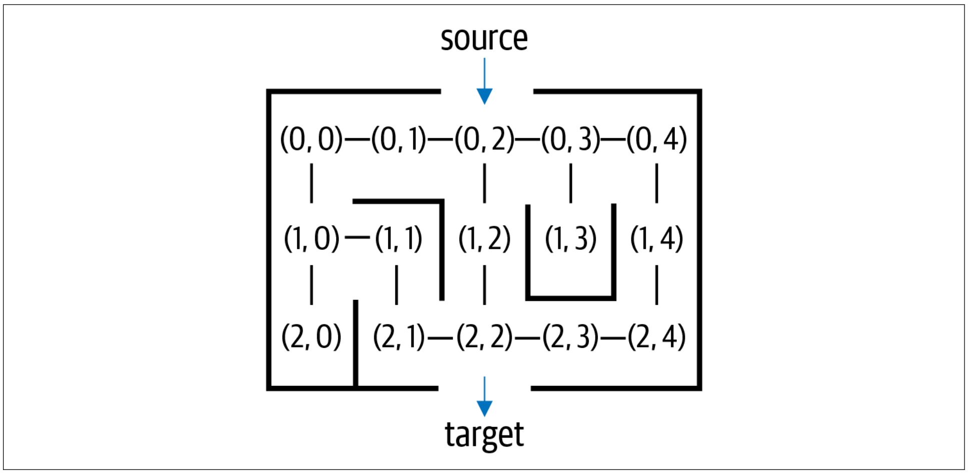


Figure 7-3. A graph modeling a rectangular maze

Let's develop a strategy to solve a maze using its corresponding undirected graph, as seen in Figure 7-4. These mazes are randomly generated, which is an interesting exercise all on its own.[3]

Starting from the source `(0, 2)`, you see three adjacent nodes that are connected by edges; you arbitrarily head east to `(0, 3)` but remember `(0, 1)` and `(1, 2)` as potential points to explore. Node `(0, 3)` has three adjacent nodes, but *you remember that you came from* `(0, 2)`, and you don't repeat where you've already been, so you arbitrarily head south to `(1, 3)` but remember `(0, 4)` as a potential point to explore. You have just followed the highlighted path shown in Figure 7-4 and have reached a dead end.

You know `(1, 3)` is a dead end because *there is no adjacent node you have not already seen*. What should you do? In Figure 7-4, I circled those nodes you came across but didn't explore: perhaps your search will be more fruitful if *you backtrack to one of these previous nodes and continue searching from there*.

2 Which can happen in real-life corn mazes! *Cool Patch Pumpkins* in Dixon, California, is the largest corn maze in the world, measuring a total 63 acres. It takes several hours to complete.

3 See the `ch07.maze` program for details.

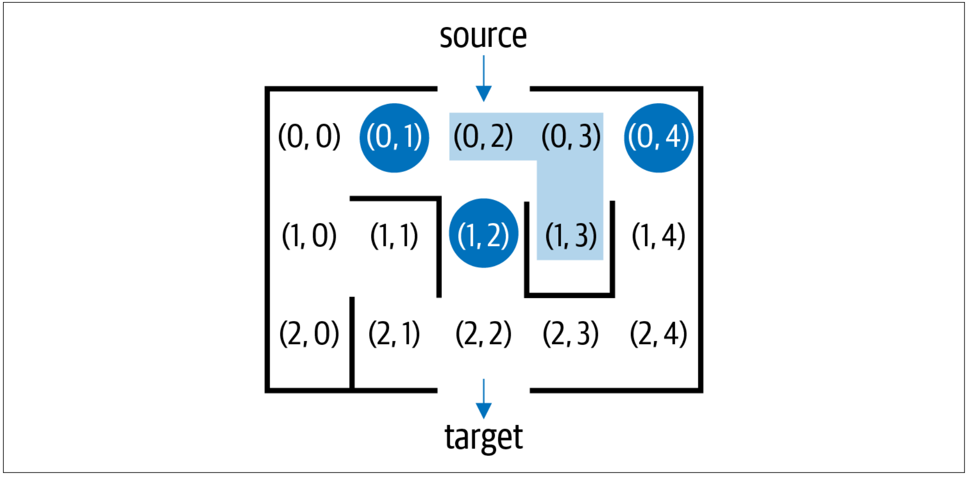


Figure 7-4. Hitting a dead end while exploring a maze

Here is a rough sketch of the activity of a graph-searching algorithm that explores a graph starting from a designated *source* node:

- Mark each node that you visit.
- Find the adjacent nodes to your current node that have not yet been marked as visited, and *arbitrarily select one of these* to explore.
- Go back to the last unmarked node you remembered when you hit a dead end.
- Continue exploring until all reachable nodes are marked.

Once the search algorithm completes, it should be possible to reconstruct the path from the source node to any node in the graph. To make this possible, the search algorithm must return a structure that contains enough information to support this capability. A common solution is to return a `node_from[]` structure, where `node_from[v]` either is `None`, if `v` is not reachable from the source node, or it is `u`, that is, *the prior node discovered before exploring* `v`. In Figure 7-4, `node_from[(1, 3)]` is the node `(0, 3)`.

This sketch of an algorithm cannot be performed within a simple `while` loop, like when searching a linked list for a value. Instead, I now show how to use the *stack* abstract data type to maintain the search state while exploring the graph.

If you have ever eaten a meal at a cafeteria, you've no doubt grabbed the top tray from a stack of trays. The stack data type represents the behavior of such a stack of trays. A stack has a `push(value)` operation that adds `value` to become the newest value at the *top* of the stack, and `pop()` removes the value at the top of the stack. Another way to describe this experience is "Last in, first out" (LIFO), which is shorthand for "Last

[one] in [the stack is the] first [one taken] out [of the stack]." If you push three values, 1, 2, and 3, onto a stack, they will pop off the stack in order 3, 2, and finally 1.

Using the `Node` linked list data structure from Listing 6-1, the `Stack` implementation in Listing 7-2 has a `push()` operation to prepend a value to the front of a linked list. The `pop()` method removes and returns the first value in a linked list; as you can see, this provides the behavior for a stack. In `Stack`, the `push()` and `pop()` operations perform in constant time, independent of the total number of values in the stack.

Listing 7-2. Linked list implementation of `Stack` data type

```
class Stack:
  def __init__(self):
    self.top = None                          ❶

  def is_empty(self):
    return self.top is None                  ❷

  def push(self, val):
    self.top = Node(val, self.top)           ❸

  def pop(self):
    if self.is_empty():                      ❹
      raise RuntimeError('Stack is empty')

    val = self.top.value                     ❺
    self.top = self.top.next                 ❻
    return val
```

❶ Initially, `top` is `None`, reflecting an empty `Stack`.

❷ A `Stack` is empty if `top` is `None`.

❸ Ensures new `Node` is the first one in linked list, with existing linked list becoming the rest.

❹ An empty `Stack` causes a `RuntimeError`.

❺ Extract the newest value from `top` of stack to be returned.

❻ Reset `Stack` so the next `Node` is now on top (if `None`, then `Stack` becomes empty).

The Depth First Search algorithm uses a stack to keep track of marked nodes it will explore in the future. Listing 7-3 contains a stack-based implementation for Depth First Search. The search strategy is called *depth first* because it constantly tries to advance forward, always expecting that the solution is just one step away.

It starts at a source node, `src`, which is marked as having been visited (i.e., setting `marked[src]` to `True`) and is then pushed onto the stack for further processing. Each time through the `while` loop, the stack contains nodes that have already been visited and marked: they are popped from the stack, one at a time, and unmarked adjacent neighbors are marked and added to the stack for further processing.

Listing 7-3. Depth First Search of graph from designated source node, `src`

```
def dfs_search(G, src):           ❶
  marked = {}                     ❷
  node_from = {}                  ❸

  stack = Stack()
  marked[src] = True              ❹
  stack.push(src)

  while not stack.is_empty():     ❺
    v = stack.pop()
    for w in G[v]:
      if not w in marked:
        node_from[w] = v          ❻
        marked[w] = True          ❼
        stack.push(w)

  return node_from                ❽
```

❶ Conduct a Depth First Search over graph, `G`, starting from source node, `src`.

❷ The `marked` dictionary records nodes that have already been visited.

❸ Record how search got to each node: `node_from[w]` is the prior node working backward to `src`.

❹ Mark and place `src` node into `Stack` to start the search. The top node in the `Stack` represents the next node to explore.

❺ If the Depth First Search has not yet completed, `v` is the next node to explore.

❻ For each unmarked node, `w`, adjacent to `v`, remember that to get to `w`, the search came from `v`.

❼ Push `w` onto the top of the stack and mark it so it won't be visited again.

❽ Return the *structure of the search* that records for each node, `v`, the prior node from a search initiated at `src`.

Figure 7-5 visualizes the execution of Depth First Search, showing the updated state of the stack each time through the `while` loop. The highlighted node at the top of the stack is the current cell being explored; other nodes in the stack represent nodes that will *eventually be processed in the future*. You might wonder how Depth First Search avoids getting stuck wandering around aimlessly, forever. Each time a node is pushed onto the stack, it is marked, which means it will never be pushed onto the stack again. The `for` loop over `w` will find all unmarked nodes adjacent to `v`, yet to explore: it will mark each `w` and push `w` onto the stack for further exploration.

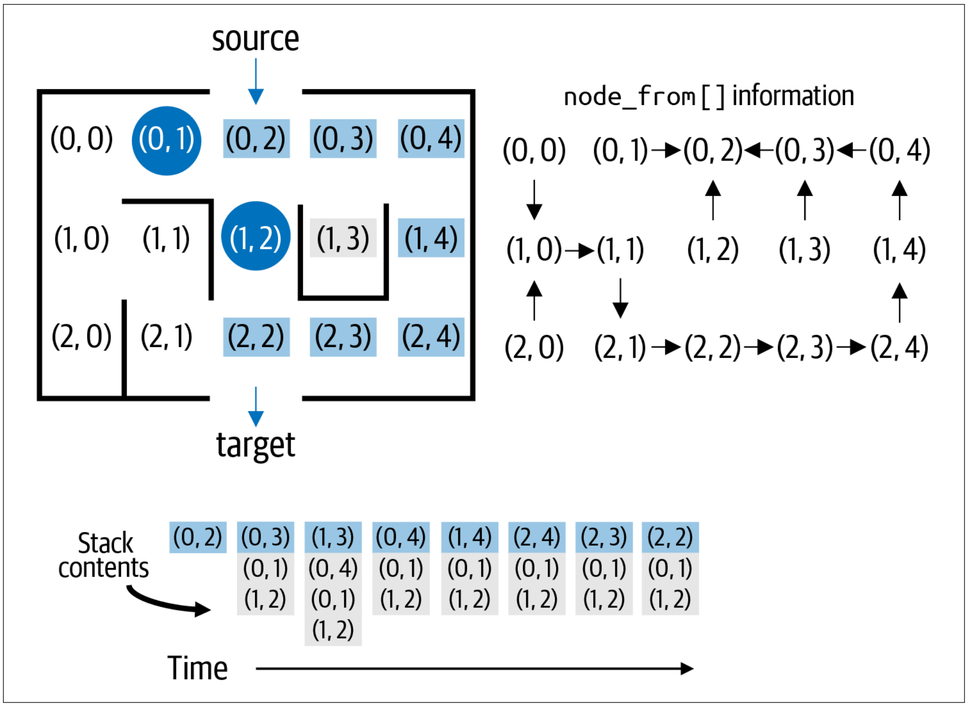


Figure 7-5. Depth First Search locates target if reachable from source

The search finds its first dead end, at `(1, 3)`, but quickly recovers, popping off node `(0, 4)` to resume its search. The search is paused in Figure 7-5 to show the state once the target is found, but the search continues until all nodes in the graph reachable from `src` are explored and the stack becomes empty.

The stack will ultimately become empty, since (a) there are a finite number of nodes in the graph, and (b) unmarked nodes are first marked before being pushed onto the `stack`. Since a node can never become "unmarked," eventually each node that is reachable from `src` will be pushed exactly once onto `stack` and subsequently removed within the `while` loop.

The resulting depth-first *search tree* is shown on the right in Figure 7-5 as contained in the `node_from[]` structure. This structure can be called a *tree* because there are no cycles among the arrows. It encodes information that can be used to recover the path from `(0, 2)` to any reachable node in the graph by working *backward*. For example, `node_from[(0, 0)] = (1, 0)`, which means the second-to-last node on the path from `(0, 2)` to `(0, 0)` was `(1, 0)`.

The computed six-move solution is not the shortest possible path to the target. Depth First Search offers no guarantee regarding the length of the discovered path, but it will eventually find a path to every reachable node from a designated source node. Given the computed `node_from[]` structure resulting from a Depth First Search initiated at `src`, the `path_to()` function in Listing 7-4 computes the sequence of nodes from `src` to any `target` reachable from `src`. Each `node_from[v]` records the prior node encountered in a search from `src`.

Listing 7-4. Recovering actual path from `node_from[]`

```
def path_to(node_from, src, target):          ❶
  if not target in node_from:
    raise ValueError('Unreachable')           ❼

  path = []
  v = target                                  ❷
  while v != src:
    path.append(v)                            ❸
    v = node_from[v]                          ❹

  path.append(src)                            ❺
  path.reverse()                              ❻
  return path
```

❶ `node_from` structure is needed to recover path from `src` to any `target`.

❷ To recover the path, set `v` to target node.

❸ As long as `v` is not `src`, append `v` to `path`, a backward list of nodes found on path from `src` to `target`.

❹ Continue backward by setting `v` to the prior node recorded by `node_from[v]`.

❺ Once `src` is encountered, the `while` loop terminates, so `src` must be appended to complete the backward `path`.

❻ Return the reverse of `path` to produce the proper ordering from `src` to `target`.

❼ If `node_from[]` doesn't contain `target`, then it is not reachable from `src`.

The `path_to()` function computes the sequence of nodes in reverse order from `target` backward until `src` is encountered; it then simply reverses the order of the discovered nodes to produce a solution in the proper order. If you try to recover a path to an unreachable node, then `path_to()` will raise a `ValueError`.

Depth First Search repeatedly heads off in arbitrary directions with the expectation that it is just one node away from the destination. Now let's look at a more methodical search strategy.

Breadth First Search Offers Different Searching Strategy

Breadth First Search explores nodes *in order of their distance from the source*. Using the same maze from Figure 7-3, Figure 7-6 identifies each cell in the graph by *its shortest distance* from the source. As you can see, it finds a path through the maze just three cells long. In fact, Breadth First Search will always find the shortest path in a graph in terms of the number of edges visited.

