Homework Assignment 3

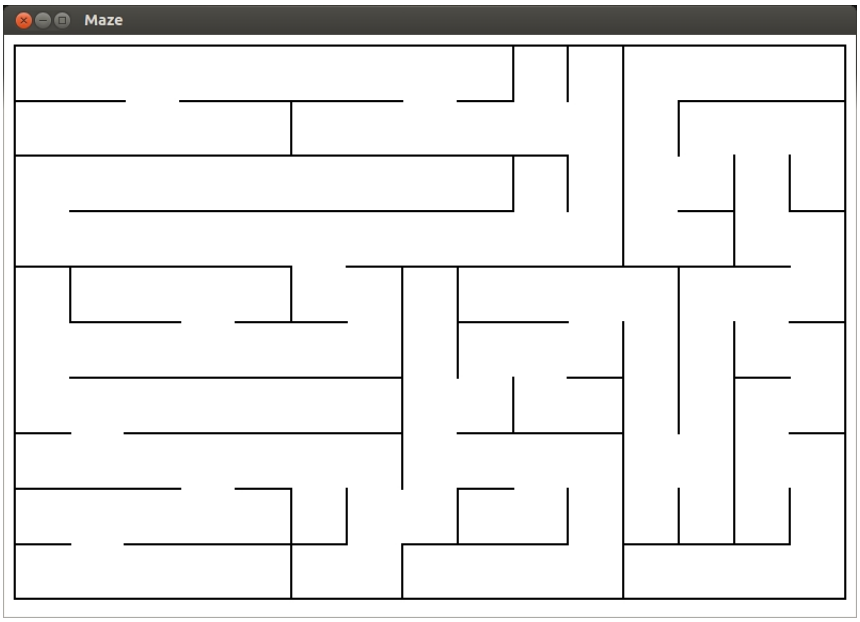
CS 0445 (Sprint 2013) — Data Structure

Due Friday November 14, 2014 at 11:59pm

The purpose of this project is for you to practice recursion and backtracking by creating a good maze using recursion and solve a maze using backtracking and recursion.

Part I: The Class `Maze`

For this project, you should imagine that a maze is an object. It is an object that can be constructed by a width (greater than or equal to 1) and a height (also greater than or equal to one). Once a maze is constructed, it consists of a number of chambers of size one by one organized as two-dimensional matrix. Each chamber may or may not have north wall, east wall, south wall, and west wall. Each chamber can be identify by a row (start at 0) and a column (start at 0). An example of a maze size 15 by 10 is shown below.



In the above maze, the top-left chamber is the chamber at row 0 and column 0. It consists of a north wall, a south wall, and a west wall. The bottom-right chamber is the chamber at row 9 and column 14. It consists of an east wall and a south wall. Note that the outer-most wall for every maze must be closed (no opening).

The class `Maze` allows users to construct a maze of any size (any width and height). The width is the number of chambers in each row and the height is the number of chambers in each column. Note that the width and the height must be greater than or equal to one and they do not have to be equal.

Constructor

The class `Maze` should have only one constructor and the signature should be as follows:

```
public Maze(int width, int height)
```

This constructor constructs a maze where the width of the maze which is the number of chambers in each row and the height of the maze which is the number of chambers in each column. To create a maze, you **MUST** use the algorithm discussed in class using recursion as follows:

1. Randomly pick a point in the maze
2. Create four walls, north, east, south, and west.
3. Randomly pick three out of four walls
4. For each picked wall, randomly pick a position to create an opening
5. Four new smaller chambers are created after you create walls, use the same process for each chamber and repeat until the width or the height of the chamber is 1.

Public Methods

For the class `Maze`, you must have the following public methods:

1. `int getWidth()`: This method should return the width of this maze.
2. `int getHeight()`: This method should return the height of this maze.
3. `boolean isNorthWall(int row, int column)`: This method should return true if the chamber identified by `row` and `column` contains north wall. Otherwise, return false.
4. `boolean isEastWall(int row, int column)`: This method should return true if the chamber identified by `row` and `column` contains east wall. Otherwise, return false.
5. `boolean isSouthWall(int row, int column)`: This method should return true if the chamber identified by `row` and `column` contains south wall. Otherwise, return false.
6. `boolean isWestWall(int row, int column)`: This method should return true if the chamber identified by `row` and `column` contains west wall. Otherwise, return false.

