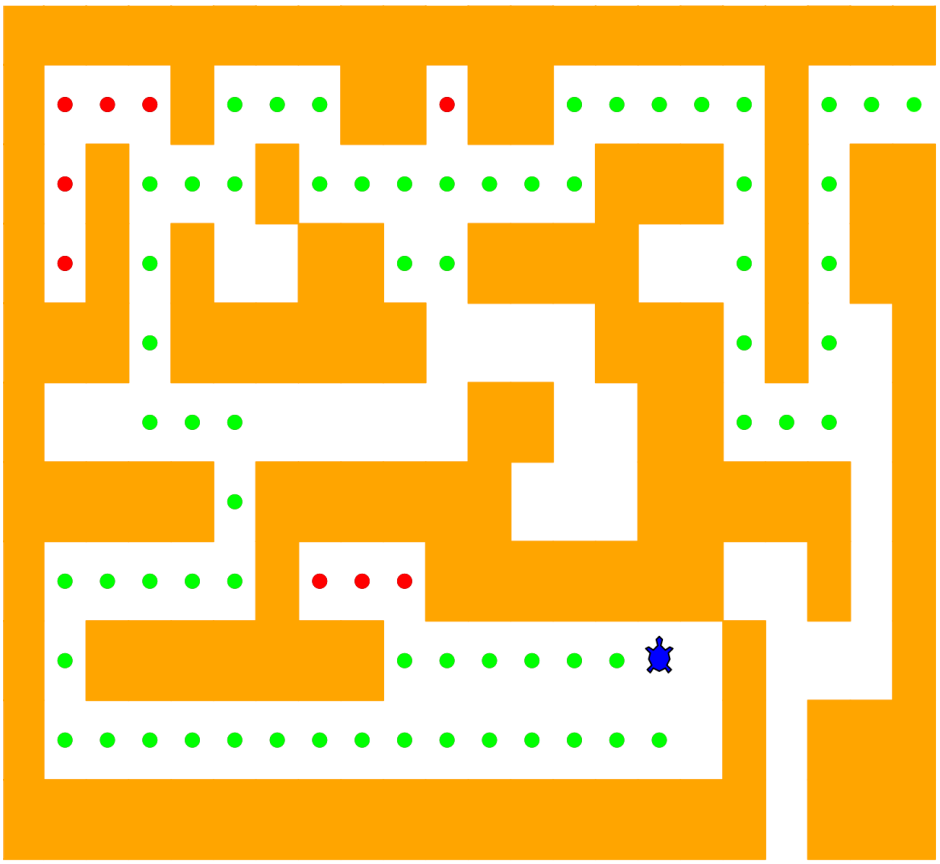
4.11. Exploring a Maze

In this section we will look at a problem that has relevance to the expanding world of robotics: how do you find your way out of a maze? If you have a Roomba vacuum cleaner for your dorm room (don't all college students?) you will wish that you could reprogram it using what you have learned in this section. The problem we want to solve is to help our turtle find its way out of a virtual maze. The maze problem has roots as deep as the Greek myth about Theseus, who was sent into a maze to kill the Minotaur. Theseus used a ball of thread to help him find his way back out again once he had finished off the beast. In our problem we will assume that our turtle is dropped down somewhere into the middle of the maze and must find its way out. Look at Figure 4.12 to get an idea of where we are going in this section.

Figure 4.12: The Finished Maze Search Program



(../_images/maze.png)

To make it easier for us we will assume that our maze is divided up into squares. Each square of the maze is either open or occupied by a section of wall. The turtle can only pass through the open squares of the maze. If the turtle bumps into a wall, it must try a different direction. The turtle will require a systematic procedure to find its way out of the maze. Here is the procedure:

1. From our starting position we will first try going north one square and then recursively try our procedure from there.
2. If we are not successful by trying a northern path as the first step then we will take a step to the south and recursively repeat our procedure.
3. If south does not work then we will try a step to the West as our first step and recursively apply our procedure.
4. If north, south, and west have not been successful then we will apply the procedure recursively from a position one step to our east.
5. If none of these directions works then there is no way to get out of the maze and we fail.