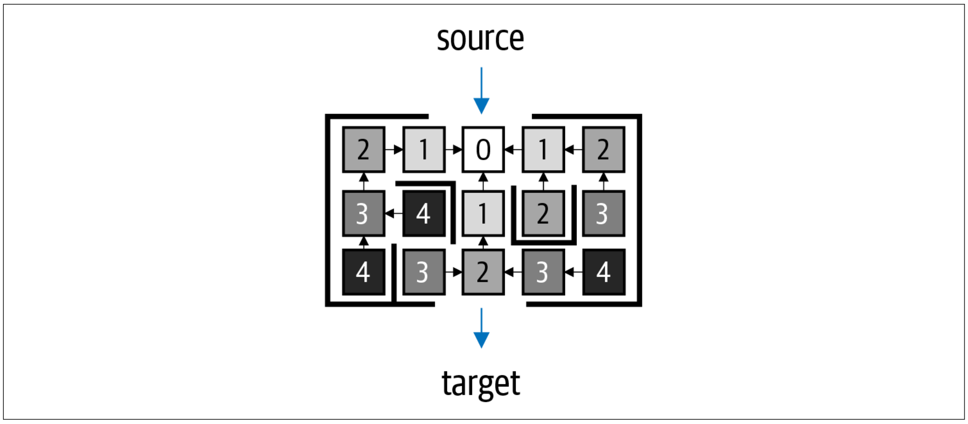


Figure 7-6. Breadth First Search locates the shortest path to target, if reachable from source

To provide some intuition behind Breadth First Search, observe in Figure 7-6 that from the source of the maze, there are three nodes that are just one step away—any one of these could lead to the shortest distance solution to the target, but, of course, you can't know which one. Instead of picking just one of these nodes to explore, Breadth First Search takes each one and advances one step further until it finds those nodes that are two steps away. With this methodical approach, it will not make any rash decisions while exploring the graph.

Instead of being optimistic like Depth First Search, Breadth First Search explores each of these nodes *in order* until all nodes that are one step away from the source have been visited; this results in four different nodes (labeled with 2 in Figure 7-6) that are two steps away from the source. In similar fashion, each of these nodes will be explored *in order* until all nodes that are two steps away from the source have been visited, resulting in four nodes that are three steps away from the source. This process continues until every node in the graph reachable from the source is visited.

Breadth First Search needs a structure to keep track of the nodes because it has to make sure that a node with distance `d` + 1 is not explored before all nodes with distance `d` are visited. The queue data type covered in Chapter 4 will process nodes in this order, because it enforces a "first in, first out" (FIFO) policy for adding and removing values. The code in Listing 7-5 is nearly identical to the code for Depth First Search, with the exception that a queue stores the *active search space*, that is, the nodes actively being explored.

Listing 7-5. Breadth First Search of graph from designated source node

```
def bfs_search(G, src):                 ❶
  marked = {}                           ❷
  node_from = {}                        ❸

  q = Queue()
  marked[src] = True                    ❹
  q.enqueue(src)

  while not q.is_empty():               ❺
    v = q.dequeue()
    for w in G[v]:
      if not w in marked:
        node_from[w] = v                ❻
        marked[w] = True                ❼
        q.enqueue(w)

  return node_from                      ❽
```

❶ Conduct a Breadth First Search over graph, `G`, starting from source node, `src`.

❷ The `marked` dictionary records nodes that have already been visited.

❸ Record how search got to each node: `node_from[w]` is the prior node working backward to `src`.

❹ Mark and place `src` node into `Queue` to start the search. The first node in the `Queue` represents the next node to explore.

❺ If the Breadth First Search has not yet completed, `v` is the next node to explore.

❻ For each unmarked node, `w`, adjacent to `v`, remember that to get to `w`, the search came from `v`.

❼ Place `w` as last node to explore at the end of the queue, and mark it so it isn't visited multiple times.

❽ Return the *structure of the search* that records for each node, `v`, the prior node from a search initiated at `src`.

Because Breadth First Search explores nodes in increasing order of their distance from the source, the resulting path to any reachable node in the graph will be a shortest path.[4] You can use the same `path_to()` function to recover the path from `src` to any node in the graph reachable from `src`. As visualized in Figure 7-7, Breadth First Search methodically explores the graph.

The queue maintains the search space in order of distance from the source; the nodes are shaded in the queue based on their distance from the source. Whenever a dead end is encountered, no new nodes are enqueued. Note that the target node `(2, 2)` is added to the queue within the `for` loop, but the visualization shows the moment when it is dequeued by the outer `while` loop. All nodes fewer than 2 steps away from the source have been processed, and the very last node in the queue is 3 steps away from the source.

Just for fun, I'll show a third approach for solving rectangular mazes *that takes into account how far away a node is from the target*. Both Depth First Search and Breadth First Search are *blind searches*: that is, they complete their search with only local information about adjacent nodes. The field of artificial intelligence has developed numerous path-finding algorithms that can more effectively complete a search when provided with information about the application domain.

4 There may be multiple paths of the same length, but Breadth First Search will discover one of these paths for which none can be shorter.

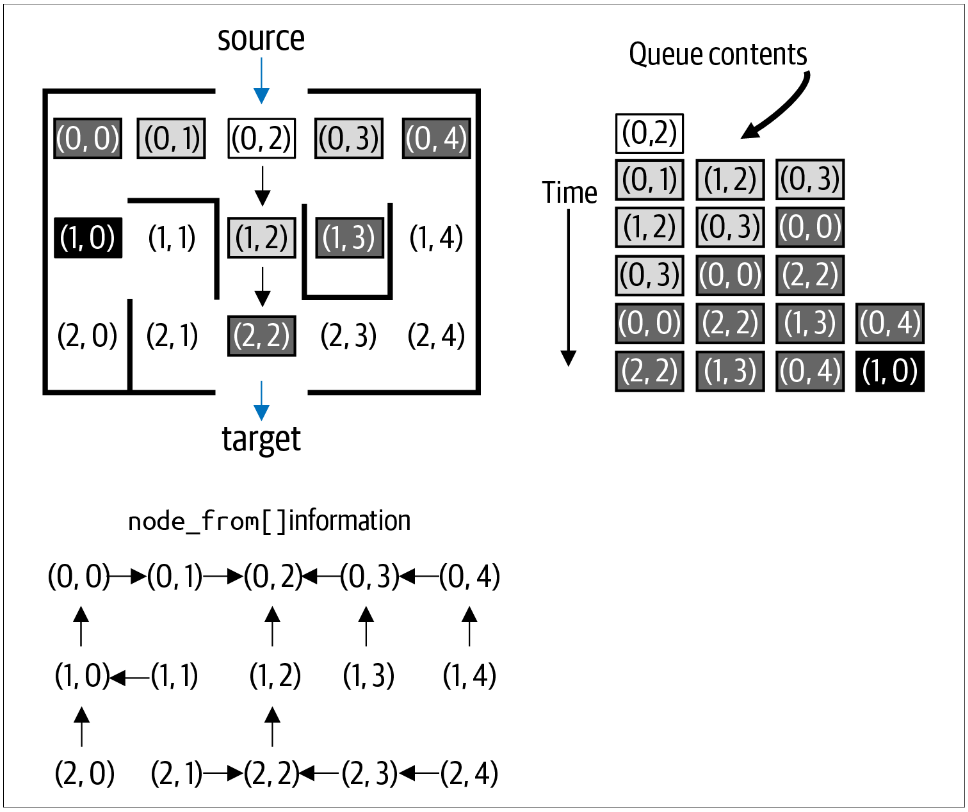


Figure 7-7. Breadth First Search finds shortest path to each node

A Guided Search explores nodes *in order of their shortest physical distance from the target*; to do this, I need to determine how far a node is from the target. Let's define the *Manhattan distance* between two cells in a maze as the sum of the number of rows and columns separating the two nodes.[5] For example, node `(2, 0)` in the lower-left-hand corner of the sample maze is four steps away from node `(0, 2)`, because it is both two rows away and two columns away.

Given the three adjacent nodes to the source node `(0, 2)` in the sample maze, Guided Search would first explore `(1, 2)` because it is only one step away from the target of `(2, 2)`; the other two adjacent nodes are both three steps away using the Manhattan distance. For this idea to work, the nodes being explored need to be stored using a data structure that allows you to retrieve the node closest to the target.

5 So called because in a city with a grid street layout, you can't move diagonally, only up, down, left, and right.

One common trick is to use a max priority queue, as presented in Chapter 4, by defining the priority of a node to be *the negative of the Manhattan distance from node to target*. Consider two nodes, where node *u* is ten steps away from the target, and node *v* is five steps away. If these nodes are stored in a max priority queue with (*u*, `-10`) and (*v*, `-5`), then the node with larger priority is *v*, which is the one closer to the target. The structure of Listing 7-6 is identical to the Breadth First and Depth First Search code, but it uses a priority queue to store the active search space of nodes to explore.

Listing 7-6. A Guided Search using Manhattan distance to control search

```
def guided_search(G, src, target):                       ❶
  from ch04.heap import PQ
  marked = {}                                            ❷
  node_from = {}                                         ❸

  pq = PQ(G.number_of_nodes())                           ❹
  marked[src] = True
  pq.enqueue(src, -distance_to(src, target))             ❺

  while not pq.is_empty():                               ❻
    v = pq.dequeue()

    for w in G.neighbors(v):
      if not w in marked:
        node_from[w] = v                                 ❼
        marked[w] = True
        pq.enqueue(w, -distance_to(w, target))           ❽

  return node_from                                       ❾

def distance_to(from_cell, to_cell):
  return abs(from_cell[0] - to_cell[0]) + abs(from_cell[1] - to_cell[1])
```

❶ Conduct a Guided Search over graph, `G`, starting from source node, `src`, knowing the `target` node to locate.

❷ The `marked` dictionary records nodes that have already been visited.

❸ Record how search got to each node: `node_from[w]` is the prior node working backward to `src`.

❹ Using the heap-based priority queue, you must pre-allocate sufficient space to include, potentially, all nodes in the graph.

❺ Mark and place `src` node into max priority queue to start the search using as its priority the negative of its distance to `target`.

❻ If the Guided Search has not yet completed, the node *closest* to `target` is the next node to explore.

❼ For each unmarked node, `w`, adjacent to `v`, remember that to get to `w`, the search came from `v`.

❽ Place `w` into its appropriate location in the priority queue, using *the negative of the Manhattan distance* as priority, and mark it so it isn't visited multiple times.

❾ Return the *structure of the search* that records for each node, `v`, the prior node from a search initiated at `src`.

The intelligence guiding the search is the `distance_to()` function, which computes the Manhattan distance between two nodes.

There is no guarantee that Guided Search will find the shortest path, plus it presumes in advance a single target to guide its search. In particular, it cannot outperform Breadth First Search, which is guaranteed to locate the shortest path not just to the target but to every node in the graph reachable from the source: in doing so, however, Breadth First Search might explore much more of the graph. The hope is that Guided Search reduces unnecessary searches on random maze graphs. Figure 7-8 presents a side-by-side comparison of these three search algorithms on the same maze.

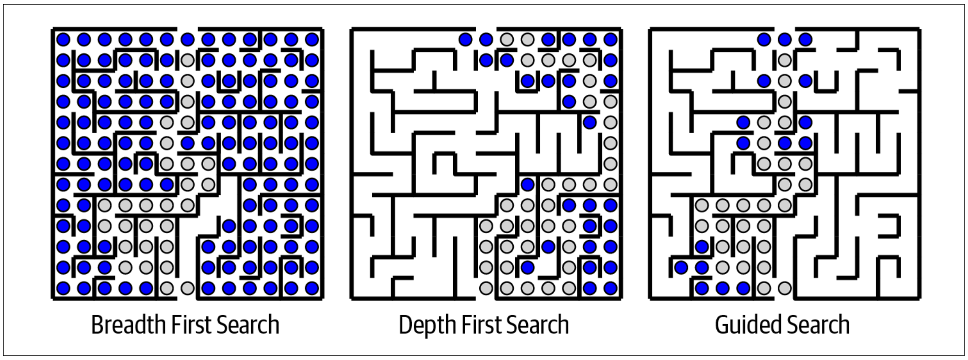


Figure 7-8. Comparing Depth First Search, Breadth First Search, and Guided Search

Breadth First Search will likely explore the most nodes because of its methodical nature. For Depth First Search to discover the shortest path, it must repeatedly choose the right direction to pursue, which is unlikely. There is no guarantee that Guided Search will compute the shortest path between source and target, but as shown in Figure 7-8, it reduces side explorations because it can aim toward the target.

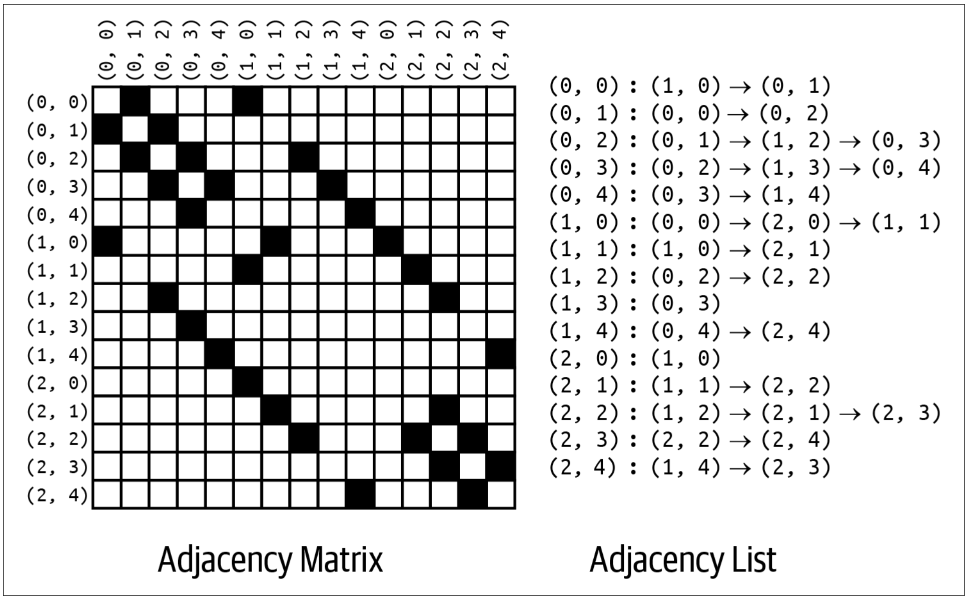
These search algorithms use a `marked` dictionary to ensure each of the N nodes is visited only once. This observation suggests that the runtime performance of each algorithm is O(N), but to confirm this you must validate the performance of the individual operations. With a `Stack`, its operations are performed in constant time—`push()`, `pop()`, and `is_empty()`. The only remaining concern is the efficiency of the `for w in G[v]` loop, which returns the adjacent nodes to *v*. To classify the performance of this `for` loop, you need to know how the graph stores edges. There are two options, visualized in Figure 7-9, for the maze from Figure 7-3: an adjacency matrix and an adjacency list.

