Listing 2.12 maze.py continued

```
class Maze:
    def __init__(self, rows: int = 10, columns: int = 10, sparseness: float =
     0.2, start: MazeLocation = MazeLocation(0, 0), goal: MazeLocation =
     MazeLocation(9, 9)) -> None:
        # initialize basic instance variables
        self._rows: int = rows
        self._columns: int = columns
        self.start: MazeLocation = start
        self.goal: MazeLocation = goal
        # fill the grid with empty cells
        self._grid: List[List[Cell]] = [[Cell.EMPTY for c in range(columns)]
     for r in range(rows)]
        # populate the grid with blocked cells
        self._randomly_fill(rows, columns, sparseness)
        # fill the start and goal locations in
        self._grid[start.row][start.column] = Cell.START
        self._grid[goal.row][goal.column] = Cell.GOAL

    def _randomly_fill(self, rows: int, columns: int, sparseness: float):
        for row in range(rows):
            for column in range(columns):
                if random.uniform(0, 1.0) < sparseness:
                    self._grid[row][column] = Cell.BLOCKED
```

Now that we have a maze, we also want a way to print it succinctly to the console. We want its characters to be close together so it looks like a real maze.

Listing 2.13 maze.py continued

```
# return a nicely formatted version of the maze for printing
def __str__(self) -> str:
    output: str = ""
    for row in self._grid:
        output += "".join([c.value for c in row]) + "\n"
    return output
```

Go ahead and test these maze functions.

```
maze: Maze = Maze()
print(maze)
```

2.2.2 Miscellaneous maze minutiae

It will be handy later to have a function that checks whether we have reached our goal during the search. In other words, we want to check whether a particular `MazeLocation` that the search has reached is the goal. We can add a method to `Maze`.

Listing 2.14 maze.py continued

```
def goal_test(self, ml: MazeLocation) -> bool:
    return ml == self.goal
```

How can we move within our mazes? Let's say that we can move horizontally and vertically one space at a time from a given space in the maze. Using these criteria, a `successors()` function can find the possible next locations from a given `MazeLocation`. However, the `successors()` function will differ for every `Maze` because every `Maze` has a different size and set of walls. Therefore, we will define it as a method on `Maze`.

Listing 2.15 maze.py continued

```
def successors(self, ml: MazeLocation) -> List[MazeLocation]:
    locations: List[MazeLocation] = []
    if ml.row + 1 < self._rows and self._grid[ml.row + 1][ml.column] !=
     Cell.BLOCKED:
        locations.append(MazeLocation(ml.row + 1, ml.column))
    if ml.row - 1 >= 0 and self._grid[ml.row - 1][ml.column] != Cell.BLOCKED:
        locations.append(MazeLocation(ml.row - 1, ml.column))
    if ml.column + 1 < self._columns and self._grid[ml.row][ml.column + 1] !=
     Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column + 1))
    if ml.column - 1 >= 0 and self._grid[ml.row][ml.column - 1] !=
     Cell.BLOCKED:
        locations.append(MazeLocation(ml.row, ml.column - 1))
    return locations
```

`successors()` simply checks above, below, to the right, and to the left of a `MazeLocation` in a `Maze` to see if it can find empty spaces that can be gone to from that location. It also avoids checking locations beyond the edges of the `Maze`. It puts every possible `MazeLocation` that it finds into a list that it ultimately returns to the caller.

2.2.3 Depth-first search

A *depth-first search* (DFS) is what its name suggests: a search that goes as deeply as it can before backtracking to its last decision point if it reaches a dead end. We'll implement a generic depth-first search that can solve our maze problem. It will also be reusable for other problems. Figure 2.4 illustrates an in-progress depth-first search of a maze.

