CS2040S: Data Structures and Algorithms

Problem Set 8

Due: Wednesday, April 8, 11:59pm

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.

Overview

It's easy to get lost in life. This week 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, 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.

Your second job involves determining how best to use your superpower. You have been given the power to bypass some fixed number of walls. Please remember to specify your superpower:

- Able to walk through walls.
- Able to fly over walls.
- Strong enough to knock down walls.
- Possesses dynamite.
- Some other superpower.

For zero points of extra credit, explain how your superpowers work and how you acquired them (e.g., on the forum). In any case, your job is to find the shortest path from the source to the destination, while bypassing no more than the allowed number of walls.

Problem Details

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 there are exactly c rooms and at most $c + 1$ walls (including the left and right borders) on each row.

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).

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].

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.)

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

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

Figure 1: Example printMaze output of a solved maze (with no superpowers)

In this problem set, you will implement two classes `MazeSolver` and `MazeSolverWithPower`, that will implement the provided interfaces `IMazeSolver` and `IMazeSolverWithPower` respectively.

Problem 8.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 that will return the **minimum** number of steps to get from a given starting coordinate to a target coordinate. He did so 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 algorithm be able to 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.)

Problem 8.b. Exploring the Maze

Now implement the class `MazeSolver` that correctly implements the `IMazeSolver` interface. First, 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 that `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, you will be able to execute `printMaze(Maze maze)` after your search is completed to draw the path taken by the solver, as in Figure 1.

Next, implement the method: `Integer numReachable(int k)` that will return an integer indicating how many rooms there are such that the **minimum** number of steps required to reach it is k, based on your most recent `pathSearch` starting location. Your `numReachable` method should count the number of rooms reachable from the initial location for each possible number of steps. For example, how many rooms are there such that the minimum path distance is 0? 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:

```
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 8.c. Maze Exploration for Real SuperPeople

In this part, as in the previous part, your job is to determine the shortest path from a specified source to a specified destination. However, you are also given the ability to bypass (demolish, jump over, or magically traverse) up to a limited number of walls along the way. Of course, you will not be allowed to use your power to demolish the outer walls that surround your maze. You begin with a fixed amount of superpowers, and every time you demolish a wall, your power reduces by one. Your goal is to find the shortest path from the start to the end.

Create a new class `MazeSolverWithPower` that implements `IMazeSolverWithPower` which contains the overridden method:

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

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. As before, it should also update, for each room, the `onPath` attribute. (Notice that the `IMazeSolverWithPower` interface inherits from `IMazeSolver`, and hence must correctly implement all the methods from there as well. In this case, if `numReachable` is called

after a `pathSearch` with superpowers, then its answer should also be based on having the same number of superpowers (in addition to having the same starting location))

Hint: One strategy is to represent the state as a combination of the current room and the remaining amount of superpower. If you explore this graph of all possible states, you will visit all possible rooms, with all possible remaining amounts of superpower.

Problem 8.d. (Bonus) Maze Generation

Sometimes you need to escape a maze, sometimes you need to build a maze[2]. In this problem, we would like to implement an algorithm for generating a maze. To get started, you may wish 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!

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