(TowerofHanoi.html) (DynamicProgram

Now that sounds pretty easy, but there are a couple of details to talk about first. Suppose we take our firs recursive step by going north. By following our procedure, our next step would also be to the north. But if the north is blocked by a wall, we must look at the next step of the procedure and try going to the south. Unfortunately, that step to the south brings us right back to our original starting place. If we apply the recursive procedure from there, we will just go back one step to the North and be in an infinite loop. So w must have a strategy to remember where we have been. In this case we will assume that we have a bag of bread crumbs we can drop along our way. If we take a step in a certain direction and find that there is a bread crumb already on that square, we know that we should immediately back up and try the next direction in our procedure. As we will see when we look at the code for this algorithm, backing up is as simple as returning from a recursive function call.

As we do for all recursive algorithms, let us review the base cases. Some of them you may already have guessed based on the description in the previous paragraph. In this algorithm, there are four base cases to consider:

1. The turtle has run into a wall. Since the square is occupied by a wall, no further exploration can take place.
2. The turtle has found a square that has already been explored. We do not want to continue exploring from this position so we don't get into a loop.
3. We have found an outside edge, not occupied by a wall. In other words, we have found an exit from the maze.
4. We have explored a square unsuccessfully in all four directions.

For our program to work we will need to have a way to represent the maze. Figure 4.13 is an example of a maze data file.

Figure 4.13: An Example Maze Data File

```
++++++++++++++++++++++
+   +   ++ ++     +
+ +   +       +++ + ++
+ + +  ++  ++++   + ++
+++ ++++++    +++ +  +
+          ++  ++    +
+++++ ++++++   +++++ +
+     +   +++++++  + +
+ +++++++      S +   +
+                + +++
++++++++++++++++++ +++
```

To make this even more interesting we are going to use the `turtle` module to draw and explore our maze so we can watch this algorithm in action. The `Maze` object will provide the following methods for us to use in writing our search algorithm:

- `__init__` Reads in a data file representing a maze, initializes the internal representation of the maze, and finds the starting position for the turtle.
- `draw_maze` Draws the maze in a window on the screen.
- `update_position` Updates the internal representation of the maze and changes the position of the turtle in the window.
- `is_exit` Checks to see if the current position is an exit from the maze.

(TowerofHanoi.html) (DynamicProgram

The `Maze` class also overloads the index operator `[]` so that our algorithm can easily access the status of any particular square.

Listing 4.11 includes the global constants used by the `Maze` class methods (Listings 4.12–4.15) and the

`search_from` function (Listing 4.16).

Listing 4.11: The Maze Program Global Constants

```
START = "S"
OBSTACLE = "+"
TRIED = "."
DEAD_END = "-"
PART_OF_PATH = "O"
```

The `__init__` method takes the name of a file as its only parameter. This file is a text file that represents a maze by using “+” characters for walls, spaces for open squares, and the letter “S” to indicate the starting position.

Listing 4.12: The Maze Class Constructor

```
class Maze:
    def __init__(self, maze_filename):
        with open(maze_filename, "r") as maze_file:
            self.maze_list = [
                [ch for ch in line.rstrip("\n")]
                for line in maze_file.readlines()
            ]
        self.rows_in_maze = len(self.maze_list)
        self.columns_in_maze = len(self.maze_list[0])
        for row_idx, row in enumerate(self.maze_list):
            if START in row:
                self.start_row = row_idx
                self.start_col = row.index(START)
                break

        self.x_translate = -self.columns_in_maze / 2
        self.y_translate = self.rows_in_maze / 2
        self.t = turtle.Turtle()
        self.t.shape("turtle")
        self.wn = turtle.Screen()
        self.wn.setworldcoordinates(
            -(self.columns_in_maze - 1) / 2 - 0.5,
            -(self.rows_in_maze - 1) / 2 - 0.5,
            (self.columns_in_maze - 1) / 2 + 0.5,
            (self.rows_in_maze - 1) / 2 + 0.5,
        )
```

