Math 560 Project 2: DFS/BFS Spring 2024

Due: Wednesday, April 3rd

In this project, you will implement both DFS and BFS to solve several different mazes. You will be required to implement your solutions in their respective locations in the provided project 2 files. You must submit your `project2.py`, `p2stack.py`, and `p2queue.py` code online on Gradescope.

Contents

1 Problem Statement

You will have three primary tasks in this project:

1. Properly implement a Stack class for use in DFS.
2. Properly implement a cyclic Queue class for use in BFS.
3. Implement the function `bdfs` that can run either DFS or BFS to solve the maze.

You should test your code in the main function of the `project2.py` file, since this file imports all of the others.

You must submit your `project2.py`, `p2stack.py`, and `p2queue.py` code online on Gradescope. If you worked with a partner, only submit one version of your completed project and indicate clearly the names and NetIDs of both partners.

This project is worth 100 points. The points will be assigned as follows:

- 20 points each for a correct Stack and Queue
- 30 points each for a correct DFS and BFS implementation

1.1 Stack Class

You have been provided with a partially implemented Stack class in `p2stack.py`. It has 3 class attributes:

- stack: the array (a python list we are treating like an array) representing the stack
- top: the index of the top element of the stack (will point directly at the top element)
- numElems: the number of elements currently in the stack

There are 2 fully implemented member functions:

- `__init__`: the constructor which initializes an empty stack

- `__repr__`: the printing function that displays the stack's info in a readable format

You will be responsible for implementing five more member functions:

- `isFull`: this function should return true when the stack is full (and requires resizing)
- `isEmpty`: this function should return true when the stack is empty
- `resize`: this function should resize the stack by doubling the size of the array
- `push`: this function should take in some value and push it onto the top of the stack

  Note: it is intended that you call the `isFull` and `resize` functions from **within** the `push` function only, i.e., you should not call them elsewhere before the call to `push`.
- `pop`: this function should pop the top value off the stack and return it

Note: you are not allowed to use built-in functions like append or remove for these tasks. Instead, you are expected to implement these functions directly using the provided attributes of the Stack class.

It is highly suggested that you ensure your Stack code is working correctly before proceeding.

1.2 Queue Class

You have been provided with a partially implemented Queue class in `p2queue.py` (note that you need to implement a cyclic queue as discussed in lecture). It has 4 class attributes:

- queue: the array representing the queue
- front: the index of the front element of the queue (will point directly at the front element), this is the next element to be popped
- rear: the index that is **ONE PAST** the rear element of the queue, this is the location where the next pushed value will be written
- numElems: the number of elements currently in the queue

There are 2 fully implemented member functions:

- `__init__`: the constructor which initializes an empty queue
- `__repr__`: the printing function that displays the queue's info in a readable format

You will be responsible for implementing five more member functions:

- `isFull`: this function should return true when the queue is full (and requires resizing)
- `isEmpty`: this function should return true when the queue is empty
- `resize`: this function should resize the queue by doubling the size of the array, while also unwrapping the queue so that the front and rear values are reset

- `push`: this function should take in some value and push it into the rear of the queue

  Note: it is intended that you call the `isFull` and `resize` functions from **within** the `push` function only, i.e., you should not call them elsewhere before the call to `push`.

- `pop`: this function should pop the front value from the queue and return it

Note: you are not allowed to use built-in functions like append or remove for these tasks. Instead, you are expected to implement these functions directly using the provided attributes of the Queue class.

It is highly suggested that you ensure your Queue code is working correctly before proceeding.

1.3 DFS/BFS Implementation

The bulk of your submission will concern the function `bdfs` in `project2.py`. This function should take in an input Maze object and a string that is either `'DFS'` or `'BFS'`. Your implementation of this function should be able to run either DFS or BFS depending on this input string. Recall that the only major difference between the two algorithms is that one uses a stack and the other uses a queue.

The Maze class provides you with both an adjacency list of Vertex objects, and an adjacency matrix. So, you may use whichever representation you prefer when creating this function.

The output of this `bdfs` should be a list of vertex ranks (NOT Vertex objects) representing the path starting at the start vertex (which should be the first rank in your returned path) and ending at the exit vertex (which should be the last rank in your returned path).

You should not call this function directly when creating your code, but should instead use the Maze class's member function `solve`, i.e., to test the small open room maze with printing and plotting enabled:

```
m = Maze(0, True)
m.solve('DFS')
m.solve('BFS')
```

Note that the path will be overwritten with each call to solve, so you can re-solve a maze that has already been solved.

Note that the function call `testMazes(True)` will test all four provided mazes with both DFS and BFS, and print the resulting paths (you can set the verbosity input to False to suppress the printing).

