**Lab #7: Queues and Mazes**

CS 270 – Fall 2023

## Goals for this assignment

* Use Queue<E> to determine if a maze is solvable.

## Overview

You will use a queue to determine whether or not a maze is solvable.

## Using Queue<E> to solve a maze

You will use Queue<E> to implement a maze solver. The basic idea is that your queue will serve as a list of spaces/squares within the maze that are on the frontier of the explored area of the maze.

I have provided you with a GUI (Graphical User Interface) that takes care of loading mazes from text files, and graphically visualizing the maze. You will write a class that will "plug-in" to the GUI to provide the maze solving logic.

### The maze solving algorithm

The basic maze solving algorithm involves the following. The algorithm starts by marking the start space/square (use the '@’ character) and adding it to your queue. The queue is your "agenda" of spaces/squares to explore within the maze. While the queue is not empty and you have not found the goal you will take the next space/square from the agenda (the queue), and check to see if that space/square is the goal. If not, you will scan all neighboring (above, below, right and left) spaces/squares and for any open spaces that have not yet been explored, mark them as explored (use the '@’ character) and add the spaces/squares to the queue (for later use). The algorithm then repeats until the goal is found, or the queue becomes empty (which means that the maze is not solvable).

* **Hint**: make sure to mark spaces/squares explored *before* they are added to the queue. This will avoid adding a given space/square to the queue multiple times.
* **Hint**: it would be useful to create a class named Cell. Instances of this class could be used to represent a space/square location in the maze, and your queue will contain elements of type Cell.

### Visualizing the maze

The maze is visualized using the class MazeGUI. Your program needs to create an instance of this class in order to make the GUI visible on the screen.

The GUI has a status bar at the bottom that can be used to display messages to the user. You can change the text in the message bar by calling the setStatusText method:

gui.setStatusText("Maze is solvable.");

In order to visualize the solving of the maze, you will need to "plug-in" to the GUI. I have provided an interface called MazeSolver. You will implement this interface and override the method. See the documentation of the interface for more details.

You will write the class QueueMazeSolver that implements MazeSolver. QueueMazeSolver creates an instance of MazeGUI using "this" class. I suggest the following design.

**public class QueueMazeSolver implements MazeSolver {**

**...**

**private MazeGUI gui;**

**public QueueMazeSolver() {**

**gui = new MazeGUI( this );**

**...**

**}**

**public void solve(char[][] maze, int startR, int startC,**

**int endR, int endC) {**

**...**

**}**

**...**

**}**

Your job is to write the code that is inside the method public void solve(...). The algorithm is described above. The char[][] array maze is described in the MazeSolver interface. Look at the documentation inside MazeSolver to understand how the maze is represented.

Your code within the solve(...) method will implement the algorithm described above. It can (and should) modify the array parameter maze. In order to visualize the algorithm, your code will call the method gui.drawMaze(maze) every time it updates the maze. Note that the display may update too quickly to be useful. To slow things down, you can use the following method.

**gui.drawMaze(maze);**

**try { Thread.sleep(250); }**

**catch( InterruptedException e ) { System.err.println("Thread interrupted!"); }**

The method Thread.sleep causes the currently executing thread to stop for a period of time. The argument is the number of milliseconds that it will "sleep." In the above example, it will sleep for a quarter of a second. The InterruptedException is a checked exception so must be caught or declared to be thrown.

### When your solver is finished

If your program is successful in finding the goal then it will change the status text to "Maze is solvable." If your program is unable to find a solution, then your program will set the status text to "Maze is unsolvable." To do so, call the setStatusText method in the MazeGUI object.

gui.setStatusText("Maze is unsolvable.");

### Starting the Program

To run the program, and make the GUI visible on the screen, you need to create an instance of the class QueueMazeSolver as described below.

**public class QueueMazeSolverMain {**

**public static void main( String[] args ) {**

**new QueueMazeSolver();**

**}**

**}**

## Before the Lab Session

1. Create an eclipse project for this lab and download the files: MazeGUI.java, Maze.java, and MazeSolver.java and add them to your project.
2. Download the maze data files (e.g., maze-17x17.txt) and add them to your project.
3. Create a class called QueueMazeSolver that implements the MazeSolver interface. Its constructor will create a MazeGUI object.
4. Create the class QueueMazeSolverMain with the main method that instantiates your QueueMazeSolver class (see code above). You will see the GUI when you run this program. Of course, it will not function until you implement the method solve().
5. Review Section 4.5 in the textbook.
6. Read through the documentation for interface MazeSolver before coming to lab.

## During the Lab Session

1. If you have not read the documentation in interface MazeSolver do so now.
2. Think about how you will solve this problem. Review the maze solving algorithm described above. I strongly recommend you write a pseudocode outline for the solve() method. Before you start coding.
3. Think about what you want to store within the queue. It might be helpful to create another class for this purpose (see Cell hint above).
4. Begin to implement the solution to the maze in your solve() method, utilizing your Queue.

## After the Lab Session

Complete, test and document your program.

## Creative additions

* Think about how you could find and display the shortest path to the goal. The GUI will display cells in a bright cyan color if you use the '%' character.  
  To find this optimal path you need to keep track of the cell that "found" the cell. In other words, when a cell is marked as explored, it was done "from" another cell. Keep track of that cell. For example, if I'm at cell (1,1), I might look at cell (1,2) and see that it hasn't been explored. I would need to remember that cell (1,1) took me to cell (1,2). When I find the goal cell, I can move backwards from the goal to the start by following these references. (5 points)
* Use a stack instead of a queue to solve the maze, and allow the user to choose which technique to use (stack or queue). (5 points)
* Improve the GUI to include a visualization of the queue and/or stack as the solver is running (5 points)

**Submission Instructions**

See the “Lab Requirements” document for details on submitting your lab assignment, and other important requirements. Include all of your JUnit tests in your submission.