CS2040S: Data Structures and Algorithms

Problem Set 7 Part 1

Due: Monday, March 27, 11:00am

Collaboration Policy.   You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so by leaving a comment at the start of your .java file. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 7.   (Mazes are Just Dynamite! Part 1)

It's easy to get lost in life. In this series of problem sets, we're going to develop a program to help us get unlost. Even better, we're going to give you some extra superpowers: the ability to break down the obstacles in your way, helping you to get to your destination even faster.

To start off for now, you are given a map of the maze you are lost in. Like most mazes, this maze consists of one or more rooms. Each of the rooms has doors to one or more other rooms. Your first job is to write a program that will explore the maze, as is, and discover the shortest path from your current location to a destination location. Simultaneously, your program will be expected to determine some other geographic information about the maze.

We'll talk about your superpowers later in part 2 of this problem set series.

Preliminaries.   We have provided you with some basic framework code for handling the maze. A maze consists of an $r \times c$ grid of rooms, with adjacent rooms separated by either a wall or an empty space. Furthermore, the entire maze is bordered by walls.

The rooms are numbered starting from the top left corner (which is $(0,0)$). The first coordinate specifies the row number, and the second coordinate specifies the column number. For example, in a 5×5 maze, the bottom left corner is $(4,0)$ and the top right corner is $(0,4)$.

Here is a pictorial example of a 5×5 maze:

```
    0 1 2 3 4
   ###########
 0 #R R R#R#R#
   ### # # # #
 1 #R#R#R R R#
   # ### ### #
 2 #R R R#R R#
   # ### # # #
 3 #R#R R#R#R#
   # # # ### #
 4 #R#R#R#R R#
   ###########
```

In this diagram, each wall is depicted using a hash symbol, i.e., `#`, while each room is depicted using the letter `R` (this is for visualization purposes only – in the actual maze files, the rooms are represented as empty spaces as well!). Notice that in each row, there are exactly c rooms and at most $c+1$ walls (including the left and right borders).

You may move between adjacent rooms in any of the four cardinal directions (north, south, east and west) if there is no wall in that direction. Diagonal movement is not allowed. In the above example, one can move from (0,0) to (0,1), but **NOT** from (0,0) to (1,0), nor from (0,0) to (1,1).

```
##########
#PPPPP# # #
### #P# # #
# # #PPPPP#
# ### ###P#
#       #  P#
# ### # #P#
# #     # #P#
# # # ###P#
# # # #  P#
##########
```

Figure 1: Example printMaze output of a solved maze

We provide you with two classes `Maze` and `Room` that represent the maze that your program will solve. The size of the maze is represented by the number of *rows* and *columns* in the maze (in the above example, both *rows* and *columns* are 5). The maze itself is represented by a matrix of rooms. You will be able to check if there exists a wall in the four directions of the room through the public methods `hasNorthWall()`, `hasSouthWall()`, `hasEastWall()` and `hasWestWall()`. On top of that, there is a public boolean attribute `onPath` that you can set for each room (for printing of mazes).

The `Maze` class has a static method `readMaze(String fileName)` that reads in a maze from a text file and returns the maze object. We have provided several sample mazes for you to experiment with. We also provide a simplistic way of visualizing a maze through the static class `MazePrinter`. The static method `void printMaze(Maze maze)` of the `MazePrinter` class prints out a maze to the standard output[1], as per the example in Figure 1.

We also provide you with the class `ImprovedMazePrinter` which prints out a much prettier version of the maze. (It was contributed by a former student: Mai Anh Vu.)

NOTE: Please remove any references to `ImprovedMazePrinter` before submitting your code to Coursemology as it is not included as part of the submission files.

In this problem set, you will implement the class `MazeSolver` that will implement the provided interface `IMazeSolver`.

---

[1]For an additional zero extra bonus points, implement a prettier graphical maze display and share it with the rest of the class.

Problem 7.a.   The Average Coder

Joe the Average Coder is eager to solve this problem and implemented the class `MazeSolverNaive` (found in *MazeSolverNaive.java*). In particular, Joe implemented the `pathSearch` method using the **Depth-First Search** traversal that he learned in his Algorithms and Data Structures class.

Take a look at Joe's code. Given an arbitrary maze, will his `pathSearch` method find the shortest path from the start to the target? Why or why not? (You do not need to submit any code here. Just explain your answer. Your explanation must be correct and complete. If your answer is no, a well-explained counter-example is sufficient.)

Problem 7.b.   Exploring the Maze

Now implement the class `MazeSolver` that correctly implements the `IMazeSolver` interface.

NOTE: Your solution for this part must not modify anything apart from from `MazeSolver`.

First, implement the constructor `MazeSolver()`. Then, implement `void initialize(Maze maze)` to initialize the solver given a new maze.

Next, implement the method:

`Integer pathSearch(int startRow, int startCol, int endRow, int endCol)`

which searches for the shortest path from the specified start room to the specified end room. It should return an integer representing the minimum number of steps needed if a path is found, or `null` if no such path is available. When your search is complete, we should have `R.onPath == true` for every room R on the path and `R.onPath == false` for every room R not on the path. If done correctly, when you execute `printMaze(Maze maze)` after your search, it draws the path found, as in Figure 1.

Finally, implement the method:

`Integer numReachable(int k)`

which returns the number of rooms whose minimum number of steps to reach it from the most recent `pathSearch` starting location, is k. For example, how many rooms are reachable with minimum 0 steps? How many rooms are reachable with minimum 1 step? How many rooms are reachable with minimum 2 steps? etc. For the example maze shown in Figure 1, with the most recent `pathSearch` start location being `(0,0)`, the following holds (on the next page):

```
0 Step:  1 Room
1 Step:  1 Room
2 Steps: 2 Rooms
3 Steps: 1 Room
4 Steps: 2 Rooms
5 Steps: 4 Rooms
6 Steps: 5 Rooms
7 Steps: 5 Rooms
8 Steps: 3 Rooms
9 Steps: 1 Room
```

Notice that for each k, it outputs the number of rooms for which the minimum path distance is k exactly. If you calculate this for every initial starting point in the maze, then you can determine the *diameter* of the maze.

Your `numReachable` method should be **efficient**, as it will likely be called several times after a `pathSearch`.

Problem 7.c. Maze Generation (Optional)

Sometimes you need to escape a maze, sometimes you need to build one[2]. In this problem, we would like to implement an algorithm for generating a maze. To get started, you may wish to read through this interesting review of various maze generation algorithms:

http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

This blog entry summarizes 11 different methods for generating a maze. All of these techniques yield mazes that have only one route from start to finish (i.e., there are no cycles—they generate a tree). You might think about how to modify them to generate interesting graphs/mazes that have more than one possible solution.

Inside the `MazeGenerator` class, implement the following method:

`Maze generateMaze(int rows, int columns)`.

It should return a randomly generated maze given the number of rows and columns we would like the maze to contain. If you feel that it will make the generated mazes more interesting, feel free to add or remove parameters from the `generateMaze` method.

Submit your `MazeGenerator` class, along with a discussion of the design decisions that you made, and your favorite maze that was generated by the algorithm (with no manual intervention).

Please share any fun mazes you come up with (whether by hand or by algorithm) on the forum to receive "The Architect" achievement!

---

[2]See, for example, *Inception* (2010).