STACKS

The depth-first search algorithm relies on a data structure known as a *stack*. (If you read about stacks in chapter 1, feel free to skip this section.) A stack is a data structure that operates under the Last-In-First-Out (LIFO) principle. Imagine a stack of papers. The last paper placed on top of the stack is the first paper pulled off the stack. It is common for a stack to be implemented on top of a more primitive data structure like a list. We will implement our stack on top of Python's `list` type.

Stacks generally have at least two operations:

- `push()`—Places an item on top of the stack
- `pop()`—Removes the item from the top of the stack and returns it

We will implement both of these, as well as an `empty` property to check if the stack has any more items in it. We will add the code for the stack to the generic_search.py file that we were working with earlier in the chapter. We already have completed all of the necessary imports.

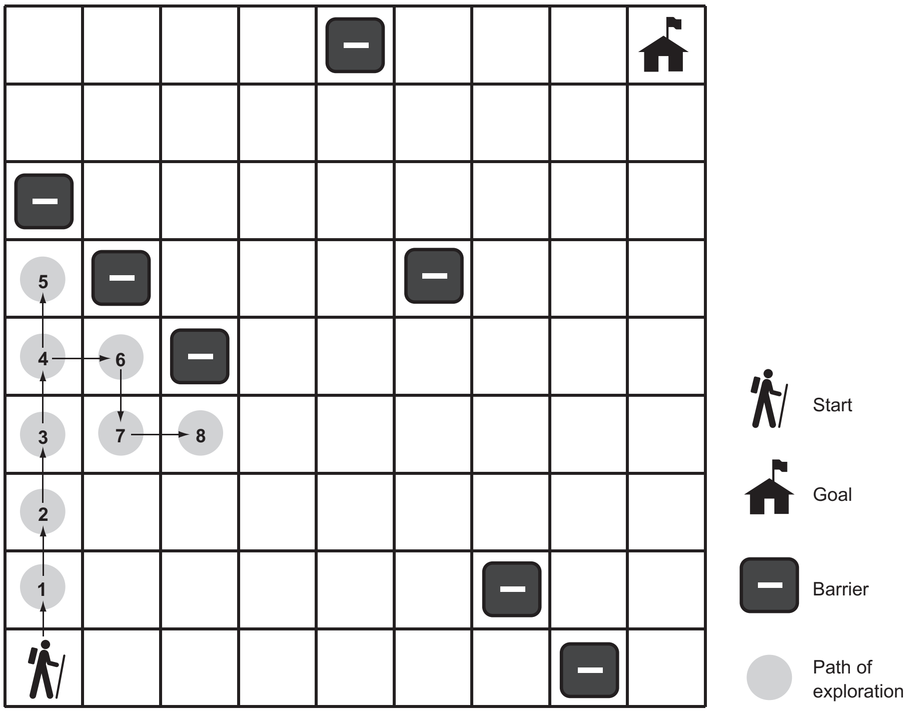

Figure 2.4 In depth-first search, the search proceeds along a continuously deeper path until it hits a barrier and must backtrack to the last decision point.

Listing 2.16 generic_search.py continued

```
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container  # not is true for empty container


    def push(self, item: T) -> None:
        self._container.append(item)


    def pop(self) -> T:
        return self._container.pop()  # LIFO


    def __repr__(self) -> str:
        return repr(self._container)
```

Note that implementing a stack using a Python `list` is as simple as always appending items onto its right end and always removing items from its extreme right end. The `pop()` method on `list` will fail if there are no longer any items in the list, so `pop()` will fail on a `Stack` if it is empty as well.

THE DFS ALGORITHM

We will need one more little tidbit before we can get to implementing DFS. We need a `Node` class that we will use to keep track of how we got from one state to another state (or from one place to another place) as we search. You can think of a `Node` as a wrapper around a state. In the case of our maze-solving problem, those states are of type `MazeLocation`. We'll call the `Node` that a state came from its `parent`. We will also define our `Node` class as having `cost` and `heuristic` properties and with `__lt__()` implemented, so we can reuse it later in the A* algorithm.