Adjacency matrix

An *adjacency matrix* creates a two-dimensional N × N matrix, `M`, with N^2 Boolean entries. Each node, *u*, is assigned an integer index, u_{idx}, from 0 to N – 1. If `M`[u_{idx}][v_{idx}] is `True`, then there is an edge from *u* to *v*. These edges are shown as a shaded box, where *u* is the row label and *v* is the column label. With an adjacency matrix, retrieving all adjacent nodes for a node, *u*, requires O(N) runtime performance to check each entry in `M` for *u*, *regardless of how many adjacent nodes to u actually exist*. Since the `while` loop executes N times, and now the inner `for` loop requires O(N) performance to check N entries in `M`, this means that the search algorithm is classified as $O(N^2)$.

Adjacency list

An *adjacency list* uses a symbol table that associates for each node, *u*, a *linked list of adjacent nodes*. Retrieving all adjacent nodes for a node, *u*, requires runtime performance directly proportional to `d`, where `d` is the *degree* of node *u*, or the number of adjacent nodes to *u*. There is no predetermined ordering for these adjacent nodes, since that is based on how the edges were added to the graph. These linked lists are visualized for each node, *u*, in Figure 7-9. With an adjacency list, some nodes have few adjacent nodes, while others have a large number of adjacent nodes.



```
(0, 0) : (1, 0) → (0, 1)
(0, 1) : (0, 0) → (0, 2)
(0, 2) : (0, 1) → (1, 2) → (0, 3)
(0, 3) : (0, 2) → (1, 3) → (0, 4)
(0, 4) : (0, 3) → (1, 4)
(1, 0) : (0, 0) → (2, 0) → (1, 1)
(1, 1) : (1, 0) → (2, 1)
(1, 2) : (0, 2) → (2, 2)
(1, 3) : (0, 3)
(1, 4) : (0, 4) → (2, 4)
(2, 0) : (1, 0)
(2, 1) : (1, 1) → (2, 2)
(2, 2) : (1, 2) → (2, 1) → (2, 3)
(2, 3) : (2, 2) → (2, 4)
(2, 4) : (1, 4) → (2, 3)
```

Figure 7-9. Adjacency matrix versus adjacency list representation

The code fragment in Listing 7-7 shows that the runtime performance of Depth First Search is based on E, the number of edges in the graph, *when an adjacency list is used for representation*. Instead of counting the number of *nodes* being processed, count the number of *edges* being processed.

Listing 7-7. Code fragment showing performance based on the number of edges

```
while not stack.is_empty():
  v = stack.pop()
  for w in G[v]:
    if not w in marked:
      marked[w] = True
      stack.push(w)
       ...
```

You have already seen that each node, *v*, can be inserted into the stack only once. This means the `if` statement *will execute once for every adjacent node to v*. If you consider a graph with just two nodes, *u* and *v*, with a single edge (*u*, *v*), then the `if` statement will execute *twice*, once when processing the adjacent nodes to *u* and once when processing the adjacent nodes to *v*. So for an undirected graph, the number of times that the `if` statement executes is 2 × E, where E is the number of edges in the graph.

Put all this together—up to N `push()` and `pop()` invocations and 2 × E invocations of the `if` statement—and you can declare that the runtime performance of these search algorithms when using adjacency lists is O(N + E), where N is the number of nodes and E is the number of edges. The same is true for Breadth First Search, which uses a queue instead of a stack but exhibits the same runtime performance.

In some ways, these results are actually compatible; specifically, in an undirected graph with N nodes, E is less than or equal to N × (N – 1)/2 edges.[6] Regardless of whether the graph is stored using an adjacency matrix or an adjacency list, searching over graphs with a high number of edges will be proportional to N × (N – 1)/2, or $O(N^2)$, so in the *worst case* it will be $O(N^2)$.

Guided Search, however, relies on a priority queue to maintain the nodes closest to the designated target node. The `enqueue()` and `dequeue()` operations are O(`log` N) in the *worst case*. Since these methods are called N times, and each edge is visited twice, the *worst case* performance for Guided Search is O(N `log` N + E).

Directed Graphs

Graphs can also model problems where the relationship between two nodes is directional, typically represented as an edge with an arrow. A directed edge (*u*, *v*) only declares that *v* is adjacent to *u*: this edge does not make *u* adjacent to *v*. In this edge, *u* is the *tail*, while *v* is the *head*—this is easy to remember because the arrow head is adjacent to *v*. The graph in Figure 7-10 contains the edge (B3, C3) but not (C3, B3).

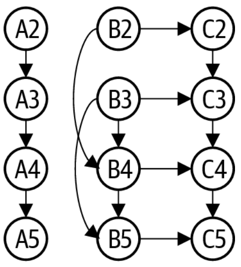


Figure 7-10. Sample directed graph with 12 nodes and 14 edges

The Depth First and Breadth First searches are still relevant for directed graphs: the only difference is that with the edge (*u*, *v*), *v* is adjacent to *u*, but the opposite is only true if the graph contains a separate edge (*v*, *u*). With directed graphs, many algorithms are simplified when using recursion, as shown in Listing 7-8. This code includes many familiar elements from the nonrecursive implementation.

6 The triangle numbers make yet another appearance!

Listing 7-8. Recursive implementation of Depth First Search on a directed graph

```
def dfs_search(G, src):          ❶
  marked = {}                    ❷
  node_from = {}                 ❸

  def dfs(v):                    ❹
    marked[v] = True             ❺
    for w in G[v]:
      if not w in marked:
        node_from[w] = v         ❻
        dfs(w)                   ❼

  dfs(src)                       ❽
  return node_from               ❾
```

❶ Conduct a Depth First Search over graph `G` starting from source node, `src`.

❷ The `marked` dictionary records nodes that have already been visited.

❸ Record how `dfs()` found each node: `node_from[w]` is the prior node working backward to `src`.

❹ Recursive method to continue search from an unmarked node, `v`.

❺ Be sure to mark that `v` has been visited.

❻ For each unmarked node, `w`, adjacent to `v`, remember that to get to `w`, the search came from `v`.

❼ In the recursive case, continue search in direction of unmarked node, `w`. When recursive call ends, continue with `for` loop over `w`.

❽ Invoke the initial recursive call on source node, `src`.

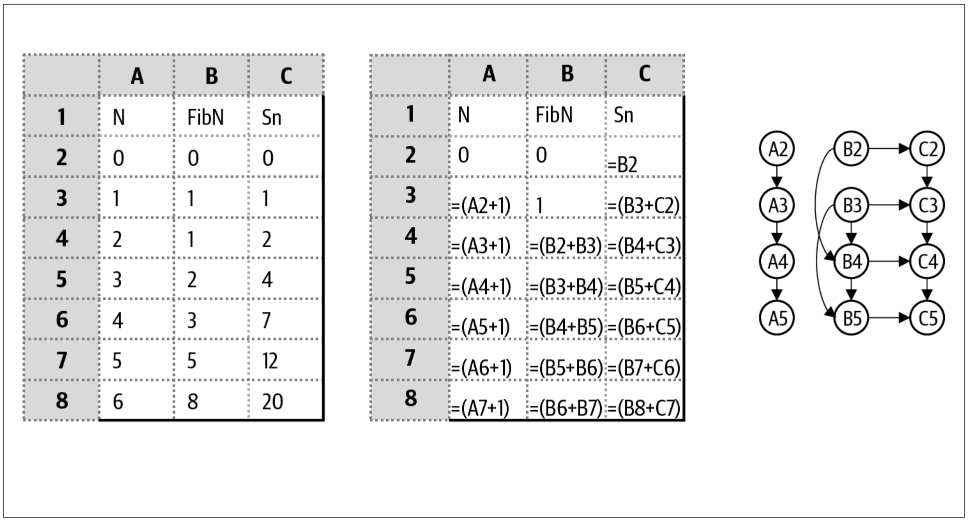
❾ Return the *structure of the search* that records for each node, `v`, the prior node from a search initiated at `src`.

A recursive algorithm remembers its partial progress using the recursive call stack, so there is no need for a `Stack` data type.

For every `dfs(v)` in the recursive call stack (where `v` is different with each invocation), node `v` is part of the active search space. In the base case of `dfs(v)`, node `v` has no unmarked adjacent nodes, so it performs no work. In the recursive case, for each unmarked adjacent node, `w`, a recursive `dfs(w)` invocation is launched. When it returns, the `for` loop over `w` continues, trying to find additional unmarked nodes adjacent to `v` to explore with `dfs()`.

As mentioned earlier in this book, Python limits the recursion depth to 1,000, which means some algorithms will not work for large problem instances. For example, a 50 x 50 rectangular maze has 2,500 cells. A Depth First Search will likely exceed the recursion limit. Instead, use a `Stack` data type to store the search progress.[7] The resulting code is often more complicated to understand, so for the rest of this chapter, I use a recursive Depth First Search.

Directed graphs can model application domains that have different problems to solve. Figure 7-11 depicts a small spreadsheet business application containing cells uniquely identified by a column and a row; cell B3 contains the constant 1, and this means the value of B3 is 1. The left image in Figure 7-11 shows the view presented to the user; the middle image shows the actual contents of each cell, including formulas. A cell can contain formulas that refer to other constants, and possibly the values computed from other formulas. These formulas are represented using infix expressions, which you learned in Chapter 6. For example, cell A4 contains the formula "= (A3 + 1)". Now, A3 contains the formula "= (A2 + 1)", which can be computed as 1 (since A2 contains the value 0), which means that A4 is computed to be 2. The code for this sample spreadsheet application is contained in the repository.



	A	B	C
1	N	FibN	Sn
2	0	0	0
3	1	1	1
4	2	1	2
5	3	2	4
6	4	3	7
7	5	5	12
8	6	8	20

	A	B	C
1	N	FibN	Sn
2	0	0	=B2
3	=(A2+1)	1	=(B3+C2)
4	=(A3+1)	=(B2+B3)	=(B4+C3)
5	=(A4+1)	=(B3+B4)	=(B5+C4)
6	=(A5+1)	=(B4+B5)	=(B6+C5)
7	=(A6+1)	=(B5+B6)	=(B7+C6)
8	=(A7+1)	=(B6+B7)	=(B8+C7)

$$\begin{tikzcd}
A2 \arrow[d] & B2 \arrow[r] \arrow[dd, bend right] & C2 \arrow[d] \\
A3 \arrow[d] & B3 \arrow[r] \arrow[d] \arrow[dd, bend right] & C3 \arrow[d] \\
A4 \arrow[d] & B4 \arrow[r] \arrow[d] & C4 \arrow[d] \\
A5 & B5 \arrow[r] & C5
\end{tikzcd}$$

Figure 7-11. Sample spreadsheet with underlying directed graph

7 Find the stack-based Depth First Search in `ch07.search` in the code repository.

This spreadsheet computes increasing values for N in column A, while column B contains the first seven Fibonacci numbers.[8] Column C contains the *running total* of the first N Fibonacci numbers (for example, the 12 in cell C7 represents the sum of 0 + 1 + 1 + 2 + 3 + 5). The right image in Figure 7-11 represents the directed graph that captures the relationships between the cells. For example, there is an edge from A2 to A3, reflecting that the value for A3 must change when A2 changes; another way to phrase this relationship is that the value for A2 must be known before the value for A3 can be computed.

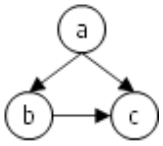
In a spreadsheet, if cell C2 contains the formula "= B2" and cell B2 contains the formula "= C2", these cells refer to each other, resulting in a *circular reference*, which is an error. Using the terminology of directed graphs, this situation would be a *cycle*, that is, a sequence of directed edges that starts at a node, `n`, and returns to `n`. Every spreadsheet program checks for cycles to make sure that its cells can be properly computed without error. Returning to Figure 7-11, the cells containing constants do not require any computations. The value for A3 needs to be computed before the value for A4 (which is subsequently needed when computing A5). The relationship between the B and C cells is more complicated, making it harder to know the order in which these cells should be computed, let alone whether a cycle even exists.

To operate safely, a spreadsheet application can maintain a directed graph of references between its cells to record dependencies between the cells. Whenever the user changes the contents of a cell, the spreadsheet must remove edges from this graph if the cell had formerly contained a formula. If the changed cell introduces a new formula, then the spreadsheet adds edges to capture the new dependencies in the formula. For example, given the spreadsheet from Figure 7-11, the user could mistakenly create a cycle by changing the contents of cell B2 to be the formula "= C5". With this change, a new edge, C5 → B2, would be added to the directed graph, leading to several cycles; here is one: [B2, B4, B5, C5, B2].

Given a directed graph, Depth First Search can determine whether a cycle exists in the directed graph. The intuition is that when Depth First Search finds a marked node *that is still part of the active search space*, a cycle exists. When `dfs(v)` returns, you are assured that *all nodes that are reachable from* `v` *have been marked*, and `v` is no longer part of the active search space.

8 Recall from Chapter 5 that these are the numbers 0, 1, 1, 2, 3, 5, and 8, computed by adding two successive terms.

A Depth First Search exploring a directed graph with just three nodes can encounter a `marked` node in a graph without a cycle. Initiating `dfs()` from node `a` could explore to `b` and finally `c`, which is a dead end. When the search returns to process the remaining adjacent nodes to `a`, although `c` is `marked`, no cycle exists.



This Cycle Detection algorithm differs from the other algorithms in this chapter because there is no initial dedicated source node from which the exploration starts. The question is whether a cycle exists *anywhere* in the graph, so the algorithm, described in Listing 7-9, has to investigate potentially every node in the graph.