2 Provided Code

Your goal in this project will be to implement both DFS and BFS to solve several given mazes. To aid you in this, you have been provided with several fully functioning classes,

several partially implemented classes, and functions for creating and testing the mazes.

2.1 Vertex Class

The first of the fully functioning classes is the Vertex class in `p2maze.py`. This class has 5 class attributes:

- rank: the rank (label) of the given vertex
- neigh: the list of the neighboring vertices
- dist: the distance from the start vertex
- visited: a flag for marking a vertex as visited
- prev: the previous vertex in the path

Along with these are three implemented member functions:

- `__init__`: this is the constructor function for the Vertex class. It requires an input rank for the vertex, and sets all of the attributes to have reasonable starting values. You will create a new Vertex with a call: `v = Vertex(rank)`.
- `__repr__`: this function is called whenever a Vertex is printed, i.e. when the call `print(v)` is made. It simply prints the rank of the vertex.
- `isEqual`: this takes in a second Vertex as an input, and compares the rank of the two vertices, returning True if they are equal rank (i.e., if they had the same label). This function can be called using: `v.isEqual(u)`.

2.2 Maze Class

You have also been provided with a fully functioning Maze class in `p2maze.py` with 7 class attributes:

- maze: a 2D array representing the maze (for internal use only)
- adjList: the adjacency list of Vertex objects
- adjMat: the adjacency matrix (stored as a 2D list of 0 or 1)
- start: the starting Vertex object
- exit: the exit Vertex object
- path: what will ultimately contain the path from start to exit, stored as a list of ranks (not a list of vertices)
- verb: a flag that controls the verbosity of the printing, set to False to suppress printing the maze when solving and testing

The Maze class has 6 member functions:

- `__init__`: this is the constructor for the Maze class. It has two optional inputs: the mazeNum which selects which maze to use (options: 0,1,2,3 - default: 0); and the verbosity (True/False - default: False). This initialization function correctly creates the adjacency list and matrix, and finds the start and exit vertices. The path attribute is initialized as an empty list. A new maze can be created with the call `m = Maze(mazeNum, verbosity)`.
- `__repr__`: this function is called when a Maze object is printed. It will check to see if a path has been set, and if that path correctly solves the maze. If verbosity is True, this function will print out the maze with a highlighted path if one was present. If verbosity if False, only statements concerning the correctness of the path will be printed.

  When the maze is printed, walls are marked with an `'X'`, the start with an `'s'`, the exit with an `'e'`, and the path with `'o'`. If you see a `'G'`, it means your path attempted to pass through a wall. If you see an `'R'`, your path had a repeated vertex.
- `printList`: this function can be used to aid with debugging. It prints the adjacency list in a more readable format.
- `printMat`: this function can be used to aid with debugging. It prints the adjacency matrix in a more readable format.
- `plot_maze_solution`: this function will create a matplotlib figure based on the current maze and path info. It takes in as input a flag for plotting (default: True), which will actually call the `plt.show()` command when True. If this flag is set to False, the figure is created, and a later call to `plt.show()` will display it.

- `solve`: this function takes in an alg (either the string `'DFS'` or the string `'BFS'`), a flag for verbosity (default: True), and a flag for plotting (default: True). It then calls the function `bdfs` on the maze to obtain the path through the maze, which you will be responsible for implementing.

2.3 `getMaze` and `testMazes`

You have also been provided with two functions that create the maze and test your code. The `getMaze` function is found in `p2maze.py`, while the `textMazes` function is in `p2tests.py`:

- `getMaze`: this function takes in 0, 1, 2, 3, 4, or 5 and outputs the 2D array of 0s and 1s representing the maze. This is used internally by the Maze class. The mazes are:

  0. A small open room for debugging.
  1. A small maze for debugging.
  2. A large maze.
  3. A large zig-zag map.
  4. A specially designed test that will guarentee different behaviors for DFS and BFS. For this maze, DFS cannot find the shortest path, while BFS must find the shortest path.
  5. A randomly generated maze.

- `testMazes`: this function will allow you to test your code on the entire set of mazes. It takes in an optional argument verbosity, which controls whether each maze gets printed as it is solved (default = False).

3 Pair Programming

You are allowed to work in pairs for this project. If you elect to work with a partner:

- You should submit only one version of your final code and report. Have one partner upload the code and report on their Sakai site. Make sure that both partner's names are clearly indicated at the top of both the code and the report.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating').
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

4 Style Points

Part of your grade for this project will be 'style points'. The idea here is that the code you turn in must be well commented and readable. A reasonable user aught to be able to read through your provided code and be able to understand what it is you have done, and how your functions work. For these sorting algorithms, this means that a grader should be able to read over your code and tell that your algorithms are implemented correctly.

The guidelines for these 'style points':

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.
- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code. In python, you can use the symbol \ to indicate a line break/continuation.