CS 570: Homework Assignment 4

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Viola-tions will be penalized appropriately.

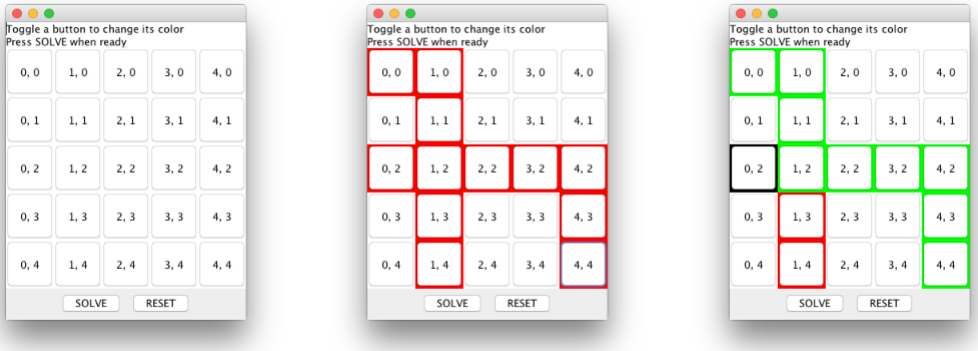
Late Policy. Check the course syllabus for the late submission policy.

2 Assignment

The aim of this assignment is to use recursion and *backtracking* to find a path through a maze. If you are attempting to walk through a maze, you will probably walk down a path as far as you can go. Eventually, you will reach your destination or you won't be able to go any farther. If you can't go any farther, you will need to consider alternative paths. Therefore we need to be able to systematically perform trial and error search. *Backtracking* is a way for doing just this. It is a systematic, nonrepetitive approach to trying alternative paths and eliminating them if they don't work. Recursion allows you to implement backtracking in a relatively straightforward manner. Each activation frame is used to remember the choice that was made at that particular decision point. After returning from a recursive call you can perform other recursive calls to try out different paths.

2.1 The Maze

The maze itself will consist of a two dimensional grid of colored cells. The maze will start at the cell (0,0) and the exit point is the cell (getNCols()-1,getNRows()-1). The operations getNCols() and getNRows() are methods of the class TwoDimGrid described below. All cells that can be part of the path will be in the NON_BACKGROUND color. All the cells that represent barriers



(a) (b) (c)

Figure 1: Sample grids

and cannot be part of the path will be in the BACKGROUND color. To keep track of a cell that we have visited, we will set it to the TEMPORARY color. If we find a path, all cells on the path will be reset the PATH color. So there are a total of four possible colors. Initially, the maze has all cells set to color BACKGORUND, as depicted in Fig. 1. The values we will use for each of these constants is:

- Green for PATH;
- White for BACKGROUND;
- Red for NON_BACKGROUND; and
- Black for TEMPORARY.

This is specified in the interface GridColors described below.

2.2 Class Layout

You are provided with a stub which may be downloaded from Canvas. It consists of the following classes and interfaces:

- Class TwoDimGrid. This class implements the two dimensional grid. It is used by the class Maze, which has a data field of this type. You will not need to modify this class in your code.
- Interface GridColors. This interface simply assigns names to the various colors that a cell can have. Here is the code:

```
import java.awt.Color;
public interface GridColors {

    Color PATH = Color.green;
    Color BACKGROUND = Color.white;
```

```
6       Color N O N _ B A C K G R O U N D = Color .
        red ; Color TE MP O RA RY = Color . black ;
8   }
```

- Class Maze. This is the class which you have to modify by implementing your search algorithm. It has a private field maze of type TwoDimGrid, where the two dimensional grid is stored.
  The method you have to implement is **public boolean** findMazePath(**int** x, **int** y). Also, you shall be asked to implement some variations of this algorithm, each of which will result in a new method. The details of these methods are described in Sec. 3 together with a description of other methods of this class that you will need.
- Class MazeTest. This class provides a graphical interface to test your algorithm. This should be the main class of your project. The main method of this class:
  1. asks you for the size of the grid,
  2. displays an initial grid where all cells have been set to color BACKGORUND (see Fig. 1(a) which depicts a 5 by 5 grid), and
  3. allows you to edit it in order to indicate which cells are potentially considered to be part of a path.