Listing 2.17 generic_search.py continued

```
class Node(Generic[T]):
    def __init__(self, state: T, parent: Optional[Node], cost: float = 0.0,
     heuristic: float = 0.0) -> None:
        self.state: T = state
        self.parent: Optional[Node] = parent
        self.cost: float = cost
        self.heuristic: float = heuristic

    def __lt__(self, other: Node) -> bool:
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

TIP The `Optional` type indicates that a value of a parameterized type may be referenced by the variable, or the variable may reference `None`.

TIP At the top of the file, the `from __future__ import annotations` allows `Node` to reference itself in the type hints of its methods. Without it, we would need to put the type hint in quotes as a string (for example, `'Node'`). In future versions of Python, importing `annotations` will be unnecessary. See PEP 563, "Postponed Evaluation of Annotations," for more information: http://mng.bz/pgzR.

An in-progress depth-first search needs to keep track of two data structures: the stack of states (or "places") that we are considering searching, which we will call the `frontier`; and the set of states that we have already searched, which we will call `explored`. As long as there are more states to visit in the frontier, DFS will keep checking whether they are the goal (if a state is the goal, DFS will stop and return it) and adding their successors to the frontier. It will also mark each state that has already been searched as explored, so that the search does not get caught in a circle, reaching states that have prior visited states as successors. If the frontier is empty, it means there is nowhere left to search.

Listing 2.18 generic_search.py continued

```
def dfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
     List[T]]) -> Optional[Node[T]]:
    # frontier is where we've yet to go
    frontier: Stack[Node[T]] = Stack()
    frontier.push(Node(initial, None))
    # explored is where we've been
    explored: Set[T] = {initial}

    # keep going while there is more to explore
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # if we found the goal, we're done
        if goal_test(current_state):
            return current_node
        # check where we can go next and haven't explored
        for child in successors(current_state):
            if child in explored:  # skip children we already explored
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None  # went through everything and never found goal
```

If `dfs()` is successful, it returns the `Node` encapsulating the goal state. The path from the start to the goal can be reconstructed by working backward from this `Node` and its priors using the `parent` property.

Listing 2.19 generic_search.py continued

```
def node_to_path(node: Node[T]) -> List[T]:
    path: List[T] = [node.state]
    # work backwards from end to front
    while node.parent is not None:
        node = node.parent
        path.append(node.state)
    path.reverse()
    return path
```

For display purposes, it will be useful to mark up the maze with the successful path, the start state, and the goal state. It will also be useful to be able to remove a path so that we can try different search algorithms on the same maze. The following two methods should be added to the `Maze` class in maze.py.

Listing 2.20 maze.py continued

```
def mark(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.PATH
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL

def clear(self, path: List[MazeLocation]):
    for maze_location in path:
        self._grid[maze_location.row][maze_location.column] = Cell.EMPTY
    self._grid[self.start.row][self.start.column] = Cell.START
    self._grid[self.goal.row][self.goal.column] = Cell.GOAL
```

It has been a long journey, but we are finally ready to solve the maze.

Listing 2.21 maze.py continued

```
if __name__ == "__main__":
    # Test DFS
    m: Maze = Maze()
    print(m)
    solution1: Optional[Node[MazeLocation]] = dfs(m.start, m.goal_test,
     m.successors)
    if solution1 is None:
        print("No solution found using depth-first search!")
    else:
        path1: List[MazeLocation] = node_to_path(solution1)
        m.mark(path1)
        print(m)
        m.clear(path1)
```

A successful solution will look something like this:

```
S****X X
 X  *****
       X*
 XX******X
  X*
  X**X
 X  *****
        *
     X  *X
        *G
```

The asterisks represent the path that our depth-first search function found from the start to the goal. Remember, because each maze is randomly generated, not every maze has a solution.

2.2.4 Breadth-first search

You may notice that the solution paths to the mazes found by depth-first traversal seem unnatural. They are usually not the shortest paths. Breadth-first search (BFS) always finds the shortest path by systematically looking one layer of nodes farther away from the start state in each iteration of the search. There are particular problems in which a depth-first search is likely to find a solution more quickly than a breadth-first search, and vice versa. Therefore, choosing between the two is sometimes a trade-off between the possibility of finding a solution quickly and the certainty of finding the shortest path to the goal (if one exists). Figure 2.5 illustrates an in-progress breadth-first search of a maze.

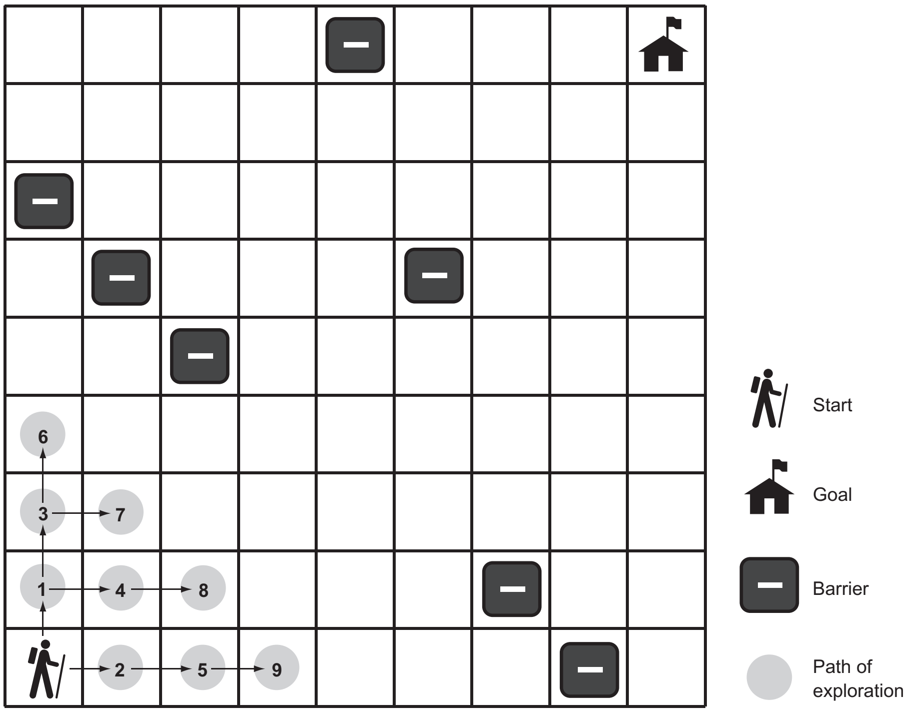

Figure 2.5 In a breadth-first search, the closest elements to the starting location are searched first.

To understand why a depth-first search sometimes returns a result faster than a breadth-first search, imagine looking for a marking on a particular layer of an onion. A searcher using a depth-first strategy may plunge a knife into the center of the onion and haphazardly examine the chunks cut out. If the marked layer happens to be near the chunk cut out, there is a chance that the searcher will find it more quickly than another searcher using a breadth-first strategy, who painstakingly peels the onion one layer at a time.

To get a better picture of why breadth-first search always finds the shortest solution path where one exists, consider trying to find the path with the fewest number of stops between Boston and New York by train. If you keep going in the same direction and backtracking when you hit a dead end (as in depth-first search), you may first find a route all the way to Seattle before it connects back to New York. However, in a breadth-first search, you will first check all of the stations one stop away from Boston. Then you will check all of the stations two stops away from Boston. Then you will check all of the stations three stops away from Boston. This will keep going until you find New York. Therefore, when you do find New York, you will know you have found the route with the fewest stops, because you already checked all of the stations that are fewer stops away from Boston, and none of them was New York.

QUEUES

To implement BFS, a data structure known as a *queue* is required. Whereas a stack is LIFO, a queue is FIFO (First-In-First-Out). A queue is like a line to use a restroom. The first person who got in line goes to the restroom first. At a minimum, a queue has the same `push()` and `pop()` methods as a stack. In fact, our implementation for `Queue` (backed by a Python `deque`) is almost identical to our implementation of `Stack`, with the only changes being the removal of elements from the left end of the `_container` instead of the right end and the switch from a `list` to a `deque`. (I use the word "left" here to mean the beginning of the backing store.) The elements on the left end are the oldest elements still in the `deque` (in terms of arrival time), so they are the first elements popped.

Listing 2.22 generic_search.py continued

