Assignment 3a

COMP 2526 Object-Oriented Programming with Java

Complete assignment due on Friday April 15th at 11:59 PM

1 Purpose

Use object oriented programming techniques to design and implement a recursive solution to the classic maze solving problem.

2 Description

2.1 Recursion and Solving Mazes

How do you solve mazes? How do robots solve mazes? Which are the most efficient ways to find the solution(s)?

Suppose you are walking through a maze and you don't have any string, markers, bread crumbs, or any other tools to record your path. You might try to follow the general approach of walking down every path as far as you can go. You will either reach the exit and complete the maze, or you won't be able to go any further. If you cannot go any further, then you will probably retrace your steps (backtrack) until you reach the most recent fork in the path.

At the fork, if there is a path you haven't taken, you will probably take it and follow that branch. If there isn't a path you haven't taken, you will retrace your steps to the next oldest fork, and so on.

If you do this systematically, eventually you will walk through every path in the maze and find the exit(s).

2.2 Depth First Search, Breadth First Search, and Dijkstra's Algorithm

This systematic state search is called a depth first search. A depth first search follows each path as deeply as it can before considering alternate routes. Implementing this systematic, methodical backtracking is a perfect job for recursion. You will never take the exact same path more than once, and you will eventually find a solution if one exists.

A closely related search, breadth first search, forms the backbone of some very interesting algorithms like the Dijkstra algorithm. In a breadth first search, we keep visiting each path one after another, taking one more step each turn and branching in all directions at once when we encounter intersections.

2.3 The Problem and its Parameters

Use recursion to implement backtracking (depth first search) and find and display the correct path(s) through a maze. We will observe the following constraints:

1. The maze must be read from a text file. There are 25 lines, each containing 25 characters. Mazes consist of a) asterisks (*) which represent walls, and b) other characters which represent points in the maze that your maze solver may occupy.
2. From each point in a maze, you can move to the next cell horizontally (left and right) or vertically (up and down), as long as the next cell is not a wall (so there are at most 4 possible moves from each point, minus the direction you entered)
3. We will always enter the maze from row 0, column 1 (in matrix-speak this is maze[0][1])
4. The path may not contain loops, e.g., a cell may only be used once in a path
5. The maze exit(s) will always be on the right hand side of the maze. As soon as you reach the rightmost column of the maze, i.e., column 24, you have exited.

3 Requirements

This assignment should introduce you to a classic application of recursion, and demonstrate how solving a challenging problem can be easier with a well planned design. You must adhere to these rules:

1. Import the zipped framework into Eclipse (File >Import >General >Existing projects into workspace >Next and choose the archive)
2. You may NOT modify the Main.java, GameFrame.java, or MazeEntryPointException.java files (except you may modify the GameFrame.java file where noted below).
3. Maze stores a representation of the maze and also contains the logic for extracting it from correctly formatted files (more below). Maze contains the following methods:
   (a) init() initializes the maze
   (b) clear() eliminates all walls from the maze and 'unvisits' each maze section, effectively wiping it clean
   (c) makeSolid() renders the specified MazeSection solid
   (d) makeNavigable() renders the specified MazeSection navigable (a path section)
   (e) generateMazeFromFile() generates and stores the maze stored in the file passed as a parameter. The file must be a .txt file which contains 25 lines of 25 characters each. There are two characters: * delineates a solid section, and anything else delineates path. The entry to the maze must be at location [0][1] and the (possibly multiple) exit(s) to the maze must be on the right edge (rightmost column) of the maze [0...24][24]
   (f) reset() resets but does not delete the current maze by 'unvisiting' all visited maze sections and resetting their colour to white
   (g) generateRandomMaze() should generate a random maze
   (h) we have included the headers for some additional methods you may find helpful.
4. MazeSection represents a section of the maze that has a location and colour. A section can be either wall (impassable) or path (navigable). MazeSection contains the following methods:
   (a) init() initializes the MazeSection. We recommend you set the border and layout, and maybe even add a MouseListener to respond to clicks (it may be helpful to edit the maze by turning walls into paths and vice versa)
   (b) we have included the headers for some additional methods you may find helpful.
5. MazeSolver contains the solutions to the maze and the logic that solves it. The MazeSolver uses an ArrayList of ArrayLists to store the solutions it finds. MazeSolver contains the following methods:
   (a) clear() deletes the currently stores collection of solutions
   (b) solveMaze() returns a collection of solutions to the maze
   (c) generatePaths() is a recursive method invoked by solveMaze – this is where you will implement the recursive portion of your algorithm
   (d) we have included the headers for some additional methods you may find helpful.

6. Note that the GUI has some primitive menu bar functions. You may open a file, generate a random maze, or execute the algorithm on the currently loaded file. It would be helpful if you modified the menu bar (in the GameFrame class) to include some keyboard shortcuts. Also, can you use the depth first algorithm and some clever use of the Random() object to generate a better random maze?

It only seems fair that we provide you with some mazes to test your code, and the correct output your program should generate for each:

1. Maze 0 Solutions.txt contains no solutions. Your program should open it and display it without issue, and report no solutions when you try to solve it:

```
*  ************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
```

2. Your program should open and display Maze Entry Point Exception.txt without issue. When you try to solve it, however, the maze in Maze Entry Point Exception.txt should trigger the MazeSolver's solveMaze() method to throw a MazeEntryPointException. The generateMazeSolutions() method in GameFrame will catch the exception and produce a suitable message for the user:

```
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
**************************
```

```
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
*************************
```

3. Maze 1 Solution.txt has one solution (a path that does not loop). Your program should report that there is one solution, and then trace it (the Swing code for this is already implemented):

```
* **** ****** ***********
*     * *     * *    *      *
* ** *    ** * * *** * * * *
* *   * * * ** *      * * *
  **** *       * ***** * * *
*        *******     ** * *
* *****          **   * * *
*  *     *** * * ***** * * *
** * ****** * ** *   * * * *
   *       * * ** ** * * * *
 ******** * *   *       * * *
 *   *   *   * **** ****** *
 ** ** **** ***   *       *
    **          * *********
****** ** ***** *         *
        **      * * ***** *
 *****    * ** * * *    * * *
 **    **** ** *    * *    *
 *   *     * *   * *** *****
 ** * **** ** *    *    *
    * *     ** * * *** * * *
 **** **** ** * *    * * * *
 *     *   * **** *** * * * *
 ****** **   *    *    *    *
         *************** **
```

4. Maze 16 Solutions.txt contains a maze which has 16 distinct solutions (paths to the exit that do not loop). Your program should report that there are 16 methods, trace through all of them, and finish by highlighting the shortest route:

```
* ************************
*
* ******************** ***
* ******************** ***
* ******************** ***
```

```
* ******************** ***
* ******************** ***
*
* ************************
* ************************
* ************************
* ************************
*
* ******************** ***
* ******************** ***
* ******************** ***
* ******************** ***
* ******************** ***
*
* ************************
* ************************
* ************************
*
*************************
*************************
```

It is likely you will want to add more classes and methods. You may wish to create an inheritance hierarchy for the MazeSections, for example, or use generics to generate an algorithm class that applies a chosen algorithm to the maze. This design can be improved, and you should apply what you have learned about encapsulation and other object oriented principles. Good luck, and have fun!

4 Marking Guidelines

50% Functionality (does your submission correctly solve mazes)
25% Good object oriented design
25% Comments and programming style (follow your lab instructor's guidelines)