For example, return values of each method of the maze shown in previous page is shown below:

Method	Return Value
`getWidth()`	`15`
`getHeight()`	`10`
`isNorthWall(1,1)`	`true`
`isEastWall(2,0)`	`false`
`isSouthWall(2,2)`	`true`
`isWestWall(3,3)`	`false`

The Test Class

The class `MazeTester.java` is provided. Note that a maze is a complex structure. Thus, there is no straightforward way to test a maze. However, the `MazeTester` will generate a maze with random width and height and test the following:

- Test the method `getWidth()`
- Test the method `getHeight()`
- Test that the east most wall and the west most wall contain no opening
- Test that the north most wall and the south most wall contain no opening
- Test all adjacent walls between every two chambers
- Test that there must be exactly one long wall inside the maze with only one opening
- Test closed rectangular or square chambers

Note that you can also view your maze using another provided program `MazeFrame.java`. The `main` method of this program is shown below:

```
public class MazeFrame
{
   public static void main(String[] args) throws InterruptedException
   {
      int width = 15;
      int height = 10;
      JFrame frame = new JFrame();
      Maze maze = new Maze(width, height);
      ArrayList<Pair<Integer,Integer>> solution = new ArrayList<Pair<Integer,Integer>>();
      MazeComponent mc = new MazeComponent(maze, solution);
      frame.setSize(800,800);
      frame.setTitle("Maze");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.add(mc);
      frame.setVisible(true);

      //solution.add(new Pair<Integer,Integer>(0,0));
      //Thread.sleep(1000);
      //solveMaze(solution, mc, maze, ...);
      //mc.repaint();
   }
```

The code above can be use to simply view your maze. To change the width and the height of your maze, simply change the values of variables `width` and `height`. **Note** that the last four lines of the above `main` function are commented out. Do not uncomment them until you are working on Part II.

Part II: Solve Maze

After you finish creating a maze, the next part is to solve it using backtracking and recursion. For this project, we will always assume that the starting point is at the top-left corner chamber (chamber at row 0 and column 0) and the finishing point is at the bottom-right chamber (chamber at row `height - 1` and column `width - 1`). A solution or a route will be an `ArrayList` of pairs (`Pair<Integer,Integer>`) named `solution`. The first item in the list is the starting chamber which is always be a pair (0,0). The last item in the list will the end of a route. Note that a line in the `main` function of the program `MazeFrame` is as follows:

```
//solution.add(new Pair<Integer,Integer>(0,0));
```

After you uncomment the above line, it is the same as we set the starting point to the chamber at row 0 and column 0. **Note** that the class `Pair` is provided. You **must** use the provided class `Pair` (`Pair.java`) because other program (`MazeComponent.java`) also use this class `Pair`.

What you have to do is to finish up the **recursive** method `solveMaze()` which can be found in `MazeFrame.java`. Note that one of the arguments of the method `solveMaze()` is the `solution`. This allows the method `solveMaze()` to `add` pairs to the `solution` or remove `pairs` from the solution base on backtracking. The method `solveMaze()` should stop making recursive call when the last item is the finishing chamber.

Note that the signature of the method `solveMaze()` is incomplete. Your job is to figure it out what do you need to make the method `solveMaze()` able to solve a maze using backtracking. However, you must keep the following components:

- The method `solveMaze()` must return a boolean. This will allow the method `solveMaze()` to send a signal back to its caller whether it **can** reach the finishing point or not.
- The first argument must be `ArrayList<Pair<Integer,Integer>> solution` as explained earlier about the purpose of the variable `solution`.
- The second argument must be `MazeComponent mc` because this will allow your method `solveMaze()` to show an animation of backtracking.
- The third argument must be `Maze maze`. Obviously you need a maze to solve.

So, do not modify the return value and the first three arguments. You can add as many arguments as you need.

To see an animation of backtracking, **EVERY TIME** you **add** a new pair or **remove** a pair from the `solution`, you should call `mc.repaint();` and follows by `Thread.sleep(sleepTime);` as follows:

```
:
solution.add(...) // Add a new chamber (moving forward)
mc.repaint();
Thread.sleep(sleepTime);
:
solution.remove(...) // Remove a chamber (backtrack)
mc.repaint();
Thread.sleep(sleepTime);
:
```

This will allow the `MazeComponent` to paint a new route. By the way, do not forget to uncomment the last four lines in the `main` function.

Hints

An example of solving the Eight Queens problem using backtracking and recursion is provided in the CourseWeb. You can use the program `EightQueens.java` as a guideline how to achieve backtracking using recursion.

Due Date and Submission

This assignment is due on Friday November 14, 2014 at 11:59pm. No late submission will be accepted. All source files (`Maze.java`, and `MazeFrame.java` must be zipped into a single .zip file and submitted to the CourseWeb under Project 3.