Listing 7-9. Detecting cycles in a directed graph using Depth First Search

```
def has_cycle(DG):
  marked = {}
  in_stack = {}

  def dfs(v):                                      ❶
    in_stack[v] = True                             ❷
    marked[v] = True                               ❸

    for w in DG[v]:
      if not w in marked:
        if dfs(w):                                 ❹
          return True
      else:
        if w in in_stack and in_stack[w]:          ❺
          return True

    in_stack[v] = False                            ❻
    return False

  for v in DG.nodes():                             ❼
    if not v in marked:
      if dfs(v):                                   ❽
        return True
  return False
```

❶ Conduct a Depth First Search over graph, `DG`, starting from `v`.

❷ `in_stack` records nodes that are in the recursive call stack. Mark that `v` is now part of a recursive call.

❸ The `marked` dictionary records nodes that have already been visited.

❹ For each unmarked node, `w`, adjacent to `v`, initiate recursive `dfs()` on `w`, and if `True` is returned, a cycle is detected, so it returns `True` as well.

❺ If a node, `w`, is marked as visited, it could still be in our call stack—if it is, then a cycle has been detected.

❻ Equally important, when the `dfs()` recursive call ends, set `in_stack[v]` to `False` since `v` is no longer on the call stack.

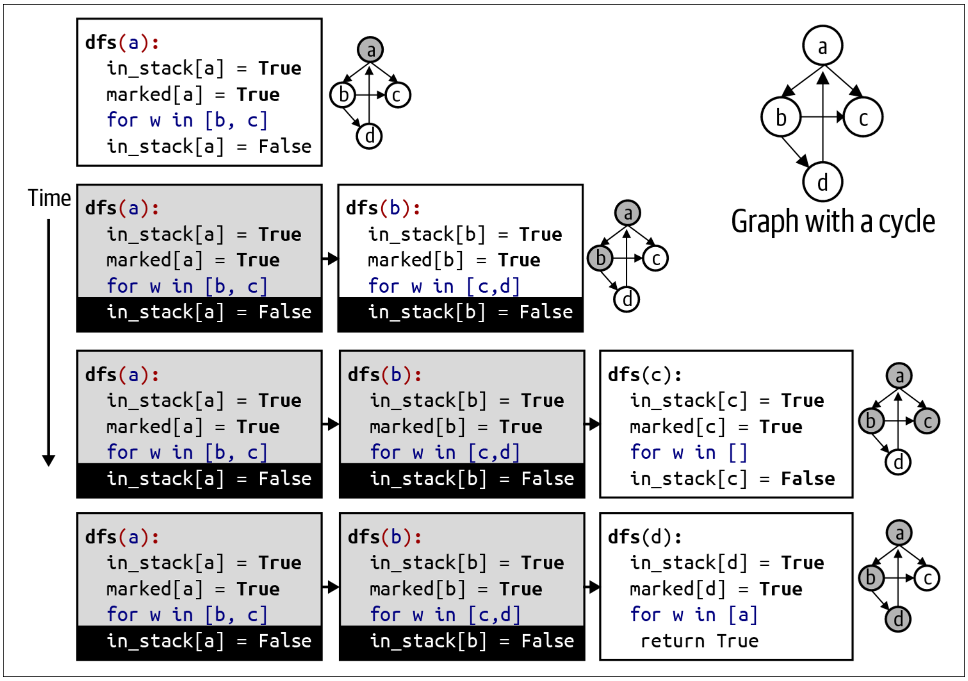
❼ Investigate each unmarked node in the directed graph.

❽ If invoking `dfs(v)` on a node, `v`, detects a cycle, return `True` immediately.

As `dfs()` recursive calls execute, more of the graph is explored until eventually each node is `marked`—even those with no edges.

If you want to also compute the actual cycle, try the challenge exercise at the end of this chapter that modifies `has_cycle()` to compute and return the first detected cycle in a directed graph. Figure 7-12 visualizes the recursive execution of `dfs()`. Each discovered node is eventually marked, but only nodes *in the active search space*—those nodes where `in_stack[]` is `True`—are highlighted as the recursion proceeds and unwinds. The image shows the moment in the recursion when the cycle [`a`, `b`, `d`, `a`] is detected. When exploring the adjacent nodes to `d`, the marked node, `a`, is encountered, but this doesn't immediately mean that a cycle exists. The algorithm must check whether `in_stack[a]` is `True` to confirm that a cycle exists.

The final recursive invocation of `dfs(d)` has not yet ended, which is why `in_stack[d]` is still `True`. To summarize, when the recursive `dfs()` function encounters a node, `n`, that has already been `marked` *and* `in_stack[n]` is `True`, a cycle has been found.



```
dfs(a):
  in_stack[a] = True
  marked[a] = True
  for w in [b, c]
  in_stack[a] = False
```

```
dfs(b):
  in_stack[b] = True
  marked[b] = True
  for w in [c,d]
  in_stack[b] = False
```

```
dfs(c):
  in_stack[c] = True
  marked[c] = True
  for w in []
  in_stack[c] = False
```

```
dfs(d):
  in_stack[d] = True
  marked[d] = True
  for w in [a]
   return True
```

Figure 7-12. Visualizing execution of Depth First Search for Cycle Detection

Assuming that a spreadsheet contains no circular references, in what order should its cells be computed? Returning to the spreadsheet example in Figure 7-11, the cells that contain constants (such as A1) are not involved in any computation, so they do not matter. The cell B4 contains a formula that directly depends on both B2 and B3, so these two cells must be computed before B4. One linear ordering that works is:

B2, C2, B3, C3, B4, C4, B5, C5, A2, A3, A4, A5

The preceding ordering is one possible result from `topological_sort()`, shown in Listing 7-10. The structure of this algorithm is identical to the Cycle Detection algorithm described earlier. It relies on a recursive Depth First Search to explore the graph. When `dfs(v)` is about to return from its recursive invocation, all nodes that are reachable from *v* have been `marked`. This means that `dfs()` has already visited all "downstream" nodes that are dependent on *v*, so it adds *v* to the growing list of nodes (in reverse order) whose dependencies have been processed.

Listing 7-10. Topological sort over the directed graph

```
def topological_sort(DG):
  marked = {}
  postorder = []                    ❶

  def dfs(v):                       ❷
    marked[v] = True                ❸
    for w in DG[v]:
      if not w in marked:
        dfs(w)                      ❹
    postorder.append(v)             ❺

  for v in DG.nodes():
    if not v in marked:             ❻
      dfs(v)

  return reversed(postorder)        ❼
```

❶ Use a list to store (in reverse order) a linear ordering of nodes to be processed.

❷ Conduct a Depth First Search over `DG` starting from `v`.

❸ The `marked` dictionary records nodes that have already been visited.

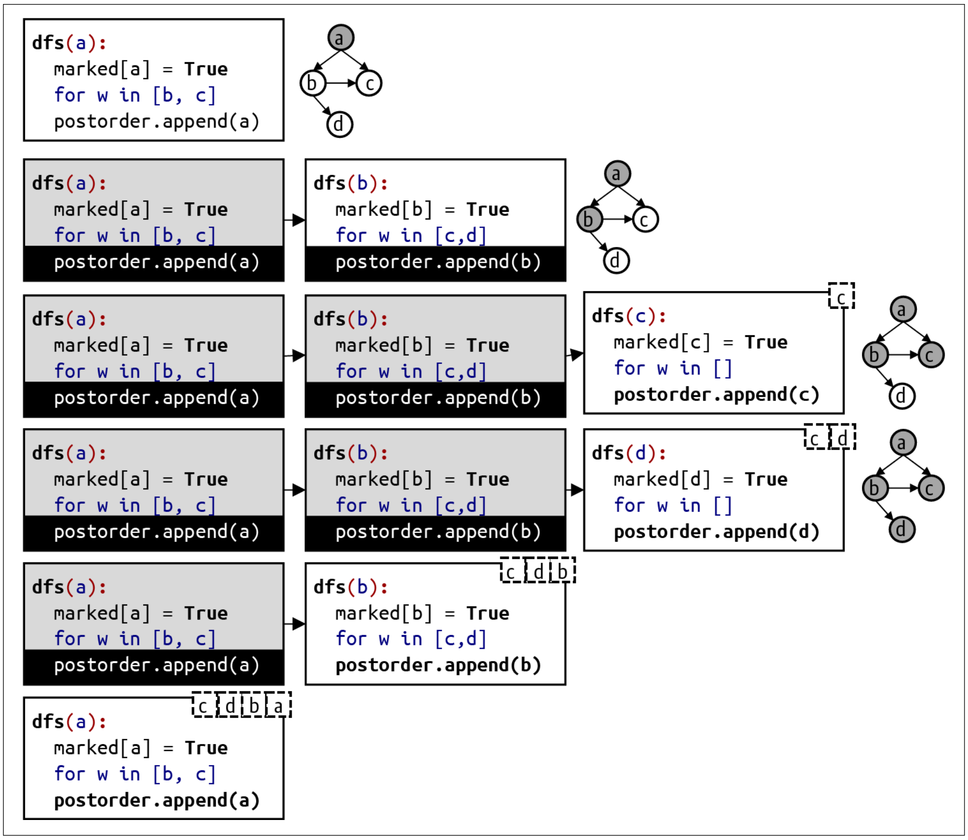
❹ For each unmarked node, `w`, adjacent to `v`, recursively explore `dfs(w)`.

❺ When `dfs(v)` gets to this key step, it has fully explored all nodes that (recursively) depend on `v`, so append `v` to `postorder`.

❻ Ensure that all unmarked nodes are visited. Note that each time `dfs(v)` is invoked, a different subset of graph `DG` is explored.

❼ Because the list holds the linear ordering in reverse order, return its reverse.

This code is nearly identical to the Cycle Detection algorithm, except it maintains the `postorder` structure instead of `in_stack[]`. Using a similar runtime analysis, you can see that each node has one chance to be explored with `dfs()`, and the inner `if` statement is executed once for every directed edge in the graph. Since appending to a list has constant time performance (see Table 6-1), this guarantees that the runtime performance of Topological Sort is O(N + E), where N is the number of nodes and E is the number of edges.



```
dfs(a):
  marked[a] = True
  for w in [b, c]
  postorder.append(a)
```

```
dfs(a):
  marked[a] = True
  for w in [b, c]
  postorder.append(a)
```

```
dfs(b):
  marked[b] = True
  for w in [c,d]
  postorder.append(b)
```

```
dfs(a):
  marked[a] = True
  for w in [b, c]
  postorder.append(a)
```

```
dfs(b):
  marked[b] = True
  for w in [c,d]
  postorder.append(b)
```

```
dfs(c):
  marked[c] = True
  for w in []
  postorder.append(c)
```

```
dfs(a):
  marked[a] = True
  for w in [b, c]
  postorder.append(a)
```

```
dfs(b):
  marked[b] = True
  for w in [c,d]
  postorder.append(b)
```

```
dfs(d):
  marked[d] = True
  for w in []
  postorder.append(d)
```

```
dfs(a):
  marked[a] = True
  for w in [b, c]
  postorder.append(a)
```

```
dfs(b):
  marked[b] = True
  for w in [c,d]
  postorder.append(b)
```

```
dfs(a):
  marked[a] = True
  for w in [b, c]
  postorder.append(a)
```

Figure 7-13. Visualizing execution of Depth First Search for Topological Sort

When each `dfs()` completes its execution in Listing 7-10, as shown visually in Figure 7-13, `postorder` contains the ordered list of nodes whose dependencies are satisfied; these are shown in dashed boxes. This list is reversed and returned at the end of `topological_sort()`. When the spreadsheet application loads a spreadsheet document, it can recompute the cells in the order determined by Topological Sort.

Graphs with Edge Weights

Some application domains modeled using graphs have a numeric value associated with each edge, typically called the *weight* of an edge. These edge weights can appear in undirected or directed graphs. For now, assume that all edge weights are positive values greater than 0.

The Stanford Large Network Dataset Collection contains some large data sets for social networks. Computer scientists have studied the "traveling salesman problem" (TSP) for decades, and numerous data sets are available (TSPLIB). A large highway data set is available at Travel Mapping Graph Data. I'd like to thank James Teresco for graciously providing the Massachusetts highway data set.

From the data set of highway fragments in Massachusetts, let's create a graph where each node in the graph represents a *waypoint* from the data set, represented by a (latitude, longitude) pair of values. For example, one waypoint is the intersection of highways I-90 and I-93 in Boston. It is identified by a latitude value of 42.34642 (which means it is north of the equator) and a longitude value of –71.060308 (which means it is west of Greenwich, England). An edge between two nodes represents a highway fragment: the weight of the edge is the length of the highway fragment in miles. Different roads connect these waypoints together, leading to the highway infrastructure shown in Figure 7-14.

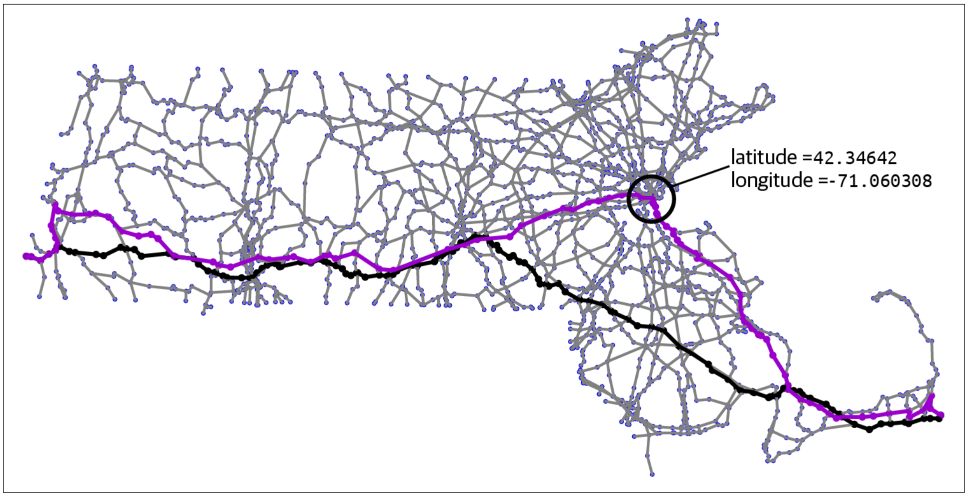


Figure 7-14. Modeling highway infrastructure in Massachusetts

To determine the shortest route (in terms of total mileage) from the westernmost highway in Massachusetts (on the New York Border) to the easternmost highway (on Cape Cod), start with Breadth First Search to compute a 236.5-mile path (highlighted in Figure 7-14). This 99-edge path passes through the identified waypoint on highway I-90/I-93 in Boston and is the shortest path (from the source to the target) in terms of *the total number of edges*. But is it the shortest total path *in accumulated mileage* when considering edge weights? It turns out the answer is no.

