EN.540.635 "Software Carpentry"

Weekly Challenge 7 - Maze Generation and Solving

To give us some practice with some common data structures in Python, this assignment will build off of the previous week's lab exercise on maze generation and maze solving. Maze generation is the act of designing the layout of passages and walls within a maze. There are many different approaches to generating mazes, with various maze generation algorithms for building them. A simple Google search will yield many different kinds of algorithms (some of which are detailed on Wikipedia). While many of these algorithms can be quite complex, one of the simplest ones is the *depth-first search* algorithm, which is what we will be focusing on in this assignment. This algorithm will actually work for both the maze generation and the solving! For the maze generation, mazes are to be generated as png images using the Python Imaging Library (PIL). For the maze solving, mazes are similarly solved by reading in a png image of a generated maze, and outputing a new image of the maze with the solution marked out in green.

The Depth First Search Algorithm

This method has us starting with a stack, which we can call "positions". It will be initialized at some coordinate to signify the start of the maze. As such, we can initialize this stack and have it contain our starting coordinate:

```
start = (0, 0)
positions = [start]
```

Then, we will look for a valid position to take, and randomly select it. Two possibilities now exist: (1) we have either found valid positions and then we take a random step, or (2) there are no valid options that exist. If (1), then we simply append the new coordinate to the positions stack, adjust variables accordingly, and repeat the process of looking for a valid position. If (2), then we pop the last position out of the stack, and continue from where we previously were (seeking out a new step to take). Once the entire space of possible choices has been explored, it should be evident that the backtracking will continue until the positions stack is empty. It is at this point that the algorithm will end. The same algorithm can be used in both the maze generation and the maze solving - **the main differences in these two functions will be the criteria for assessing if a step is valid or not.** This YouTube video shows a visualization of a maze being *generated* using this algorithm - in this video, the light blue paths are positions added to the stack and white paths are positions that have been removed from the stack. This YouTube video shows a visualization of a maze being *solved* using this algorithm - in this video, the green paths are positions added to the stack and gray paths are positions that have been removed from the stack.

The Code Framework

If you download the code framework that comes with this assignment handout, you will see that it has several different functions already filled in. The first function is *get_colors()*, which returns a dictionary of integers that correspond to different RGB tuples that represent different types of spaces we have in our maze:

- 0 - Black - A wall
- 1 - White - A space to travel in the maze
- 2 - Green - A valid solution of the maze
- 3 - Red - A backtracked position during maze solving
- 4 - Blue - Start and endpoints of the maze

We can easily represent our maze using a list of lists containing the appropriate integers as described above - it is best to think of this list of lists as a matrix, where the indices of a given element in the matrix correspond to positions in the maze. The second function is *save_maze()*, which will take our maze matrix and save it as a png image. The third function is *load_maze()*, which will take a png image of the maze and return the maze matrix. We have also included a fourth function *pos_chk()*, which will return a Boolean depending on the x and y coordinates that are input and if they are in the boundaries

of the maze or not; this function will be useful in our code for generating and solving mazes.

The remaining two functions for generating and solving the maze need to be completed for this assignment. All the relevant input parameters are detailed in the docstrings that have been provided. The *slow* parameter corresponds to whether you save the maze after every step has been taken or not - this allows to see the maze generation or solving occur in real time (similar to the YouTube videos linked above). Example mazes are shown below in Figures 1 and 2, respectively.

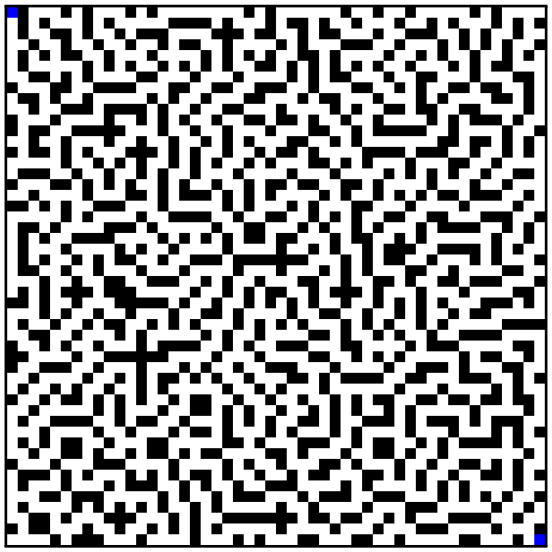

Figure 1: The image of a generated maze. The white spaces correspond to paths in the maze and the black spaces correspond to walls. Note: No blue color should be added upon maze generation. The blue color, which represents starting and ending points, should be added in the solve maze function. Further, they should be allowed to be any valid position.

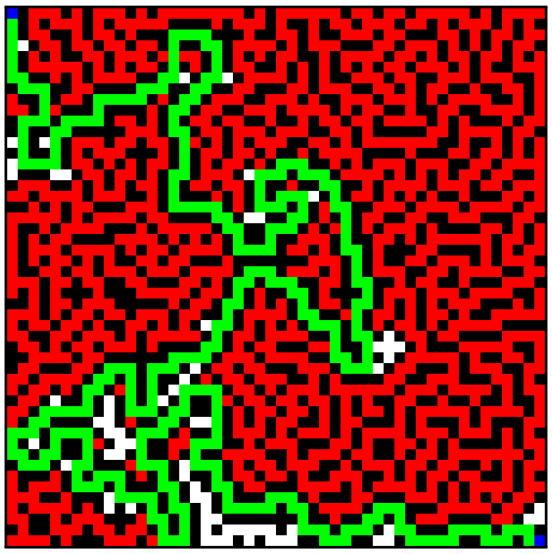

Figure 2: The image of the solved maze from Figure 1. The green spaces show the path from the start to the end point and the red spaces correspond to paths traveled by the algorithm that resulted in dead ends.

Please upload your code with completed functions for generating and solving a maze to Blackboard under the appropriate assignment. The mazes you generate and solve should look similar in format to what is shown in Figures 1 and 2. Also, keep code formatting and comments in mind.