The internal representation of the maze is a list of lists. Each row of the `maze_list` instance variable is also a list. This secondary list contains one character per square using the characters described above. For the data file in Figure 13 the internal representation looks like the following:

(TowerofHanoi.html) (DynamicProgram

```
[  ['+', '+', '+', '+', '+', ..., '+', '+', '+', '+', '+']
   ['+', ' ', ' ', ' ', '+', ..., ' ', '+', ' ', ' ', ' ']
   ['+', ' ', '+', ' ', ' ', ..., ' ', '+', ' ', '+', '+']
   ['+', ' ', '+', ' ', '+', ..., ' ', '+', ' ', '+', '+']
   ['+', '+', '+', ' ', '+', ..., ' ', '+', ' ', ' ', '+']
   ['+', ' ', ' ', ' ', ' ', ..., ' ', ' ', ' ', ' ', '+']
   ['+', '+', '+', '+', '+', ..., '+', '+', '+', ' ', '+']
   ['+', ' ', ' ', ' ', ' ', ..., ' ', ' ', '+', ' ', '+']
   ['+', ' ', '+', '+', '+', ..., '+', ' ', ' ', ' ', '+']
   ['+', ' ', ' ', ' ', ' ', ..., '+', ' ', '+', '+', '+']
   ['+', '+', '+', '+', '+', ..., '+', ' ', '+', '+', '+'] ]
```

The `draw_maze` method uses this internal representation to draw the initial view of the maze on the screen (Figure 4.12).

Listing 4.13: The Maze Class Drawing Methods

```
def draw_maze(self):
    self.t.speed(10)
    self.wn.tracer(0)
    for y in range(self.rows_in_maze):
        for x in range(self.columns_in_maze):
            if self.maze_list[y][x] == OBSTACLE:
                self.draw_centered_box(
                    x + self.x_translate, -y + self.y_translate, "orange"
                )
    self.t.color("black")
    self.t.fillcolor("blue")
    self.wn.update()
    self.wn.tracer(1)

def draw_centered_box(self, x, y, color):
    self.t.up()
    self.t.goto(x - 0.5, y - 0.5)
    self.t.color(color)
    self.t.fillcolor(color)
    self.t.setheading(90)
    self.t.down()
    self.t.begin_fill()
    for i in range(4):
        self.t.forward(1)
        self.t.right(90)
    self.t.end_fill()
```

The `update_position` method, as shown in Listing 4.14 uses the same internal representation to see if the turtle has run into a wall. It also updates the internal representation with a "." or "-" to indicate that the turtle has visited a particular square or if the square is part of a dead end. In addition, the `update_position` method uses two helper methods, `move_turtle` and `drop_bread_crumb` , to update the view on the screen.

(TowerofHanoi.html) (DynamicProgram

Listing 4.14: The Maze Class Moving Methods

```
def update_position(self, row, col, val=None):
    if val:
        self.maze_list[row][col] = val
    self.move_turtle(col, row)

    if val == PART_OF_PATH:
        color = "green"
    elif val == OBSTACLE:
        color = "red"
    elif val == TRIED:
        color = "black"
    elif val == DEAD_END:
        color = "red"
    else:
        color = None

    if color:
        self.drop_bread_crumb(color)

def move_turtle(self, x, y):
    self.t.up()
    self.t.setheading(self.t.towards(x + self.x_translate, -y + self.y_translate))
    self.t.goto(x + self.x_translate, -y + self.y_translate)

def drop_bread_crumb(self, color):
    self.t.dot(10, color)
```

Finally, the `is_exit` method uses the current position of the turtle to test for an exit condition. An exit condition occurs whenever the turtle has navigated to the edge of the maze, either row zero or column zero, or the far-right column or the bottom row.

