CS2121 - Project 1 - 2019

Stacks and Search: Automatically Solving a Maze

Out: February 15th, 2018

In: March 18th, 2018 at 11:55 pm via OWL

Note Well:

You must take note of the following:

- **Before you start** you will need to install a Python package called PyGame. To do this, simply open up your terminal/cmd and type: `pip install pygame`. If this does not work because of permissions issues, instead type `pip install --user pygame` to install PyGame in your user directory.
- **If you are a 9643B student**, you have some additional requirements that are noted below and marked with "**(for 9643B only)**". The continuation of instructions for everyone will then be indicated.
- You are given three (3) .py files, two of which you leave unchanged and one you alter. You are also given a set of maze text files and an images directory that must remain unchanged. If you happen to accidentally change any of them, you can always re-download a fresh copy and replace the changed files/directory.
- **Make sure** to give yourself **ample time** to work on this project. You are given over a month to work on this because it will take you time to develop different pieces of the project to get them working and then put it all together. **Do not** expect that you can pick this up a few days before the deadline and get it all working in time.
- Some of the descriptions are deliberately left somewhat vague. This gives you practice interpreting design instructions, but things will become clear as you get further on with your implementation. Part of the project is to figure out what the requirements mean and then plan your implementation to meet them.
- If there is something you are not sure how to do, you should first Google it or look it up in the Python documentation online: `https://docs.python.org/3/index.html`

Purpose

The purpose of this project is manyfold. You will continue learning how to interpret functional requirements (specification of the behaviour of software) and how to develop your implementation to meet them. You will also learn to work with already implemented classes, while developing your own to implement a *stack*, perform a *depth first search*, and become familiar with command line arguments.

Summary

You will be writing a program to solve a maze automatically with a depth first search, which will require you to implement a stack data structure. Note that **your algorithm is *not* expected to solve the maze optimally**.

Provided:

You are provided with three python files, two of which that are already complete (`App.py`, `Maze.py`) and one that you must complete (`DFS.py`). You are provided with four text files (`m1.txt`, `m2.txt`, `m3.txt`, `m4.txt`) containing the raw maze data for four mazes. And you are provided with an `images` folder containing images needed for the maze display to work correctly. Do not alter the location or content of this folder or the maze files.

What It Should Look Like:

Below is a link to a little video of how things should look in the end:
`https://giphy.com/gifs/2shzGoPB1une5qrC8R/fullscreen`

Maze class

Nothing to do here. This class is complete and is just needed to help with visualization of the maze.

App class

Nothing to do here either. This class is also complete and needed to help with maze visualization. This class is not documented, but you are free to investigate the code if you like to try to figure out what it does. This is a good challenge, but not required.

Note that the `App` class is set up so that for an `App` object `a`, you can call `a.pause_for_key()` to force the app window to wait for the user to press a key to continue, and so that you can call `a.on_cleanup()` to allow the window close either by pressing any key or by clicking on the x button.

Stack class

You need to implement your own stack for the algorithm to operate. You can do this however you like, using whichever underlying data structure (list, linked list, etc.) you prefer.

(for 9643B only): You must implement this class using a linked list. You can use the code from assignment 2 to do this.

(for everyone): If you do not know what a stack is, then look it up online or read ahead in whichever textbook you are following. You do not need to understand all of the nuances of stacks to implement a basic stack structure. For this, all you actually need are the methods outlined below, but you are free to add other methods if you like:

1. `__init__`
   - A "constructor" to initialize the class.
   - How you do this initialization will depend on how you implement the stack, i.e., what underlying data structure you use.
   - Note that if you choose to implement your stack with a linked list structure, then you will also need a `Node` class.
2. `push(self, data)`
   - Method to push an item of data onto the top of the stack.
3. `pop(self)`
   - Method to pop the top of the stack, i.e., remove the top data item from the stack and return it to the user.
   - If you are using a linked structure, make sure this returns a reference to the data and not to the node.
4. `peek(self)`
   - Has the same return behaviour as `pop` but leaves the top item on the stack.

