50.004 Week 6 In-Class Exercise

Prof. Ioannis Panageas and Jing Yu Koh
(Code adapted from Prof. Alexander Binder)

October 15, 2018

Abstract

In this in class exercise, you will be implementing the BFS and DFS algorithms to solve a maze.

1 Dependencies

In order to run the code in this exercise, you will need several Python libraries installed. Please install numpy and scipy. They are important libraries for numerical and scientific computing that you will definitely use again in ISTD.

All code provided has only been tested on Python 3.

2 Starter Code

Download the starter code from the 50.004 Dropbox.

The code to create mazes has been written for you. It is not very important to understand the maze helper code for the purposes of 50.004, and you can treat it as a black box (For your own sake. The code is not very nice.)

There are two files that you will need to edit: *maze_creator.py* and *maze_solver.py*.

2.1 Generating Mazes

An interesting application of DFS is that it can be used to generate mazes. This approach is easily implemented with a stack, and is one of the simplest ways to generate mazes. A pseudocode of this algorithm is provided below:

1. Initialize the maze to start with all walls, except some initial cell, such as (0, 0). Mark this cell as visited.
2. While there are unvisited cells in the maze:
    2.1. If the current cell has any neighbors (cells that are 2 tiles away in any of the four cardinal directions - not adjacent cells!) that are not visited:
        i. Choose one of the neighbors at random.

ii. Push current cell to the stack.
iii. Remove the wall between the current cell and the chosen cell.
iv. Make the chosen cell the current cell, mark it as visited.

2.2. Else, if the stack is not empty:

i. Pop a cell from the stack.
ii. Set this cell as the current cell. Repeat step (2).

Open *maze_creator_skeleton.py*. Around line 38, you will see a bunch of comments providing more hints. Implement DFS as described above to create a maze. Note the following conventions when implementing the code:

- The end of the maze is denoted with 'E'. So if $x = 2, y = 2$ is the end point:

  ```
  maze[2][2] = 'E'
  ```

- Similarly, the start of the maze is denoted with 'B'.
- Any open paths are denoted with 'O'.
- Walls are denoted with 'W'. The initial maze is an $n \times n$ matrix of walls, so you have to replace certain tiles with the appropriate character to form your maze.

2.1.1 Actually Generating Mazes

After you've completed *maze_creator.py* successfully, you can use it to create cool looking mazes. For example, if you wish to create a 20x20 tile maze, and save it as "maze1.png", run the following code:

```
python maze_creator_skeleton.py 20 20 maze1.png
```

You can also choose to save maze files as .txt files:

```
python maze_creator_skeleton.py 20 20 maze1.txt
```

The two formats don't really matter much, but .png would allow you to better visualize it. You'll see a maze in your directory that looks something like this:

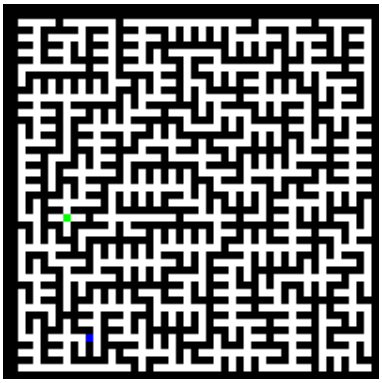

Figure 1: A 100x100 Maze!

2.2 Solving Mazes

Next, you'll be implementing an algorithm to find a solution to the mazes you previously generated. Open *maze_solver_skeleton.py*. Around line 53, you will find a section to implement BFS and DFS search. Implement the search accordingly, and return two lists:

- *path*: Containing the coordinates (in sequence) of the shortest path from the start to the exit.
- *expanded*: Contains the coordinates of all tiles visited. This does not have to be in sequence. All values in *path* should also be in *expanded*.

The maze data will be represented as outlined in section 2.1.

2.2.1 Actually Solving Mazes

After completing the skeleton code, test it on the maze you generated previously:

```
python maze_solver_skeleton.py maze1.png maze1_solution.png BFS
```

The resulting file, maze1_solution.png, will contain the route found using BFS or DFS. The pink regions represent the tiles that were searched, while the red path represents the found path from the start point to the end point of the maze. It should look something like the following:

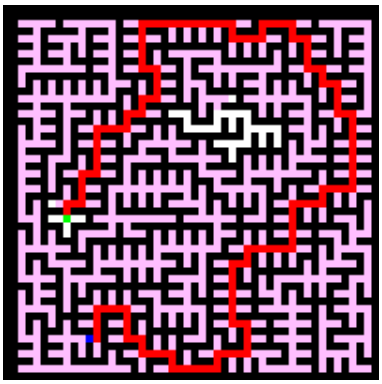

Figure 2: E s c a p e

3 Food for Thought

- Does solving the maze with DFS and BFS give you the exact same solution? If not, which is a better choice?
- Is the solution found the shortest path from the entrance to the exit?
- The default implementation of BFS has a complexity of $O(V+E)$. For this particular problem, is there a way to improve the run time? If so, what is the new complexity?