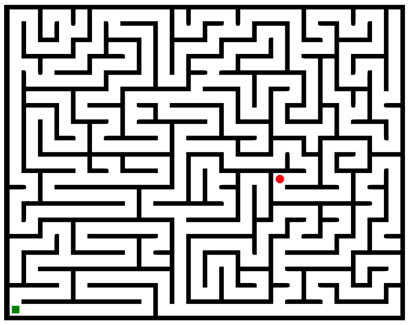
CS2204 Homework: Maze Building and Path Finding



Objectives

- Gain intuition with recursive algorithms
- Better understand depth-first search and backtracking
- Adapting similar solutions and/or abstract algorithms to an executable solution

Background

This homework assignment needs a bit more research and thinking than actual coding. The goal is to develop a recursive maze generator function and a recursive path finder algorithm. Your maze generator code should implement the Recursive Backtracker (https://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_depth-first_search) algorithm to generate a perfect random maze (in a perfect maze there is always one and only one path between any two locations). The path finder function should also implement a recursive (depth-first) search algorithm (https://en.wikipedia.org/wiki/Maze_solving_algorithm#Recursive_algorithm). Your initial assignment is to understand the problem and these two methods. Note, while I suggest to follow these algorithms, you are allowed to solve the problem with any viable algorithm. You can also search for existing Python code (except those developed by your classmates for this assignment).

Tasks

You need to finish the `maze.py` source file. It already contains significant infrastructure code. There is also a built-in testing and maze visualizing logic, so you can run `maze.py` at any time and see your current progress (there is no separate `validator.py` for this assignment). You do not need to understand and should not modify the testing code.

The maze is represented by a rectangular (2-dimensional) array of cells - each with four potential walls on their sides. The `maze.py` file contains the definition of the `Cell` class and three global variables for the size of the maze (`size_x` , `size_y`) and the 2-dimensional array (using a list of lists) of the maze itself (`maze`). Your code can access these global objects at any time.

The two important methods of the `Cell` objects are: `remove_wall(` *direction* `)` and `has_wall(` *direction* `)`. For both methods, the *direction* parameter is a single letter (string) to identify the side of the cell (`"N"` for North, `"E"` for East, `"S"` for South and `"W"` for West).

```
if maze[1][1].has_wall("E"):
    maze[1][1].remove_wall("E")
```

This code removes the east-side wall of the cell at location `x = 1 , y = 1` if it currently has one. Note, that the west-side wall of the adjacent cell (x=2, y=1) will be removed automatically by the built-in code. Also, the **y coordinate decreases towards the North direction** while the **x coordinate decreases towards the West direction**:

`(x - 1, y - 1)`	`(x, y - 1)`	`(x + 1, y - 1)`
`(x - 1, y)`	`(x, y)`	`(x + 1, y)`
`(x - 1, y + 1)`	`(x, y + 1)`	`(x + 1, y + 1)`

You task is to implement the `build_maze` (50 pts) and `find_path` (50 pts) functions. The first function should iteratively remove walls of the Cell objects in the maze array to generate a perfect random maze. Initially, all cells have all their walls standing. See the docstring for additional details.

The `find_path` function is called with two parameters, identifying the *start* and *end* positions. Both parameters are pairs (tuples) of `x` , `y` coordinates. Your task is to return a valid path from the start to the end position in the current maze (that you generated in the previous step). This path should be a sequence of pairs (tuples of two integers), where each pair is an `x` , `y` location along the path. Only horizontal and vertical steps can be made and the path cannot cross a wall. The first element of a valid path sequence is the *start* position, while the last element is the *end* location.

Hints

- The `build_maze` and `find_path` functions - especially their parameters are not well suited for a recursive algorithm. I suggest to write your own recursive function using whatever parameters you need for recursion and call your custom function from `build_maze` and `find_path` with good initial arguments.
- You can extend the `Cell` class with new attributes or methods if you wish (for example, a `visited` attribute). However, do not change the testing code at the end of the script marked by a long line of comment characters.
- You can create additional global variables if you wish (e.g. if you want to keep track of the visited cells).
- If you find yourself in a situation where you create a new `Cell()` instance, you are on the wrong path. All cell instances are created for you and stored in the global `maze` array (list of lists)

Grading

Running the `maze.py` program should give you the estimated score and some feedback on the potential problems with your solution. It also shows the state of your maze and the path you found.

Penalties

Please, set the `__author__` variable to your *VUnetID* and try to follow the PEP 8 coding style recommendations. For this assignment we are not checking nor penalizing these issues.

Submission

Please, upload the final version of the following file(s) (**and only those files**) to Brightspace:

- `maze.py`