CS310 Fall 2015

Data Structures

Programming Assignment #3
A Maze Puzzle Game

50 points

Important Clarification

The grid coordinates follow the java conventions for a two dimensional array, which is `array[row][column]`. Thus the coordinate `[5][0]` means the sixth row, and first column.

Due Date/Time:

Your project is due on **Tuesday, November 3rd** at the beginning of class. Hardcopy is due on the same day, at the beginning of class.

For this assignment, you will write a Java program that finds the shortest route (if one exists) through a randomly generated maze. This problem can easily be solved using a stack and queue with the algorithm that is given below. You will use your stack and queue from project #2 to solve the problem.

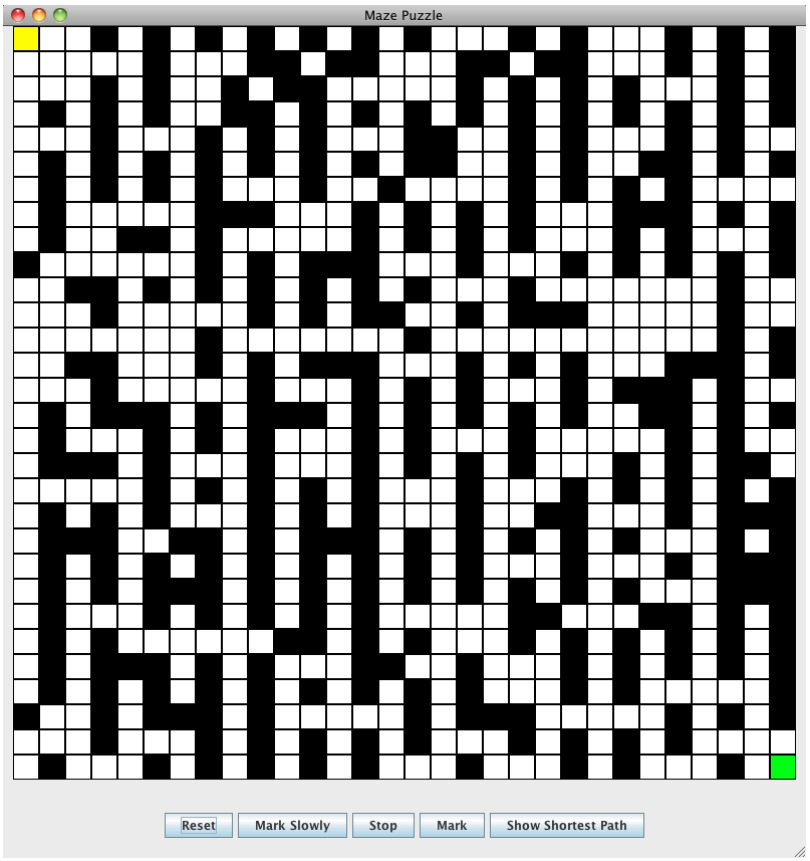
The maze is a two dimensional square grid. You will write a class named `MazeSolver` that interacts with a GUI (graphical user interface) written by your instructor which will show the maze and your solution.

Project Submission

As usual, you will submit your project by copying the java source code file into your `handin/` subdirectory. You **must** create a folder named `prog3` in your handin subdirectory, and place your files in **handin/prog3/**. Do ****NOT** create any folders in `handin/prog3`** Submit only `MazeSolver.java`; I will grade your project with my versions of your project #2.

Additionally, you will submit a printout of your MazeSolver.java file in class on the due date.

Puzzle Game Rules



The goal is to find the shortest route through the maze from entry to exit. The puzzle always begins at the top left cell (yellow) and terminates at the bottom right cell (green) of the puzzle grid. To move through the maze, you may go up, down, left or right one cell at a time. You may not move diagonally, nor may you move into a cell that is black. Since the obstructions (black cells) in the maze are randomly generated, some mazes will have no solution.

Note: A maze may have several optimum solutions--the minimum number of moves needed to traverse the maze. Your program will find only one shortest path. Your solution is correct if no shorter path exists through the candidate maze.

The graphical user interface contains a two dimensional grid, with five buttons underneath. These five buttons behave as follows:

- **Reset** Clears the grid and creates a new puzzle.
- **Mark Slowly** Marks each available (and reachable) cell with its distance to the starting cell. There is a delay so that you can see the workings of the breadth first algorithm. The cells being flagged temporarily turn red.
- **Stop** Halts the breadth first search cell flagging operation. Only useful for stopping a "Mark Slowly" operation. Just stops, does not reset the puzzle. This is so you can examine the actions that have occurred. You cannot restart the operation, but must manually reset before continuing.
- **Mark** Runs the breadth first algorithm, and marks the distance from the origin in each reachable cell.
- **Show Shortest Path** Turns the background color of the cells in the shortest path to a light blue color.