```
class Queue(Generic[T]):
    def __init__(self) -> None:
        self._container: Deque[T] = Deque()

    @property
    def empty(self) -> bool:
        return not self._container  # not is true for empty container

    def push(self, item: T) -> None:
        self._container.append(item)


    def pop(self) -> T:
        return self._container.popleft()  # FIFO


    def __repr__(self) -> str:
        return repr(self._container)
```

TIP Why did the implementation of `Queue` use a `deque` as its backing store, whereas the implementation of `Stack` used a `list` as its backing store? It has to do with where we pop. In a stack, we push to the right and pop from the right. In a queue we push to the right as well, but we pop from the left. The Python `list` data structure has efficient pops from the right but not from the left. A `deque` can efficiently pop from either side. As a result, there is a built-in method on `deque` called `popleft()` but no equivalent method on `list`. You could certainly find other ways to use a `list` as the backing store for a queue, but they would be less efficient. Popping from the left on a `deque` is an $O(1)$ operation, whereas it is an $O(n)$ operation on a `list`. In the case of the `list`, after popping from the left, every subsequent element must be moved one to the left, making it inefficient.

THE BFS ALGORITHM

Amazingly, the algorithm for a breadth-first search is identical to the algorithm for a depth-first search, with the frontier changed from a stack to a queue. Changing the frontier from a stack to a queue changes the order in which states are searched and ensures that the states closest to the start state are searched first.

Listing 2.23 generic_search.py continued

```
def bfs(initial: T, goal_test: Callable[[T], bool], successors: Callable[[T],
     List[T]]) -> Optional[Node[T]]:
    # frontier is where we've yet to go
    frontier: Queue[Node[T]] = Queue()
    frontier.push(Node(initial, None))
    # explored is where we've been
    explored: Set[T] = {initial}

    # keep going while there is more to explore
    while not frontier.empty:
        current_node: Node[T] = frontier.pop()
        current_state: T = current_node.state
        # if we found the goal, we're done
        if goal_test(current_state):
            return current_node
        # check where we can go next and haven't explored
        for child in successors(current_state):
            if child in explored:  # skip children we already explored
                continue
            explored.add(child)
            frontier.push(Node(child, current_node))
    return None  # went through everything and never found goal
```

If you try running `bfs()`, you will see that it always finds the shortest solution to the maze in question. The following trial is added just after the previous one in the `if __name__ == "__main__":` section of the file, so results can be compared on the same maze.

Listing 2.24 maze.py continued

```
# Test BFS
solution2: Optional[Node[MazeLocation]] = bfs(m.start, m.goal_test,
     m.successors)
if solution2 is None:
    print("No solution found using breadth-first search!")
else:
    path2: List[MazeLocation] = node_to_path(solution2)
    m.mark(path2)
    print(m)
    m.clear(path2)
```

It is amazing that you can keep an algorithm the same and just change the data structure that it accesses and get radically different results. The following is the result of calling `bfs()` on the same maze that we earlier called `dfs()` on. Notice how the path marked by the asterisks is more direct from start to goal than in the prior example.