We know Depth First Search offers no guarantee on path length: Figure 7-15 contains the wandering 485.2-mile journey with 267 edges produced by Depth First Search. A Guided Search algorithm makes a poor decision early in its journey (not shown here) to compute a 245.2-mile path with 141 edges. The other visualized path in Figure 7-14, which steadily progresses in a southeasterly direction, contains 128 edges and requires only 210.1 miles; Dijkstra's algorithm shows how to compute this solution.

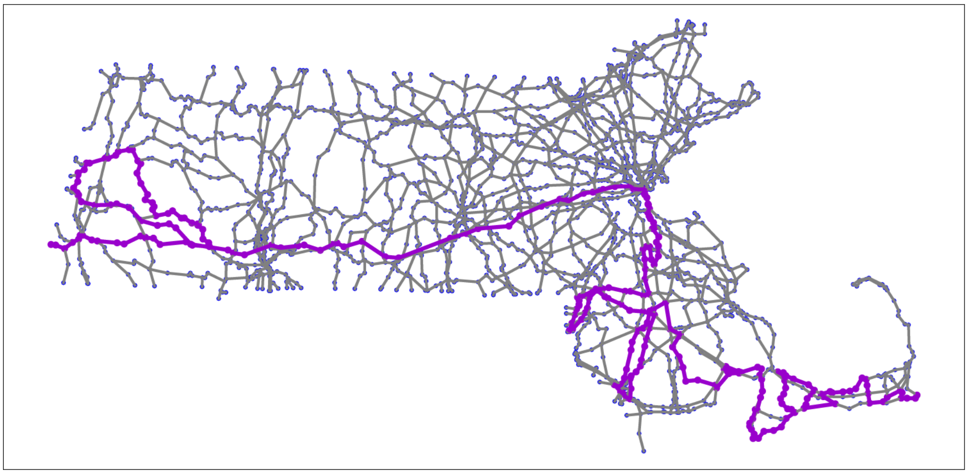


Figure 7-15. Inefficient path resulting from Depth First Search

Dijkstra's Algorithm

Edsger Dijkstra (pronounced DIKE-stra), a scientist from the Netherlands, was one of the intellectual founders of the discipline of computer science, and his algorithms are as elegant as they are insightful. Dijkstra developed an algorithm that computes the shortest path *of accumulated edge weights* from a designated source node to all reachable nodes in a weighted graph. This problem is known as the "single-source shortest path" problem. Given an undirected (or directed) graph, G, with non-negative *weights* associated with each edge,[9] Dijkstra's algorithm computes `dist_to[]` and `edge_to[]` structures, where `dist_to[v]` is the length of the shortest accumulated path from the source node to `v`, and `edge_to[]` is used to recover the actual path.

A sample weighted, directed graph is shown in Figure 7-16. The edge (`a`, `b`) has a weight of 6. There is an edge from `a` to `c` with weight of 10, but a path from `a` to `b` (with weight of 6) to `c` (with weight of 2) has an accumulated total of 8, which

9 This allows for some edge weights to be zero. If any edge weight is negative, you will need the Bellman–Ford algorithm (presented later in this chapter).

represents a shorter path. The shortest path from `a` to `c` contains two edges and a total weight of 8.

When the graph is directed, it may be impossible to construct a path between two nodes. The shortest distance from `b` to `c` is 2, but the shortest distance from `b` to `a` is infinity because there is no way to construct a path using the existing edges.

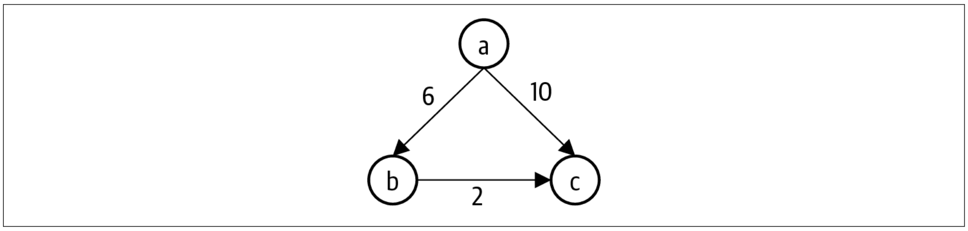


Figure 7-16. Shortest path from `a` to `c` has accumulated total of 8

Dijkstra's algorithm requires an abstract data type known as the *indexed min priority queue*, designed specifically to work with graph algorithms. The indexed min priority queue extends the priority queue data type introduced in Chapter 4. An indexed min priority queue associates a priority with each value. The min priority queue is constructed with an initial storage based on N, where N is the number of nodes in the graph being processed. The `dequeue()` operation removes the value whose priority *is smallest numerically*, in contrast to the max priority queue discussed in Chapter 4.

The most important operation is `decrease_priority(value, lower_priority)` that efficiently *reduces the priority of* `value` *to a lower priority*. In effect, `decrease_priority()` can adjust the priority of an existing value so that it *could move ahead of other values in the priority queue*. The priority queue implementations described earlier are unable to provide an efficient `decrease_priority()` function, since they would have to search the entire priority queue in O(N) to locate the value whose priority changes.

The indexed min priority queue is typically restricted to use only integer values ranging from 0 to N – 1, which allows it to store and access data easily in arrays. I use a Python dictionary to remove this restriction.

As shown in Listing 7-11, the structure and functionality of `IndexedMinPQ` is nearly the same as the heap-based max priority queue presented in Chapter 4. Instead of storing `Entry` objects, `IndexedMinPQ` stores two lists: `values[n]` stores the value for the nth item in the heap, while `priorities[n]` stores its associated priority. The `swim()` and `sink()` methods are identical to the heap-based priority implementation (shown in Listing 4-2 and Listing 4-3), so they are omitted. The primary change is a `location` dictionary that stores the index position in these lists for each value in

IndexedMinPQ. This extra information allows you to determine the location in the heap for any value in amortized constant O(1) time, using the results of hashing (discussed in Chapter 3).

The changes to swap() ensure that whenever two items in the heap are swapped, their respective entries in location are updated. This way IndexedMinPQ can efficiently locate any value in the priority queue.

Listing 7-11. Structure of an indexed min priority queue

```
class IndexedMinPQ:
  def less(self, i, j):                                              ❶
    return self.priorities[i] > self.priorities[j]

  def swap(self, i, j):
    self.values[i],self.values[j] = self.values[j],self.values[i]    ❷
    self.priorities[i],self.priorities[j] = self.priorities[j],self.priorities[i]

    self.location[self.values[i]] = i                                ❸
    self.location[self.values[j]] = j

  def __init__(self, size):
    self.N          = 0
    self.size       = size
    self.values     = [None] * (size+1)                              ❹
    self.priorities = [None] * (size+1)

    self.location   = {}                                             ❺

  def __contains__(self, v):                                         ❻
    return v in self.location

  def enqueue(self, v, p):
    self.N += 1

    self.values[self.N], self.priorities[self.N] = v, p              ❼
    self.location[v] = self.N                                        ❽
    self.swim(self.N)
```

❶ Because this is a min priority queue, item i is of lower priority than item j *if its priority is a larger numeric value.*

❷ swap() switches the values and priorities for items i and j.

❸ swap() updates the respective locations for items i and j.

❹ values stores the value of the nth item; priorities stores the priorities of the nth item.

❺ location is a dictionary that returns the index position into values and priorities for each value that is enqueued.

❻ Unlike a traditional priority queue, an indexed min priority queue can inspect location to determine in amortized O(1) time whether a value is being stored in the priority queue.

❼ To enqueue a (v, p) entry, place v in values[N] and p in priorities[N], which is next available bucket.

❽ enqueue() must also associate this new index location with v before invoking swim() to guarantee the heap-ordered property.

As you should expect with a heap, enqueue() first stores the value, v, and its associated priority, p, at the end of the values[] and priorities[] lists, respectively. To fulfill its obligation to IndexedMinPQ, it also records that value v is stored in index location N (recall that a heap uses 1-based indexing to make the code easier to understand). It invokes swim() to ensure the heap-ordered property for IndexedMinPQ.

Through its location[] array, IndexedMinPQ can find the location in the heap for any value that it stores. The decrease_priority() method shown in Listing 7-12 can move any value in the IndexedMinPQ *closer to the front of the priority queue*. The only restriction is that you can only *decrease* the numeric value of the priority—which makes it potentially more important—and swim the item up into its proper location.

Listing 7-12. Decreasing priority for a value in IndexedMinPQ

```
def decrease_priority(self, v, lower_priority):
  idx = self.location[v]                              ❶
  if lower_priority >= self.priorities[idx]:          ❷
     raise RuntimeError('...')

  self.priorities[idx] = lower_priority               ❸
  self.swim(idx)                                      ❹
```

❶ Find the location, idx, in the heap where v is found.

❷ If lower_priority is actually not lower than the existing priority in priorities[idx], raise a RuntimeError.

❸ Change the priority for value v to be the lower priority.

❹ Reestablish heap-ordered property if necessary by swimming this value up.

dequeue() removes the value with smallest priority value (which means it is the most important). The IndexedMinPQ implementation is more complicated, because it has to properly maintain the location dictionary, as shown in Listing 7-13.

Listing 7-13. Removing highest-priority value in IndexedMinPQ

```
def dequeue(self):
  min_value = self.values[1]                              ❶

  self.values[1] = self.values[self.N]                    ❷
  self.priorities[1] = self.priorities[self.N]
  self.location[self.values[1]] = 1

  self.values[self.N] = self.priorities[self.N] = None    ❸
  self.location.pop(min_value)                            ❹

  self.N -= 1                                             ❺
  self.sink(1)
  return min_value                                        ❻
```

❶ Remember min_value, the value with highest priority.

❷ Move the item at location N to the top-level location 1 and ensure that location records the new index position for this value.

❸ Remove all trace of the former min_value being removed.

❹ Remove min_value entry from location dictionary.

❺ Reduce number of entries *before* invoking sink(1) to reestablish heap-ordered property.

❻ Return the value associated with entry of highest priority (which is the one with smallest magnitude).

The IndexedMinPQ data structure ensures the invariant that if v is a value stored by the priority queue, then location[v] points to an index location, idx, such that values[idx] is v and priorities[idx] is p, where p is the priority for v.

Dijkstra's algorithm uses an IndexedMinPQ to compute the length of a shortest path from a designated src node to any node in a graph. The algorithm maintains a dictionary, dist_to[v], to record the length of the shortest known computed path from src to each v in the graph: this value may be infinite for nodes not reachable from src. As the algorithm explores the graph, it looks for two nodes, *u* and *v*, connected by an edge with weight of wt such that dist_to[u] + wt < dist_to[v]: in other words, the distance from `src` to `v` is shorter if you follow the path from `src` to `u` and then cross to `v` along the edge (`u`, `v`).

Dijkstra's algorithm shows how to find these special edges methodically, similar to the way Breadth First Search uses a queue to explore nodes based on their distance from `src` (in terms of number of edges). `dist_to[v]` summarizes the results of the active search, and `IndexedMinPQ` organizes the remaining nodes to be explored by priority, which is defined as the accumulated length of the shortest path for each node from `src`. When the algorithm starts, `dist_to[src]` is 0 because that node is the source, and all other distances are infinity. All nodes are then enqueued in the `IndexedMinPQ` with priority equal to 0 (for the source node, `src`) or infinity for other nodes.

Dijkstra's algorithm does not need to mark nodes as having been visited, since the min priority queue contains only those active nodes to be explored. One by one, the algorithm removes from the min priority queue the node whose total accumulated distance is the smallest.

Listing 7-14. Dijkstra's algorithm to solve single-source shortest path problem

```
def dijkstra_sp(G, src):
  N = G.number_of_nodes()

  inf = float('inf')                                ❶
  dist_to = {v:inf for v in G.nodes()}
  dist_to[src] = 0

  impq = IndexedMinPQ(N)                            ❷
  impq.enqueue(src, dist_to[src])
  for v in G.nodes():
    if v != src:
      impq.enqueue(v, inf)

  def relax(e):
    n, v, weight = e[0], e[1], e[2][WEIGHT]         ❺
    if dist_to[n] + weight < dist_to[v]:            ❻
      dist_to[v] = dist_to[n] + weight              ❼
      edge_to[v] = e                                ❽
      impq.decrease_priority(v, dist_to[v])         ❾

  edge_to = {}                                      ❸
  while not impq.is_empty():
    n = impq.dequeue()                              ❹
    for e in G.edges(n, data=True):
      relax(e)

  return (dist_to, edge_to)
```

❶ Initialize `dist_to` dictionary to infinity for all nodes except `src`, which is 0.

❷ Enqueue all N nodes into `impq` to prepare for `while` loop.

❸ `edge_to[v]` records the edge terminating at `v` found during the search.

❹ Find node, `n`, that has shortest computed path from `src`. Explore its edges (`n`, `v`, `weight`) to see if a new shortest path to `v` has been found. `networkx` requires `data = True` to retrieve edge weights.

❺ Extract `n`, `v`, and `weight` from the edge (`n`, `v`).

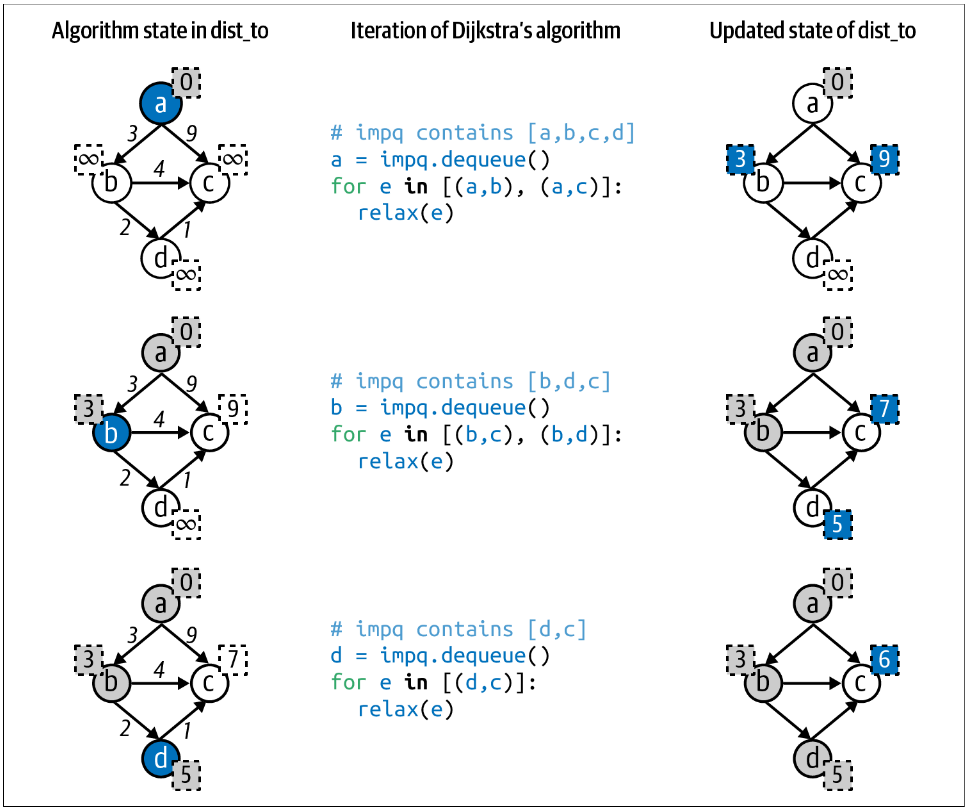
❻ If distance to `n` added to edge `weight` to `v` is smaller than best path so far to `v`, then a shorter path has been found.

