**IT340 AI**

**Project 1 – Maze**

**Deadline: October 8 (Friday)**

**(The late pass deadline: Oct. 10 (Sunday)**

**Maze Project – Module 1**

**Maze Reader:**

You will be working in a team of three in developing a maze solver program. Consider your basic maze. It has walls and pathways, and it has one starting point and one (or more) exit point(s). Furthermore, one wall is just like another, and any open spaces (not including start and exit) are also identical. Your program will read a maze stored in a text file. Each maze component has an associated one-character representation:

|  |  |
| --- | --- |
| Walls | # (hash mark) |
| Open spaces | . (period) |
| Start | o (lower case ‘O’) |
| Exit (or goal) | \* (asterisk) |

Create a class called MazeReader that can store the logical layout of a maze. MazeReader should have a constructor that can read from a Scanner. The format will be that the maze will always be in a square form, where subsequent lines each contain one row of the maze with each character representing one square of the maze. Write a toString method that makes a String representation of this Maze in the same format as the input. This will be handy in testing your code.

Be sure to include a toString method that returns a one-character-long string containing only the character corresponding to a location. Also include a static method fromChar that returns the type of square associated with the given char. That is, a method with header public static Square fromChar (char ch) that, when provided with one of the four legal characters, returns one of the Square values. Any other input should generate an IllegalArgumentException.

A simple example of the maze file:

#######

#...#o#

#\*#...#

#######

A more complex example:

############

#.#....\*...#

#.#.######.#

#.#....#...#

#.###.\*#.#.#

#...####.#.#

#.#.#..#.#.#

#.#.#.##.#.#

#o#......#.#

############

**Maze Project – Module 2**

**Maze-o-Rama:**

All classes for this project are in the package maze. You are welcome to use your previous codes as well as incorporating Java library and of course writing several new ones. Note that this project does not explicitly list all the classes you'll have to write---part of that will be design decisions left up to you.

**The maze itself**

You have a MazeReader class that stores char or Square and read it from the file. Before you get to the solving part, this project requires a Maze that can store enumerated Square values. You may assume that any well-formed maze will have exactly one start and zero or many exits (or goals). You may assume that all valid mazes will be entirely enclosed with walls (unless the exit is in the outer wall).

**The maze solver**

The main component of the project is to write a MazeSolver class (and all the associated classes), which will bundle up the functionality of determining whether a maze has a solution---that is, whether you can get from the start to the exit (without jumping over any walls). The algorithm looks like this: start at the start location, and trace along all possible paths to (eventually) all reachable open squares. If at some point you run across the exit, it was reachable. If not, it wasn't.

**The task**

In addition to creating a maze solver, your team will be required to improve upon your algorithm for the shortest path (i.e., optimal path). Hint: You are allowed to traverse the path multiple times as long as the final path result is the most optimal one. (But we will measure your time to arrive the optimal solution.)

You are required to implement at least three search algorithms to find the shortest path to find an optimal solution and analyze their complexity to find the best algorithm among your implementations theoretically.

**Maze Project – Module 3**

**The Solver and GUI**

To handle all the fiddly graphical stuff, write a MazeApp class. When the program is started, it should present the following widgets across the top of the window:

* a text field
* a button labelled "Load"
* a button labelled "Start”

When the user loads a maze file through the "Load" button, the maze in that file should be read in and displayed on the applet as follows: open squares are white, walls are black. The start button should initiate the maze solution process. Once initiated, the step button makes the solution proceed by just one step and turning on the animation uses a Timer (see below) to set the solution running at the rate of a few steps a second. (Clicking the animation button again should stop the animation.) Make sure that your display is big enough to fit a 50 x 50 maze.

What we should see as the solution process goes on is that the squares that have been explored---yes, we can reach them, but no, we haven't found the exit yet---change from white to medium-grey. We should be able to see the maze slowly shaded in until the solver reaches the exit (or determines that it's unreachable). Make sure that there is a sufficient delay in the animation to see the path that it takes while solving the maze.

Somewhere conspicuous, the GUI should display a status line on the text field containing the current status of the solution process: “No maze”, “Maze loaded”, “Solution in progress“, “Solution complete: exit reachable at X,Y in N moves”, “Solution complete: exit not reachable”. Make sure that your program is capable of loading, displaying, and solving multiple mazes without having to restart/rerun the program.

Remember, the final solution to be displayed is the shortest path to the closest reachable exit (i.e., goal).

**Notes on Timer**

You might need to use the two Timer classes in the java libraries, they have similar functionality. You would probably want the one in javax.swing or javaFX, because the way it triggers its action is exactly the same way a Button does---via an ActionListener. Once the Timer is created, you will call its start(), stop(), and isRunning() methods to control it. The one you don't want to use is in java.util. If you are freely importing everything from that package, make sure you're not accidentally using the wrong Timer!

**Schedule & Final Deliverables:**

* **Every team** must submit the complete maze package and the complete jar file for the project to ReggieNet.
* **Every team** mustsubmit one unique and valid maze text file to reggieNet.
* **Every team** mustsubmit a report that theoretically analyze the time & space complexity of their chosen search algorithms.
* **Each team** must submit one valid maze (no extra characters or unrecognized characters, etc.) with a maximum size of 50 x 50 to be tested by others.
* We will have competition and presentation in class. Each team will present the approach and strategy to solve the Maze for 10 min.

**Grades:**

You are to show that your program has completed the three main components. Although artistic value does not warrant a higher grade, intuitive and ease of use does have an impact for Module 3.

Grades are divided as follows:

* Completion of Part 1 = 20 %
* Completion of Part 2 = 50 %.
* Completion of Part 3 = 30 %.

**Extra Credit:**

* Every team that comes up with a valid maze that cannot be properly solved by ALL maze solver (excluding their own) will receive extra credits.
* Every team that solves ALL mazes will receive extra credits.
* A Team that provides the fastest solver for all mazes will get extra credits.