Algorithms

Solving the puzzle is a two step process. First, you must flag each GridCell that is reachable with its distance from the origin. Then starting at the exit point and working back toward the origin, you select the available cell with the minimum distance. For the first part, you will use a breadth first traversal algorithm to visit and mark each cell:

```
enqueue cell(0,0)
while( the queue is not empty ) {
    dequeue a GridCell from the queue.
    mark each adjacent neighboring cell and enqueue it
    }
```

For the second part, begin at the green exit point, then check each adjacent neighboring cell, and push the one with minimum distance onto the stack. The stack then contains all of the cells in the shortest path:

```
distance = terminalCell.getDistance();
if(distance == -1) return false;  // unreachable, puzzle has no solution
push terminal cell onto the stack
while(distance != 0) {
    get distance from each cell adjacent to the top of the stack
    select the cell with smallest distance and push on the stack
    }
while( stack is not empty ) {
    pop grid cell off the stack and mark it
    }
```

Program Files

Your project will have the following files:

- `MazeGrid.java` The graphical user interface, written by your instructor using Java Swing.
- `GridCell.java` The representation of a single cell in the MazeGrid.
- `MazeSolver.java` This is the class you must write to solve the maze puzzle.
- `LinearListADT.java` The list interface from program #2.
- `LinearList.java` The list implementation from program #2.
- `Stack.java` The stack implementation from program #2.
- `Queue.java` The queue implementation from program #2.

Thus, for this assignment you will write a single class, `MazeSolver`, and reuse the three classes, plus the interface from project #2.

You may get a copy of the two new files for this project by copying them from your instructors account to your account on rohan as follows:

```
rohan ~[]% cp ~riggins/MazeGrid.java .
rohan ~[]% cp ~riggins/GridCell.java .
```

Methods

Because your MazeSolver must interact with GridCell and MazeGrid, the methods and signatures must be standardized. Below are method signatures:

`GridCell` This class encapsulates information about each cell in the puzzle. **IMPORTANT** You should *never* call this class's constructor, nor create a new GridCell.

- `public void setDistance(int distance)`
  Sets the distance of this cell from the origin [0,0].
- `public int getDistance()`
  Returns the distance of this cell from the origin [0,0]. If the distance has not been set, returns -1.
- `public int getX()`
  Returns the X coordinate of this cell.
- `public int getY()`
  Returns the Y coordinate of this cell.
- `public boolean wasVisited()`
  Returns true if this cell has been visited, otherwise false.

`MazeGrid`

- `public MazeGrid(MazeSolver solver, int dimension)`
  You will create a MazeGrid instance in your MazeSolver constructor. The first argument will be *`this`*. You are passing a reference to your MazeSolver to the MazeGrid. When a button is pressed in the GUI, the MazeGrid class will call a method in your MazeSolver. The second parameters is the number of cells across and down. Depending on the resolution of your monitor, values in the range 25 .. 50 are reasonable.
- `public GridCell getCell(int x, int y)`
  Returns a reference to the GridCell at coordinates x,y.
- `public boolean isValidMove(GridCell cell)`
  Returns true if the move is valid. Valid means the XY coordinate are within the grid, and the background is not black. Does NOT detect if the move is legal since it does not maintain history. A move such as to a diagonal cell, or a move greater than 1 cell away may be valid if the cell is within the grid, and does not have a black background. However, such moves are illegal. This method does **NOT** detect illegal moves, only invalid ones.
- `public void markDistance(GridCell cell)`
  Sets the distance in the given GridCell, and displays that distance on the corresponding cell in the GUI.
- `public void markMove(GridCell cell)`
  This method is used to designate a cell in the shortest route. It turns the background color blue.

`MazeSolver` This is the class that you must write for the assignment. You must have all of the following public methods.

- `public MazeSolver(int dimension)`
  The constructor. Takes a single argument, the number of rows and columns in the grid. Suggested values are 25 .. 50.
- `public void mark()`
  This method runs the breadth first traversal, and marks each reachable cell in the grid with its distance from the origin.

- `public boolean move()`
  Does part two, indicates in the GUI the shortest path found.
- `public void reset()`
  Reinitializes the puzzle. Clears the stack and queue (calls makeEmpty()).

Additional Details

- As usual, your project must compile and run on rohan to receive any credit for the assignment.
- Be sure to put your NAME and rohan CLASS ACCOUNT number at the beginning of your source code file.
- You must use the algorithms given in the assignment. A solution that does not use these algorithms will receive no credit.
- You may not use any arrays or data structures in your MazeSolver class other than a single stack and a single queue.
- Late programs will be accepted with a 5% per day penalty. The timestamp on your files in the `handin/prog3` subdirectory will be used to determine when your project was submitted.
- You may import only `data_structures.*` in your MazeSolver class. You must use your project files from project #2 for this assignment. Thus, if you have had trouble finishing project #2, you must complete it. You will not be able to pass the course without completing it.

Cheating Policy:

There is a zero tolerance policy on cheating in this course. You are expected to complete all programming assignments on your own. Collaboration with other students in the course is not permitted. You may discuss ideas or solutions in general terms with other students, but you must not exchange code. (Remember that you can get help from me. This is not cheating, but is in fact enouraged.) I will examine your code carefully. Anyone caught cheating on a programming assignment or on an exam will receive an "F" in the course, and a referral to Judicial Procedures.