❼ Update the shortest known distance to `v`.

❽ Record the edge (`n`, `v`) that brought Dijkstra's algorithm to `v` along the new shortest path.

❾ Most importantly, reduce priority in `impq` to new shortest distance so `while` loop will be able to retrieve node with shortest computed path.

Figure 7-17 presents the first three iterations through the `while` loop in Listing 7-14. `IndexedMinPQ` stores each node, `n`, using as priority the smallest computed distance, `dist_to[n]`, which is shown in a small dashed box attached to each node. With each pass through the `while` loop, a node, `n`, is removed from `impq` to check whether its edges lead to a new shortest path from `src` to some node, `v`, by traveling from `src` to `n` and then from `n` to `v`. This process is called *relaxing an edge*. The `IndexedMinPQ` prioritizes which nodes are explored first—in this way, Dijkstra's algorithm guarantees that *the shortest path for each node dequeued from* `impq` *is correct*.



```
# impq contains [a,b,c,d]
a = impq.dequeue()
for e in [(a,b), (a,c)]:
  relax(e)
```

```
# impq contains [b,d,c]
b = impq.dequeue()
for e in [(b,c), (b,d)]:
  relax(e)
```

```
# impq contains [d,c]
d = impq.dequeue()
for e in [(d,c)]:
  relax(e)
```

Figure 7-17. Executing Dijkstra's algorithm on a small graph

The runtime performance of Dijkstra's algorithm is based on several factors:

The cost of enqueuing all N nodes
: The first node enqueued is `src` with priority of 0. All remaining N – 1 nodes are enqueued with infinite priority; since infinity is greater than or equal to every value already in `impq`, `swim()` does nothing, resulting in O(N) performance.

The cost of retrieving N nodes from `impq`
: Dijkstra's algorithm dequeues each node from `impq`. Since `impq` is stored as a binary heap, `dequeue()` is O(`log` N), which means the total time to remove all nodes is O(N `log` N) in the *worst case*.

The cost of accessing all edges in G
: The structure of the graph determines the runtime performance of retrieving all edges in the `for e in G.edges()` loop. If the graph stores edges using an adjacency matrix, accessing all edges requires O(N^2) performance. If the graph stores edges using an adjacency list, retrieving all edges requires just O(E + N).

The cost of relaxing all E edges

The `relax()` function is called on all edges in the graph. Each one has a chance to reduce the shortest computed path length to some node, so there can be as many as E invocations of `decrease_priority()`. This function relies on the `swim()` binary heap function, whose runtime performance is O(`log` N). The accumulated time will have a runtime performance of O(E `log` N).

If the graph stores edges using an adjacency list, Dijkstra's algorithm has an O((E + N) `log` N) classification. If the graph uses an adjacency matrix instead, the performance is O(N^2). For large graphs the matrix representation is simply inefficient.

Dijkstra's algorithm computes two structures: `dist_to[v]` contains the length of the shortest path by accumulated edge weights from `src` to `v`, while `edge_to[v]` contains the last edge (*u*, *v*) on the actual shortest path from `src` to `v`. The full path from `src` to each `v` can be recovered, much like Listing 7-4, except this time following the `edge_to[]` structure backward, as shown in Listing 7-15.

Listing 7-15. Recovering actual path from `edge_to[]`

```
def edges_path_to(edge_to, src, target):  ❶
  if not target in edge_to:
    raise ValueError('Unreachable')        ❼

  path = []
  v = target                              ❷
  while v != src:
    path.append(v)                        ❸
    v = edge_to[v][0]                     ❹

  path.append(src)                        ❺
  path.reverse()                          ❻
  return path
```

❶ `edge_to[]` structure is needed to recover path from `src` to any `target`.

❷ To recover the full path, start at `target`.

❸ As long as `v` is not `src`, append `v` to `path`, a backward list of nodes found on path from `src` to `target`.

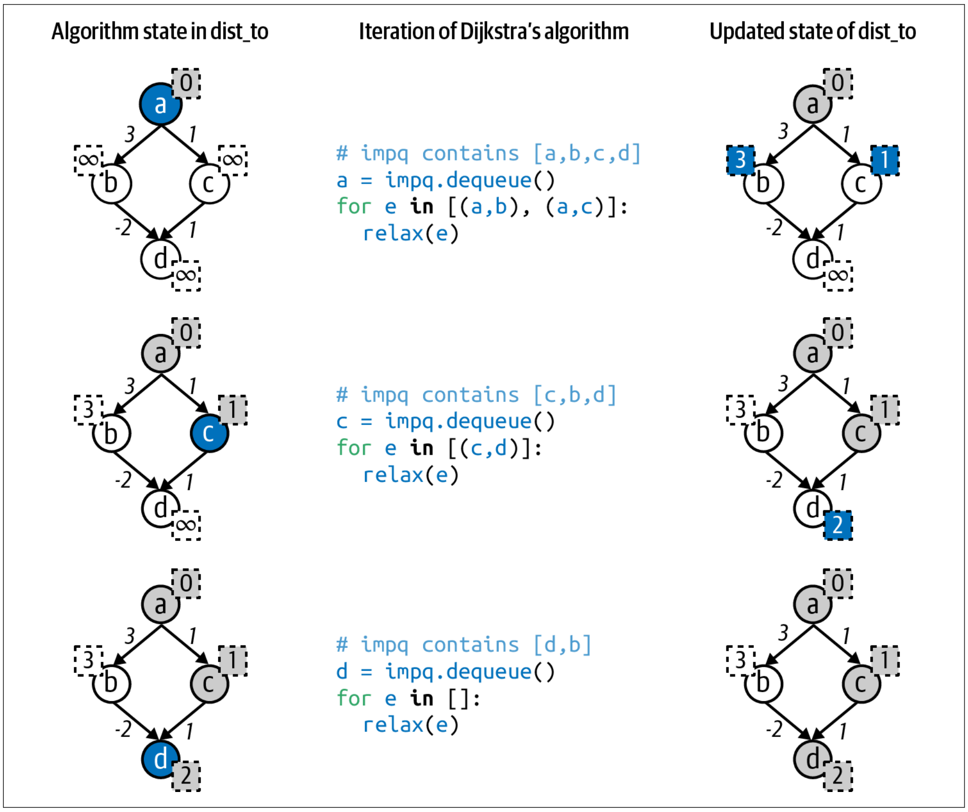
❹ Set `v` to become `u`, the prior node in the edge (`u`, `v`) from `edge_to[v]`.

❺ Once `src` is encountered, the `while` loop terminates, so `src` must be appended to complete the backward `path`.

❻ Reverse the list so all nodes appear in proper order from `src` to `target`.

❼ If `target` is not in `edge_to[]`, it is not reachable from `src`.

Dijkstra's algorithm will work *as long as all edge weights are nonnegative*. A graph might have a negative edge, for example, because it represents the refund of a financial transaction. If an edge has a negative edge weight, it could break Dijkstra's algorithm (as shown in Figure 7-18).



```
# impq contains [a,b,c,d]
a = impq.dequeue()
for e in [(a,b), (a,c)]:
  relax(e)
```

```
# impq contains [c,b,d]
c = impq.dequeue()
for e in [(c,d)]:
  relax(e)
```

```
# impq contains [d,b]
d = impq.dequeue()
for e in []:
  relax(e)
```

Figure 7-18. Negative edge weight in wrong place breaks Dijkstra's algorithm

In Figure 7-18, Dijkstra's algorithm processes three nodes from `impq` until just `b` is left. As you can see in the last row, Dijkstra's algorithm has computed its current shortest path from `a` to `d`. In its final pass through its `while` loop (not shown in the figure), Dijkstra's algorithm will remove node `b` from `impq` and relax the edge (`b`, `d`). Unfortunately, this edge suddenly reveals a shorter path to `d`. However, Dijkstra's algorithm has already removed node `d` from `impq`, finalizing its shortest path computation. Dijkstra's algorithm cannot "go back" and adjust the shortest path, so it fails.

Dijkstra's algorithm can fail with negative edge weights since it assumes that extending an existing path with a new edge will only maintain or increase the total distance from the source. The Bellman–Ford algorithm computes the shortest total distance from `src` to any other node in the graph, even with negative edge weights, with one exception: if a *negative cycle* exists in the graph, then the concept of a shortest path does not apply. The graph on the left side of Figure 7-19 has two negative edges, but no negative cycle. Using the edge (`a`, `b`), the shortest distance from `a` to `b` is 1. If you travel over the longer path `a` → `b` → `d` → `c` → `b`, the total accumulated edge weight distance is 2, so the shortest path between `a` and `b` remains 1. In the graph on the right, however, there is a *negative cycle* between nodes `b`, `d`, and `c`; that is, if you travel the edges clockwise in order from `b` → `d` → `c` → `b`, the total accumulated edge weight is –2. In this graph, the shortest distance between `a` and `b` has no meaning: you can construct a path to make this distance any odd negative number by cycling through the `b` → `d` → `c` → `b` loop a number of times. The accumulated edge weights for `a` → `b` → `d` → `c` → `b` → `d` → `c` → `b` is –3, for example.

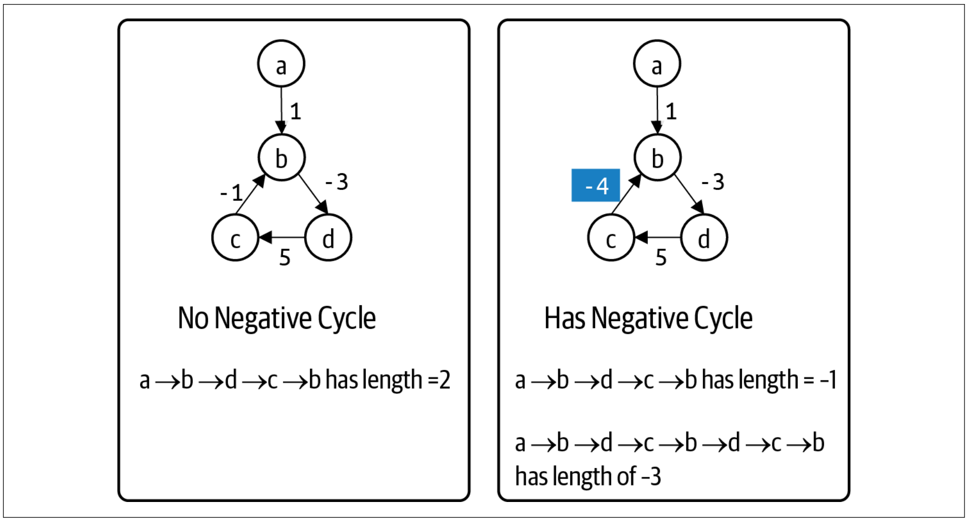


Figure 7-19. Two graphs with negative edge weights, but only one has negative cycle

The Bellman–Ford implementation provides a completely different approach to solving the same single-source shortest path problem. It works even when there are edges with a negative weight. Listing 7-16 shows the Bellman–Ford implementation, which contains many familiar elements from Dijkstra's algorithm. The good news is that you do not have to search for a negative cycle in the graph as you had to do with Topological Sort shown earlier in the chapter. As Bellman–Ford executes, it can detect when a negative cycle exists and raises a runtime exception in response.

Listing 7-16. Bellman–Ford algorithm implementation

```
def bellman_ford(G, src):
  inf = float('inf')
  dist_to = {v:inf for v in G.nodes()}                ❶
  dist_to[src] = 0
  edge_to = {}                                        ❷

  def relax(e):
    u, v, weight = e[0], e[1], e[2][WEIGHT]
    if dist_to[u] + weight < dist_to[v]:              ❺
      dist_to[v] = dist_to[u] + weight                ❻
      edge_to[v] = e                                  ❼
      return True                                     ❽
    return False

  for i in range(G.number_of_nodes()):                ❸
    for e in G.edges(data=True):                      ❹
      if relax(e):
        if i == G.number_of_nodes()-1:                ❾
          raise RuntimeError('Negative Cycle exists in graph.')

  return (dist_to, edge_to)
```

❶ Initialize `dist_to` dictionary to infinity for all nodes except `src`, which is 0.

❷ `edge_to[v]` records the edge terminating at `v` found during the search.

❸ Make N passes over the graph.

❹ For every edge `e = (u,v)` in the graph, use the same `relax()` concept as Dijkstra's algorithm; see if `e` improves on an existing shortest path from `src` to `v` by going through `u`.

❺ If distance to `u` added to edge `weight` to `v` is smaller than best path so far to `v`, then a shorter path has been found.

❻ Update the shortest known distance to `v`.

❼ Record the edge (`u`, `v`) that brought algorithm to `v` along the new shortest path.

❽ If `relax()` returns `True`, then a new shortest path was found to `v`.

❾ Bellman–Ford makes N passes over all E edges. If in the final pass, an edge, `e`, is found that still reduces the shortest distance from `src` to some `v`, there must be a negative cycle in the graph.

Why does this algorithm work? Observe that in a graph with N nodes, the longest possible path that can exist in the graph has no more than N – 1 edges. After the `for` loop over `i` has made N – 1 iterations attempting to relax any edge in the graph, it must have processed this potentially longest path in the graph: there should no longer be any edge left that relaxes the shortest total distance. For this reason, the `for` loop iterates N times. If on the final pass, the algorithm is able to relax an edge, then the graph must contain a negative cycle.