  Regarding this last item, you may indicate which cells can be part of a path by clicking on them and changing their color to the color NON_BACKGROUND. It is the cells of color NON_BACKGROUND that can be part of a path. For example, in Fig 1(b) we see an example of cell selection, the cells in red may form part of a path, but the others do not participate. Once you have selected the cells that can be part of a path, you can press the “solve” button. This will call your Maze.findMazePath method and report the results. Fig. 1(c) shows the result of solving the maze. The cells in green form the path. The cells in black are ones which were visited while exploring for a solution. The ones in red were not visited.
  You can always start over by clicking on the “reset” button.

3 Problems

3.1 Problem 1

Implement a recursive algorithm

public boolean findMazePath(**int** x, **int** y)

that returns true if a path is found. When you click the button “solve”, the method MazeTest.solve calls the wrapper method Maze.findMazePath(), that in turn calls your algo-rithm with parameters x and y set both to 0. In order to implement your algorithm, take the following into consideration:

- If the current cell being analyzed falls outside the grid, then **false** should be returned since there is no possible path through that cell.
- If the current cell being analyzed does not have NON_BACKGROUND, then **false** should be returned since there is no possible path through that cell. To determine the color of a cell use getColor(**int** x, **int** y) of the class TwoDimGrid, which is the type of the data field maze.
- If the current cell being analyzed is the exit cell, then
  - The cell must be painted color PATH. For that you may use the recolor(**int** x, **int** y, Color c) method from the class TwoDimGrid, which is the type of the data field maze.
  - You should return **true**.
- Otherwise the current cell may be assumed to be part of the path and hence
  - It should be painted color PATH.
  - Each of the four neighboring cells must be explored to see if they too are part of a path (i.e. they are the cells where the path "continues"). This is where recursion takes place. In fact, multiple recursive calls, one for each neighbor.
  - If the result of exploring at least one of the neighbors is successful, then a path has been found. Otherwise, the cell is not part of a path: it is a dead end. Hence it must be marked so that it is not visited again. For that it must be colored to color TEMPORARY.

3.2 Problem 2

Implement a recursive algorithm

```
public ArrayList<ArrayList<PairInt>> findAllMazePaths(int x, int y)
```

by modifying the solution of Problem 1 so that a list of all the solutions to the maze is returned. Each solution may be represented as a list of coordinates. If there are no solutions, then the resulting list should have the empty list as only element. Some examples follow.

3.2.1 Examples

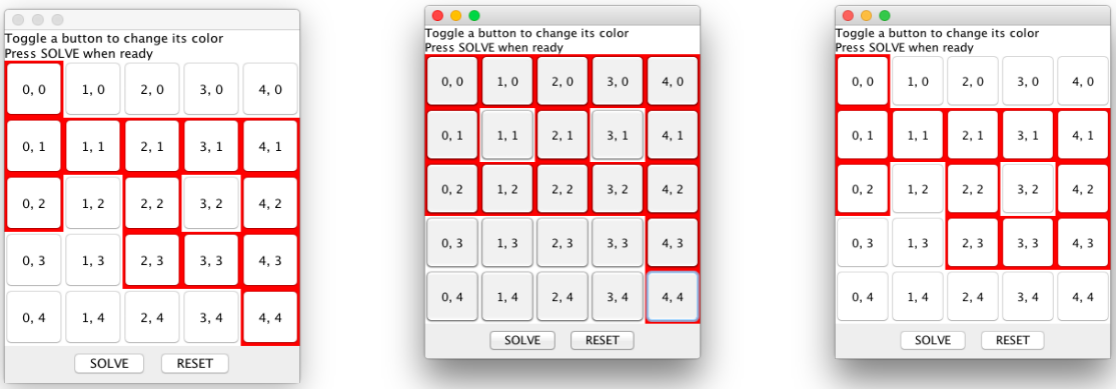
As an example, for the grid in Figure 2(a), it should report the following output since there are two paths to the exit:

```
[[(0 ,0) , (0 ,1) , (1 ,1) , (2 ,1) , (3 ,1) , (4 ,1) , (4 ,2) , (4 ,3) , (4 ,4)] ,
 [(0 ,0) , (0 ,1) , (1 ,1) , (2 ,1) , (2 ,2) , (2 ,3) , (3 ,3) , (4 ,3) , (4 ,4)]]
```