Node class

This class is only needed if you decide to implement your stack with a linked structure. If you are implementing your stack differently, just ignore this.

(for 9643B only): This class is required for you.

DFS class (for everyone)

This is the Depth First Search class that will actually be used to solve the maze. You will need to complete this class so that it works to solve the maze automatically. Below are a few methods that I am recommending that you implement in order to make solving the problem easier.

Remember that breaking up a problem into manageable subtasks is an effective strategy for making problem solving simpler, and that doing this with code helps to write clearer, more reliable, and more maintainable code. Choosing what functions/methods to implement, thereby packaging certain code into a function/method, is a part of this general strategy.

Here are the functions I recommend you write:

1. `isLocValid(y, x)`
   - This method will take a coordinate in the maze and will tell us if that coordinate is within the bounds of the maze (`True`) or not (`False`).
   - For example, if we have a 10×10 maze, we can check if location $(5, 7)$ is valid (it is). If we check $(10, 7)$ or $(-1, 3)$, it would return `False`.
2. `getNextNeighbour(y, x)`
   - All places in the maze have a potential of 4 neighbours (up, right, down, left).
   - **Always** check neighbours in a clockwise direction starting with up (up, right, down, left).
   - This method will return the coordinates of the next valid and open neighbor cell.
   - Positions that are not valid, i.e., walls and already visited cells (past or presently in our path) should not be returned; only open positions and the end will ever be returned.
   - If no such neighbour exists, return `None`.
   - **Hint**: `isLocValid` might be handy here.
3. `loadMaze(filename)`
   - Just load the data from the maze text file and return a `numpy` array of the maze.
   - If you have never used the `numpy` package, look it up online. You only need to figure out how to construct a 2D array consisting of characters, which are Python `str` objects. (Note that you should not use the deprecated `numpy.chararray` function to do this).

Instructions

Our maze search will be what we call a *depth first search*. Simply, we're going to search through the maze for the end, and whenever we get to a dead end, we will backtrack and carry on when we have new places to explore. There is no guarantee that our solution will be optimal.

(for 9643B only): You will have to implement the search strategy described below as well as a variation on this strategy, described below. See below for details on what you need to do in addition to what everyone is required to do.

(for everyone) Again, have a look here to get an idea for what should happen:
`https://giphy.com/gifs/2shzGoPB1une5qrC8R/fullscreen`

The maze is just a grid (2D array of characters) of positions. Each position in this grid can either be a wall (black) or an open space (light grey) (including the start (blue) and end (red)). The search cannot go through walls and should only ever travel through open positions. We want to mark where in the maze our search currently is (yellow). Additionally, we want to make sure we're keeping track of the path we've taken in our maze. The positions in the maze that are in our current path will be blue and the positions that we've looked at, but have backtracked from, will be dark grey. The status/type of position in the cell can be represented by a character in our 2D character array representing the maze (e.g., we change the value in the character array to `'c'` when it is the current position we are in).

A list of position statuses (character values in the 2D character array) is listed below along with the character representing position status:

- `'p'`: on the current path (blue circle)
- `'c'`: the current position (yellow circle)
- `'m'`: empty cell (white cell – ignore this possibility)
- `'E'`: end/exit of the maze (red)
- `'f'`: end/exit is found (yellow circle on red)
- `'x'`: previously taken path (dark grey circle)
- `' '`: open cell (light grey)
- `'S'`: start of the maze (blue You might never really see this pop up once the search starts)
- `'#'`: wall cell (black)

For our purposes, treat each position as a tuple representing the coordinate of the position in the array/grid. We can then use this tuple to easily access the contents in maze array/grid. By the way you can actually index

a 2D `numpy` array with a tuple, so that could come in handy, but be warned that the order of the tuple matters: (x, y) vs. (y, x). When you read in the maze files, they will just be a text file containing spaces, `#`, `S`, and `E`.