All-Pairs Shortest Path

The search algorithms presented in this chapter *start their search from a designated source node*. As its name suggests, the all-pairs shortest path problem asks to compute the shortest possible path of accumulated edge weights between any two nodes, *u* and *v*, in the graph. In an undirected graph, the shortest path from *u* to *v* is the same as the shortest path from *v* to *u*. In both undirected and directed graphs, node *v* may not be reachable from *u*, in which case the shortest path distance would be infinity. In a directed graph, *u* may be reachable from *v* even if *v* is not be reachable from *u*.

The search must return information to be able to recover the actual shortest paths between any *u* and *v*, but this seems incredibly challenging. For the small directed graph in Figure 7-20, it takes time to determine the shortest path between `d` and `c`, let alone all possible pairs of nodes. While there is an edge from `d` to `c` with weight 7, there is a path `d` → `b` → `a` → `c` whose total accumulated distance is 6, which is shorter.

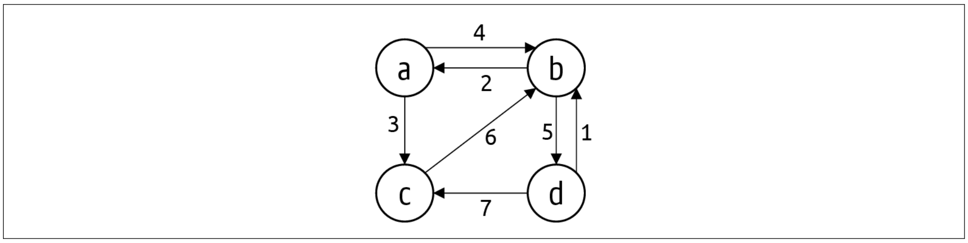


Figure 7-20. Example for all-pairs shortest path

Before delving into the details of an algorithm that solves this problem, consider what the algorithm needs to return as its result. It is similar to what you have seen for Dijkstra's algorithm and the earlier search algorithms:

- `dist_to[u][v]`—a two-dimensional structure that holds the value of the shortest path between every pair of nodes, *u* and *v*. If there is no path from *u* to *v*, `dist_to[u][v]` = infinity.

- `node_from[u][v]`—a two-dimensional structure that contains information to make it possible to compute *for any two nodes, u and v,* the actual shortest path between them.

The following insight helps lead to a solution.

Start by initializing `dist_to[u][v]` to be the weight associated with each edge *u* to *v*; if no edge exists in the graph, then set `dist_to[u][v]` to be infinity. Also initialize `node_from[u][v]` to be *u* to record that the last node on the shortest path from *u* to *v* is *u*. Figure 7-21 presents the `node_from[][]` and `dist_to[][]` after initializing these values using the graph from Figure 7-16.

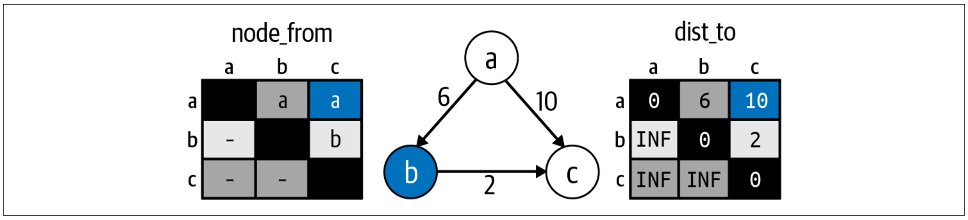


Figure 7-21. Intuition behind the all-pairs shortest path problem

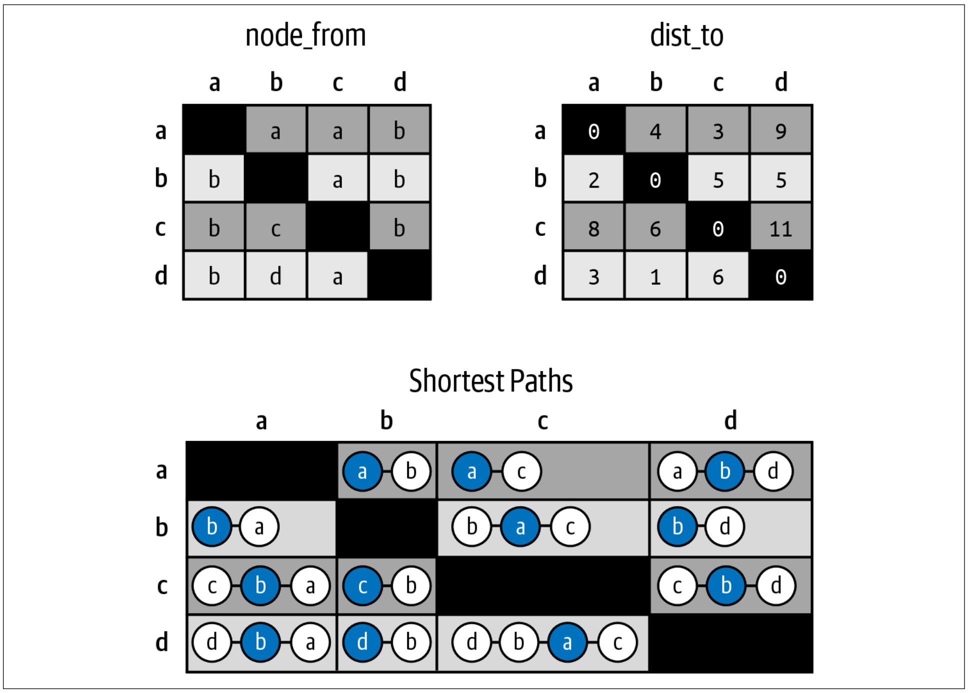
Now imagine that you select `k` = `b` and check if you can find any two nodes, *u* and *v*, where the path from *u* to `k` and then from `k` to *v* is shorter than the best known shortest computed distance of `dist[u][v]`. In this small example, `dist[a][b]` is 6, and `dist[b][c]` is 2, which means that the shortest distance from `a` to `c` now goes through `b` and is a total of 8. In addition, you can set `node_from[a][c]` to `b` to record this fact. This situation is similar to the relaxation computation that was central to Dijkstra's algorithm.

I now have a clear explanation for `node_from[u][v]`: it stores *the last node on the shortest path from node u to v*. It is similar in spirit to the `node_from[]` structures computed by earlier search algorithms, although it is more complicated.

One by one, you set `k` to each of the nodes in the graph and try to find a pair of nodes, *u* and *v*, that could reduce its shortest path distance using the logic in the previous paragraph. Once you know that `dist_to[u][k] + dist_to[k][v]` is shorter than `dist_to[u][v]`, you can update the value for `dist_to[u][v]`. You can also set `node_from[u][v]` to equal `node_from[k][v]`.

In other words, since `dist_to[k][v]` has already been computed, you know that `node_from[k][v]` is the last node on the shortest path from `k` to *v*—and since the path from *u* to *v* now *goes through* k, set `node_from[u][v]` equal to `node_from[k][v]`. To reconstruct the full path, work backward from *v* to `k`, then `node_from[u][k]` contains the former node on the shortest path all the way back to *u*.

These concepts are challenging because they are abstract—I do not compute and store each shortest path between every *u* and *v*; rather, I store partial details about the paths to compute them later. Given the graph in Figure 7-20, Figure 7-22 contains the result that an algorithm must compute. The `dist_to[][]` structure contains the computed shortest distances between any two nodes. The `dist_to[a]` row, for example, contains the computed shortest distances from `a` to all other nodes in the graph. In particular, `dist_to[a][c]` is 3 because the shortest path from `a` to `c` is along the (`a`, `c`) edge with a total distance of 3. The shortest path from `a` to `d`, `dist_to[a][d]`, is along the path `a` → `b` → `d`, whose accumulated total distance is 9.


node_from

	a	b	c	d
a		a	a	b
b	b		a	b
c	b	c		b
d	b	d	a	

dist_to

	a	b	c	d
a	0	4	3	9
b	2	0	5	5
c	8	6	0	11
d	3	1	6	0

Shortest Paths

	a	b	c	d
a		a–b	a–c	a–b–d
b	b–a		b–a–c	b–d
c	c–b–a	c–b		c–b–d
d	d–b–a	d–b	d–b–a–c	

Figure 7-22. `dist_to`, `node_from`, and actual shortest paths for graph in Figure 7-20

A solution to the all-pairs shortest path problem needs to compute `node_from[][]` and `dist_to[][]`. To explain the values in `node_from[][]`, Figure 7-22 shows the resulting shortest paths between each pair of nodes, *u* and *v*. In each of the shortest paths between *u* and *v*, the *second-to-last node in the shortest path* is highlighted. You can see that this highlighted node corresponds directly to `node_from[u][v]`.

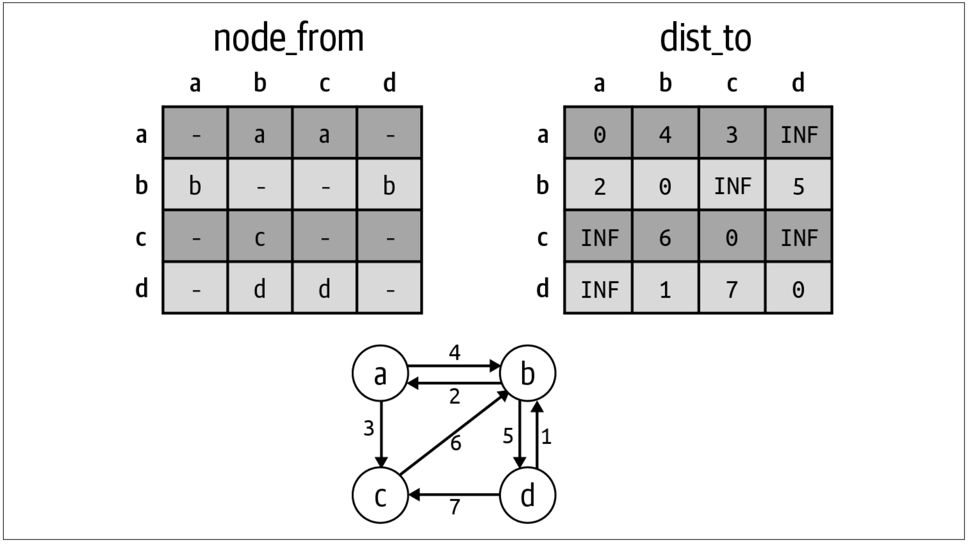
Consider the shortest path from `d` to `c`, which is the path `d` → `b` → `a` → `c`. Decompose this path into a path from `d` to `a` followed by the final edge from `a` to `c`, and you can see the *recursive solution hidden in these two-dimensional structures*. `node_from[d][c]` equals `a`, which means the last node in the shortest path from `d` to `c` is the node `a`.

Next, the shortest path from d to a goes through b, which is why node_from[d][a] equals b.

Floyd–Warshall Algorithm

Now that you are familiar with the all-pairs shortest path problem, I can present the Floyd–Warshall algorithm. The key to the algorithm is its ability to find three nodes, *u*, *v*, and k, such that there is a shorter path from *u* to *v* by going through k.

In Listing 7-17, Floyd–Warshall initializes node_from[][] and dist_to[][] using just the edge information provided in the initial graph. These initial values are depicted in Figure 7-23.



node_from

	a	b	c	d
a	-	a	a	-
b	b	-	-	b
c	-	c	-	-
d	-	d	d	-

dist_to

	a	b	c	d
a	0	4	3	INF
b	2	0	INF	5
c	INF	6	0	INF
d	INF	1	7	0

Figure 7-23. Initializing dist_to[][] and node_from[][] based on G

The entry node_from[u][v] is either None (shown as a dash) or u, to reflect the row of the entry. dist_to[u][v] is 0 whenever u is equal to v. When u and v are distinct, dist_to[u][v] is either the weight of the edge from u to v, or infinity (shown by INF) when there is no edge between u and v.

Listing 7-17. Floyd–Warshall Algorithm

```
def floyd_warshall(G):
  inf = float('inf')
  dist_to   = {}                                    ❶
  node_from = {}
  for u in G.nodes():
    dist_to[u]   = {v:inf for v in G.nodes()}       ❷
    node_from[u] = {v:None for v in G.nodes()}      ❸

    dist_to[u][u] = 0                               ❹
    for e in G.edges(u, data=True):                 ❺
      v = e[1]
      dist_to[u][v] = e[2][WEIGHT]
      node_from[u][v] = u                           ❻

  for k in G.nodes():
    for u in G.nodes():
      for v in G.nodes():
        new_len = dist_to[u][k] + dist_to[k][v]     ❼
        if new_len < dist_to[u][v]:
          dist_to[u][v] = new_len                   ❽
          node_from[u][v] = node_from[k][v]

  return (dist_to, node_from)                       ❾
```

❶ `dist_to` and `node_from` will be two-dimensional structures. Each is a dictionary containing sub-dictionaries `dist_to[u]` and `node_from[u]`.

❷ For each `u` row, initialize `dist_to[u][v]` to be infinity to reflect that each node, `v`, is initially unreachable.

❸ For each `u` row, initialize `node_from[u][v]` to be `None` to reflect that there may not even be a path from `u` to `v`.

❹ Make sure to set `dist[u][u]` = 0 to reflect that the distance from `u` to itself is 0.

❺ For each edge `e = (u,v)` from `u`, set `dist_to[u][v]` = weight of `e` to reflect that the shortest distance from `u` to `v` is the edge weight for `e`.

❻ Record that `u` is the last node on the shortest path from `u` to `v`. In fact, it is the only node on the path, which contains just the edge (`u`, `v`).

❼ Select three nodes—`k`, `u`, and `v`—and compute `new_len`, the total path length from `u` to `k` added to the path length from `k` to `v`.

❽ If `new_len` is smaller than the computed length of the shortest path from `u` to `v`, set `dist_to[u][v]` to `new_len` to record the shortest distance, and record that *the last node on the shortest path from* `u` *to* `v` is the last node on the shortest path from `k` to `v`.