```
S    X X
*X
*      X
*XX      X
* X
* X  X
*X
*
*    X   X
*********G
```

2.2.5 A* search

It can be very time-consuming to peel back an onion, layer-by-layer, as a breadth-first search does. Like a BFS, an A* search aims to find the shortest path from start state to goal state. Unlike the preceding BFS implementation, an A* search uses a combination of a cost function and a heuristic function to focus its search on pathways most likely to get to the goal quickly.

The cost function, g(*n*), examines the cost to get to a particular state. In the case of our maze, this would be how many previous steps we had to go through to get to the state in question. The heuristic function, h(*n*), gives an estimate of the cost to get from the state in question to the goal state. It can be proved that if h(*n*) is an *admissible heuristic*, then the final path found will be optimal. An admissible heuristic is one that never overestimates the cost to reach the goal. On a two-dimensional plane, one example is a straight-line distance heuristic, because a straight line is always the shortest path.[1]

The total cost for any state being considered is f(*n*), which is simply the combination of g(*n*) and h(*n*). In fact, f(*n*) = g(*n*) + h(*n*). When choosing the next state to explore from the frontier, an A* search picks the one with the lowest f(*n*). This is how it distinguishes itself from BFS and DFS.

PRIORITY QUEUES

To pick the state on the frontier with the lowest f(*n*), an A* search uses a *priority queue* as the data structure for its frontier. A priority queue keeps its elements in an internal order, such that the first element popped out is always the highest-priority element. (In our case, the highest-priority item is the one with the lowest f(*n*).) Usually this means the internal use of a binary heap, which results in O(lg *n*) pushes and O(lg *n*) pops.

Python's standard library contains `heappush()` and `heappop()` functions that will take a list and maintain it as a binary heap. We can implement a priority queue by building a thin wrapper around these standard library functions. Our `PriorityQueue` class will be similar to our `Stack` and `Queue` classes, with the `push()` and `pop()` methods modified to use `heappush()` and `heappop()`.

[1] For more information on heuristics, see Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd edition (Pearson, 2010), page 94.

Listing 2.25 generic_search.py continued

```
class PriorityQueue(Generic[T]):
    def __init__(self) -> None:
        self._container: List[T] = []

    @property
    def empty(self) -> bool:
        return not self._container  # not is true for empty container


    def push(self, item: T) -> None:
        heappush(self._container, item)  # in by priority


    def pop(self) -> T:
        return heappop(self._container)  # out by priority

    def __repr__(self) -> str:
        return repr(self._container)
```

To determine the priority of a particular element versus another of its kind, `heappush()` and `heappop()`, compare them by using the < operator. This is why we needed to implement `__lt__()` on `Node` earlier. One `Node` is compared to another by looking at its respective f(*n*), which is simply the sum of the properties `cost` and `heuristic`.

Heuristics

A *heuristic* is an intuition about the way to solve a problem.[2] In the case of maze solving, a heuristic aims to choose the best maze location to search next, in the quest to get to the goal. In other words, it is an educated guess about which nodes on the frontier are closest to the goal. As was mentioned previously, if a heuristic used with an A* search produces an accurate relative result and is admissible (never overestimates the distance), then A* will deliver the shortest path. Heuristics that calculate smaller values end up leading to a search through more states, whereas heuristics closer to the exact real distance (but not over it, which would make them inadmissible) lead to a search through fewer states. Therefore, ideal heuristics come as close to the real distance as possible without ever going over it.

Euclidean distance

As we learn in geometry, the shortest path between two points is a straight line. It makes sense, then, that a straight-line heuristic will always be admissible for the maze-solving problem. The Euclidean distance, derived from the Pythagorean theorem, states that `distance = √((difference in x)`^{2} `+ (difference in y)`^{2}`)`. For our mazes, the difference in *x* is equivalent to the difference in columns between two maze locations, and the difference in *y* is equivalent to the difference in rows. Note that we are implementing this back in maze.py.

[2] For more about heuristics for A* pathfinding, check out the “Heuristics” chapter in Amit Patel’s *Amit’s Thoughts on Pathfinding*, http://mng.bz/z7O4.