The following is pseudocode for the procedure you should use for solving the maze:

1. Open maze file and store the character data as a 2D `numpy` array.
2. Create an `App` object with the maze.
3. The maze should now be set up.
4. Find the start of the maze. This position will be where the `'S'` is in the maze. **You must** treat positions in the maze as tuples. So, for example, (x, y) coordinates – or possibly (y, x) if that works better for you.
5. Create a stack object.
6. Add the start position to the stack – so add the position tuple of the start of the maze:
   - For example, `myStack.push((x,y))`
7. `while` the stack is not empty, and we have not yet found the end, do the following:
   (a) Peek at the top of the stack and get the current position.
   (b) If the current position is the end, then awesome, we're done.
   (c) Otherwise, get the next neighbour.
   (d) If there is one, push it and update the position's status (current position, current path, previously taken path, etc.)
   (e) If there are no possible neighbours, then we're at a dead end and should backtrack. Pop the current position from the stack and update the position's status.
   - **Note**: each time through the loop, at some point, you're going to want to tell the `App` to re-draw itself. To do this, call the `App`'s `on_render()` method. If you do not do this, your maze will not look like it's doing anything. You will know you are doing this correctly when the display looks like my example video.
   - **Note**: You **must** add a delay to your program. Without this, things will happen *way* too fast. Check out `time.sleep(someNumOfSeconds)` (there's an example in the `DFS` file you got). A good place to call this is right next to the render call.
8. Update the ending tile's status and call `on_render()` one last time.
9. In the end, print out the total number of steps it took to find the exit (times that the current position changed), and print out the total number of steps in the actual path to the exit. Look at the example video provided for an example.

(for 9643B only): Once you have this all working, add an boolean flag (as an input parameter) to the `getNextNeighbour` method so that when this flag is set to `True` the method behaves the same way except that checks the possible directions in a *random* order (rather than always clockwise). This way, different runs on the same maze will give rise to different results. Set up your flag to take the value `False`, i.e., not random, by default. If you are not taking 9643B, you are welcome to try this, but **make sure that this behaviour is turned off (flag is set to `False`) by default**, as you stand to lose marks if your code does not do what it is expected to do.

Once you have the randomized next neighbour selection working, compute the average, median and standard deviation of the total number of steps taken and the number of steps in the successful path. Compare your results to the total number of steps and steps in the path for the deterministic algorithm. Your code should compute and print out this information for trials of 30 runs for each maze.

(for everyone): Your program will also have to run with command line arguments. Think of these as like parameters in a method, but rather that being for a method, we have parameters for the whole program. The command line arguments you will pass to the program will be the file name of the maze we want to solve and the delay.

If you look at the example in the video you will see that I call `python DFS.py m2.txt 0.4`. When I hit run, it automatically loaded up maze 2 with a delay of 0.4s. If I were to have typed `python DFS.py m3.txt 0.04`, it would automatically run maze 3 with a delay of 0.04s. I should be able to load up any possible maze file this way. I strongly recommend googling "python command line arguments". You will find that it's pretty easy, but you will need to learn how to do this yourself.

Note that if you use an IPython console instead, you can use the code `run DFS.py m2.txt 0.4` to execute the code in precisely the same way as my video example did.

(for 9643B only): In addition to the maze filename and delay, your code should take a third optional parameter, an *integer* indicating whether the boolean flag for the `getNextNeighbour` method is to be set to `True` (random neighbour selection) or `False` (deterministic neighbour selection).

Submitting the Assignment (for everyone)

You will submit the project via OWL, under the assignments tab. **Only submit your complete python files** (your stack files, `DFS.py` and any helper files you need) via OWL. Please follow the instructions carefully and do what is asked, but not more, to ensure that you do not unnecessarily lose marks. Make sure not to modify the function names.