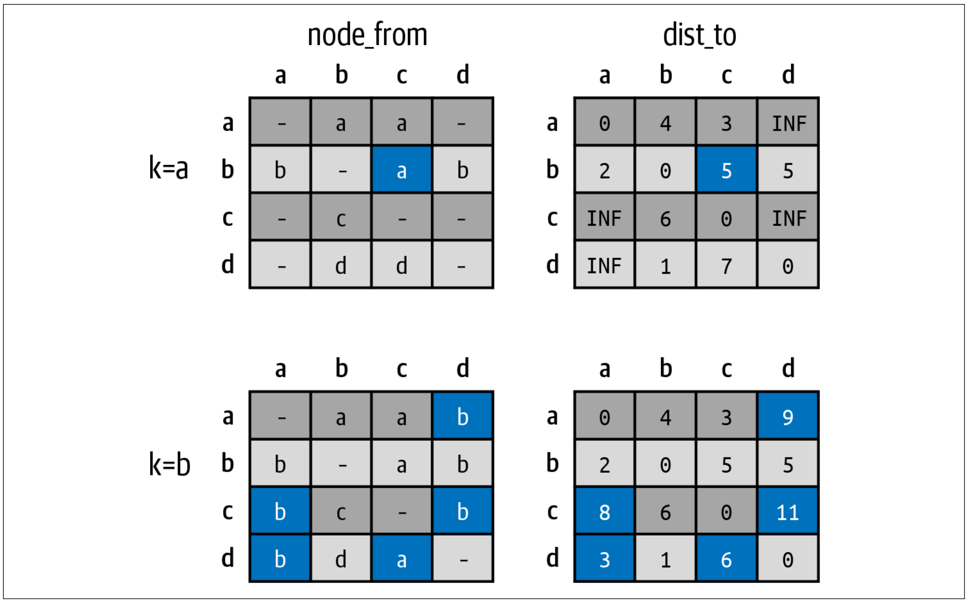
❾ Return computed `dist_to[][]` and `node_from[][]` structures so actual shortest paths can be computed for any two nodes.

Once the algorithm initializes `node_from[][]` and `dist_to[][]`, the code is quite brief. The outermost `for` loop over `k` seeks to find two nodes, *u* and *v*, such that the shortest path from *u* to *v* can be shortened by first traveling from *u* to `k` and then from `k` to *v*. As `k` explores more nodes, it eventually tries every possible improvement and ultimately computes the final correct result.

When `k` is `a`, its inner `for` loops over `u` and `v` discover that for `u = b` and `v = c`, a shorter path exists from `b` to `c` if it goes through node `a`. Observe that `dist_to[b][a]` = 2 and `dist_to[a][c]` = 3, whose total of 5 is smaller than infinity (the current computed `dist_to[b][c]` value from Figure 7-23). Not only is `dist_to[b][c]` set to 5, but also `node_from[b][c]` is set to `a` to reflect the change in the newly discovered shortest path, `b` → `a` → `c`: note that the second-to-last node on the shortest path from `b` to `c` is `a`.

Another way to explain why the algorithm works is to evaluate the values contained in `dist[u][v]`. Before the first pass through the `for` loop over `k`, `dist[u][v]` records the length of the shortest path from any `u` to any other `v` in the graph *that doesn't involve any node other than* `u` *and* `v`. After the first `k` loop iteration has ended for `k = a`, `dist[u][v]` records the length of the shortest path from any `u` to any other `v` *that can also involve* `a`. The shortest path from `b` to `c` is `b` → `a` → `c`.

When `k` is `b` during its second pass through the `for` loop, Floyd–Warshall finds five different pairs of nodes `u` and `v` for which the path from `u` to `v` is shorter if the path goes through `b`. For example, the shortest path from `d` to `c` had been 7 because of edge (`d`, `c`) in the graph. Now, however, the algorithm finds a shorter path that involves `b`, specifically traveling from `d` to `b` (with distance 1) and then traveling from `b` to `c` (with distance 5) for a shorter accumulated path of length 6. The final values in `dist[][]` and `node_from[][]` are shown in Figure 7-24.



k=a

node_from

	a	b	c	d
a	-	a	a	-
b	b	-	a	b
c	-	c	-	-
d	-	d	d	-

dist_to

	a	b	c	d
a	0	4	3	INF
b	2	0	5	5
c	INF	6	0	INF
d	INF	1	7	0

k=b

node_from

	a	b	c	d
a	-	a	a	b
b	b	-	a	b
c	b	c	-	b
d	b	d	a	-

dist_to

	a	b	c	d
a	0	4	3	9
b	2	0	5	5
c	8	6	0	11
d	3	1	6	0

Figure 7-24. Changes to `node_from[][]` and `dist_to[][]` after k processes a and b

You might be surprised that there is no check in Floyd–Warshall to make sure that `k`, `u`, and `v` are distinct nodes. You don't need one, because `dist_to[u][u]` is initialized to 0. In addition, it would only complicate the code and add unnecessary logic checks.

This algorithm surprisingly uses no advanced data structures but methodically checks all N^3 total (`k`, `u`, `v`) nodes:

- When Floyd–Warshall initializes `node_from[][]` and `dist_to[][]`, it computes all shortest paths between any *u* and *v* that involve just a single edge.
- After the first pass through `k`, the algorithm has computed all shortest paths between any *u* and *v* that involve up to two edges, limited to *u*, *v*, and node `a`.
- After the second pass through `k`, it has computed all shortest paths between any *u* and *v* that involve *up to three edges*, limited to *u*, *v*, and nodes `a` *and* `b`.

Once the outer `for` loop over `k` has completed processing N nodes, Floyd–Warshall has computed the shortest paths between any *u* and *v* that involve *up to N + 1 edges* and involve any node in the graph. Now, since a path over N nodes can only have N – 1 edges, this means that Floyd–Warshall correctly computes the distance for the shortest paths over all *u* and *v* in the graph.

The code to recover the actual shortest path is shown in Listing 7-18. This code is nearly identical to Listing 7-4, except now it processes a two-dimensional `node_from[][]` structure.

Listing 7-18. Code to recover the shortest path as computed by Floyd–Warshall

```
def all_pairs_path_to(node_from, src, target):     ❶
  if node_from[src][target] is None:
    raise ValueError('Unreachable')                 ❼

  path = []
  v = target                                        ❷
  while v != src:
    path.append(v)                                  ❸
    v = node_from[src][v]                           ❹

  path.append(src)                                  ❺
  path.reverse()                                    ❻
  return path
```

❶ `node_from[][]` structure is needed to recover path from `src` to any `target`.

❷ To recover the full path, start at `target`.

❸ As long as `v` is not `src`, append `v` to `path`, a backward list of nodes found on path from `src` to `target`.

❹ Set `v` to become the prior node in the search as recorded by `node_from[src][v]`.

❺ Once `src` is encountered, the `while` loop terminates, so `src` must be appended to complete the backward `path`.

❻ Reverse the list so all nodes appear in proper order from `src` to `target`.

❼ If `node_from[target]` is `None`, `target` is not reachable from `src`.

Summary

Graphs can model a variety of application domains, ranging from geographic data to bioinformatic information to social networks. The edges of a graph can be directed or undirected, and the edges may store numeric weights. Given a graph, there are many interesting questions that naturally arise:

- Is the graph connected? Apply Depth First Search and see if every node in the graph was visited.
- Does a directed graph contain a cycle? Apply Depth First Search and maintain extra state while searching to detect if a cycle exists.
- Given two nodes, *u* and *v*, in a graph, what is the shortest path from *u* to *v* in terms of the number of edges involved? Apply Breadth First Search to compute a solution.
- Given a weighted graph and starting node, *s*, what is the shortest path from *s* to every other node, *v*, in the graph in terms of accumulated weights of the edges? Apply Dijkstra's algorithm to compute these distances and an `edge_to[]` structure that can be used to recover the actual paths from *s* to any reachable node, *v*.
- If a graph contains negative edge weights—but has no negative cycles—is it still possible to determine the shortest path between a starting node, *s*, and every other node, *u*, in terms of accumulated weights of the edges? Apply Bellman–Ford.
- Given a weighted graph, what is the shortest path between any two nodes, *u* and *v*, in terms of accumulated weights of the edges? Apply Floyd–Warshall to compute both the distances and a `node_from[][]` structure that can be used to recover the actual paths.

When working with graphs, do not implement your own data structures to represent these graphs: better to use an existing third-party library, such as NetworkX, so you can benefit immediately from the many algorithms that it provides.

Challenge Exercises

1. Depth First Search can be coded recursively. However, doing so has a weakness when searching large graphs, because Python imposes a recursion limit of around 1,000. Still, for small mazes, you can modify the `search` to use a recursive search. Modify the skeleton code in Listing 7-19 to recursively invoke Depth First Search. Instead of using a stack to store marked nodes to be removed and processed, only invoke `dfs()` on marked nodes, and let the recursion unwind to find paths not chosen.

Listing 7-19. Complete recursive implementation for Depth First Search

```
def dfs_search_recursive(G, src):
  marked = {}
  node_from = {}

  def dfs(v):
    """Fill in this recursive function."""

  dfs(src)
  return node_from
```

2. The `path_to()` function to compute the path for Breadth First Search and Depth First Search can be implemented recursively. Implement `path_to_recursive(node_from, src, target)` as a Python generator that yields the nodes in order from `src` to `target`.
3. Design a `recover_cycle(G)` function that detects when a cycle exists *and returns the cycle.*
4. Design a `recover_negative_cycle(G)` function to augment Bellman–Ford by creating a custom `NegativeCycleError` class extending `RuntimeError` that stores the negative cycle that was discovered in the graph. Start with the offending edge that was relaxed, and try to find a cycle including this edge.
5. Construct a sample directed, weighted graph with N = 5 nodes that requires 4 iterations by Bellman–Ford to properly compute the shortest path from a designated source node. For simplicity, assign each edge a weight of 1. As a hint, it depends on the way that the edges are added to the graph. Specifically, Bellman–Ford processes all edges in order based on how `G.edges()` returns the edges.
6. For randomly constructed N × N mazes, compute the efficiency of Depth First, Breadth First, and Guided Search in reaching the designated target. Do this by revising each search algorithm to (a) stop when it reaches the target, and (b) report the total number of nodes in the `marked` dictionary.

   For N equal to powers of 2 ranging from 4 to 128, generate 512 random graphs and compute the average number of marked nodes for each search technique. You should be able to demonstrate that Guided Search is the most efficient, while Breadth First Search is the least efficient.

   Now construct a worst case problem instance for Guided Search that forces it to work almost as hard as Breadth First Search. The sample 15 × 15 maze in Figure 7-25 contains walls that form a "U" shape that blocks the path to the exit. Guided Search will have to explore this entire inner space of $(N-2)^2$ cells before "spilling over" one of the edges to find the roundabout path to the exit. The `initialize()` method in `Maze` will be helpful; you will have to manually remove south and east walls to create this shape.

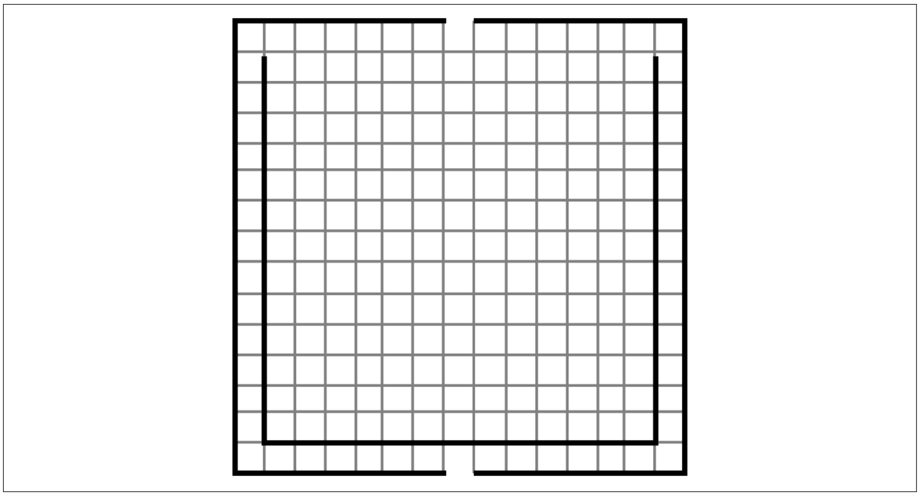


Figure 7-25. Worst case maze for Guided Search

7. A directed graph, DG, with no cycles is called a *directed acyclic graph*, or DAG for short. Dijkstra's algorithm in the *worst case* is classified as O((E+N) `log` N), but for a DAG you can compute the single-source, shortest path in O(E+N). First, apply Topological Sort to produce a linear order of the nodes. Second, process each node, *n*, in linear order, relaxing the edges that emanate from *n*. There is no need to use a priority queue. Confirm runtime behavior on random *mesh* graphs where each edge has a weight of 1. In the mesh graph in Figure 7-26, the shortest distance from node 1 to node 16 is 6.

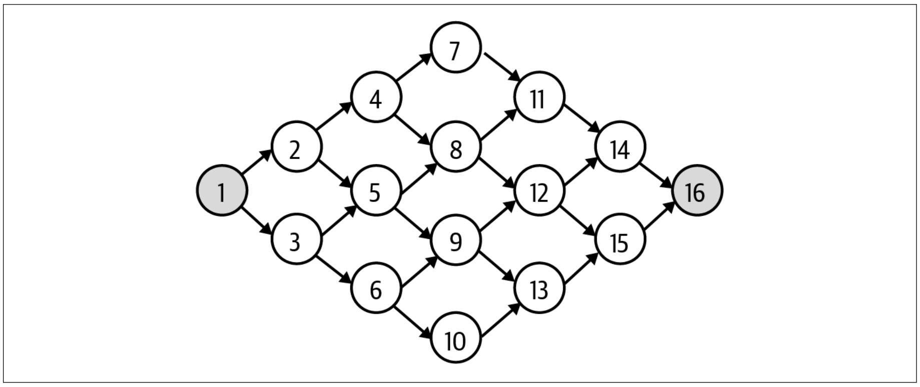


Figure 7-26. A directed, acyclic graph for single-source, shortest path optimization

8. Some drivers prefer to avoid toll roads, such as I-90 in Massachusetts. Given the graph constructed for Massachusetts highways, an edge (*u*, *v*) is part of I-90 if the label for both *u* and *v* contains `'I-90'`. Of the original 2,826 edges, 51 edges are part of I-90: remove these edges from the graph and compute

the shortest distance from the westernmost point in Massachusetts to downtown Boston, whose label (as circled in Figure 7-14) is the string `I-90@134&I-93@20&MA3@20(93)&US1@I-93(20)` representing where six highways converge. With no restrictions, the trip requires 72 edges and a total distance of 136.2 miles. However, if you choose to avoid I-90, the trip requires 104 edges and a total distance of 139.5 miles. Write code to produce these results and output an image file showing the altered route.