Listing 4.15: The Maze Class Auxiliary Methods

```
def is_exit(self, row, col):
    return (
        row == 0
        or row == self.rows_in_maze - 1
        or col == 0
        or col == self.columns_in_maze - 1
    )

def __getitem__(self, idx):
    return self.maze_list[idx]
```

Let's examine the code for the search function which we call `search_from` . The code is shown in Listing 4.16. Notice that this function takes three parameters: a `Maze` object, the starting row, and the starting column. This is important because as a recursive function the search logically starts again with each recursive call.

Listing 4.16: The Maze Search Function

```
1 def search_from(maze, row, column):
2     # Try each of four directions from this point until we find a way out.
3     maze.update_position(row, column)
4     # Base Case return values:
5     #  1. We have run into an obstacle, return false
6     if maze[row][column] == OBSTACLE:
7         return False
8     #  2. We have found an already explored square
9     if maze[row][column] in [TRIED, DEAD_END]:
10        return False
11    # 3. We have found an exit
12    if maze.is_exit(row, column):
13        maze.update_position(row, column, PART_OF_PATH)
14        return True
15    maze.update_position(row, column, TRIED)
16    # Otherwise, use logical short circuiting to try each direction
17    # in turn (if needed)
18    found = (
19        search_from(maze, row - 1, column)
20        or search_from(maze, row + 1, column)
21        or search_from(maze, row, column - 1)
22        or search_from(maze, row, column + 1)
23    )
24    if found:
25        maze.update_position(row, column, PART_OF_PATH)
26    else:
27        maze.update_position(row, column, DEAD_END)
28    return found
```

As you look through the algorithm you will see that the first thing the code does (line 3) is call `update_position`. This is simply to help you visualize the algorithm so that you can watch exactly how the turtle explores its way through the maze. Next the algorithm checks for the first three of the four base cases: Has the turtle run into a wall (lines 6)? Has the turtle circled back to a square already explored (line 9)? Has the turtle found an exit (line 12)? If none of these conditions is true then we continue the search recursively.

You will notice that in the recursive step there are four recursive calls to `search_from`. It is hard to predict how many of these recursive calls will be used since they are all connected by `or` statements. If the first call to `search_from` returns `True` then none of the last three calls would be needed. You can interpret this as meaning that a step to `(row - 1, column)` (or north if you want to think geographically) is on the path leading out of the maze. If there is not a good path leading out of the maze to the north then the next recursive call is tried, this one to the south. If south fails then try west, and finally east. If all four recursive calls return `False` then we have found a dead end. You should download or type in the whole program and experiment with it by changing the order of these calls.

The complete program is shown in ActiveCode 4.11.1. This program uses the data file `maze2.txt` shown below. Note that it is a much more simple example file in that the exit is very close to the starting position of the turtle.

(TowerofHanoi.html) (DynamicProgram

```
++++++++++++++++++++++
+   +   ++ ++        +
      +     ++++++++++
+ +    ++  ++++ +++ ++
+ +   + + ++    +++  +
+          ++  ++  + +
+++++ + +      ++  + +
+++++ +++  + +  ++   +
+          + + S+ +  +
+++++ +  + + +     + +
++++++++++++++++++++++
```

Save & Run

Original - 1 of 1

Pair?

(TowerofHanoi.html)

(DynamicProgram

Activity: 4.11.1 Complete Maze Solver (completemaze)

Self Check

Modify the maze search program so that the calls to `search_from` are in a different order. Watch the program run. Can you explain why the behavior is different? Can you predict what path the turtle will follow for a given change in order?

You have attempted 1 of 2 activities on this page

user not logged in

(TowerofHanoi.html) (DynamicProgram

(TowerofHanoi.html)

(DynamicProgram

(TowerofHanoi.html)

(DynamicProgram