For the grid in Figure 2(b), it should report the following output:

```
[[(0 ,0) , (1 ,0) , (2 ,0) , (3 ,0) , (4 ,0) , (4 ,1) , (4 ,2) , (4 ,3) , (4 ,4)] ,
 [(0 ,0) , (1 ,0) , (2 ,0) , (2 ,1) , (2 ,2) , (3 ,2) , (4 ,2) , (4 ,3) , (4 ,4)] ,
 [(0 ,0) , (0 ,1) , (0 ,2) , (1 ,2) , (2 ,2) , (3 ,2) , (4 ,2) , (4 ,3) , (4 ,4)] ,
 [(0 ,0) , (0 ,1) , (0 ,2) , (1 ,2) , (2 ,2) , (2 ,1) , (2 ,0) , (3 ,0) , (4 ,0) , (4 ,1) , (4 ,2) , (4 ,3) , (4 ,4)]]
```

For the grid in Figure 2(c), it should report the following output since there are no paths to the exit:



(a) (b) (c)

Figure 2: More sample grids

```
[[]]
```

The class PairInt is a class you should define and which encodes pairs of integers that represent coordinates. The operations it should support are described below (copy should return a new copy of a PairInt)

PairInt
private int x private int y
public PairInt(int x, int y) public int getX() public int getY() public void setX(int x) public void setY(int y) public boolean equals(Object p) public String toString() public PairInt copy()

3.2.2 Hints

- Think about implementing the following helper method

  public void findMazePathStackBased(**int** x, **int** y, ArrayList<ArrayList<PairInt>> result, Stack<PairInt> trace)

  where:

  – (x,y) are the coordinates currently being visited

– result is the list of successful paths recorded up to now

– trace is the trace of the current path being explored

and then defining findAllMazePaths as:

```
public ArrayList < ArrayList < PairInt > > findAllMazePaths( int x, int y ) {
        ArrayList < ArrayList < PairInt > > result = new ArrayList < >();
        Stack < PairInt > trace = new Stack < >();
        findMazePathStackBased(0,0, result, trace
        ); return result;
}
```

- When the exit has been reached the contents of the stack should be copied into a list (for that you may use the addAll method of lists which also accepts stacks – any Collection in fact) and placed in result.
- The colors may be ignored in this exercise *except* for their use to avoid visiting nodes that have already been visited. For that only one color is necessary to mark a cell (choose PATH to mark cells that have been visited). In the recursive call you should mark the cell with color PATH before calling your helper function recursively on neighboring cells.
- In the recursive call, *after* returning from visiting the neighbors, you should *remove* the mark (by coloring the cell with the NON_BACKGROUND color) so that this cell may be visited again after backtracking.

3.3 Problem 3

Adapt **boolean** Maze.findMazePath() so that it returns the shortest path in the list of paths. The resulting method should be called

```
public ArrayList<PairInt> findMazePathMin(int x, int y)
```

4 Submission instructions

Submit a single zip file whose name is your surname through Canvas. It should contain two java files named Maze.java and PairInt.java plus **all the supporting files necessary to run them**. No report is required. Your grade will be determined as follows:

- You will get 0 if your code does not compile.
- We will try to feed erroneous and inconsistent inputs to all methods. All arguments should be checked.
- Partial credit may be given for style, comments and readability.