CSCI 1933 Project 4

Stacks, Queues, and Mazes

Due Date: April 24, 2020

Instructions

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

- **Due:** The project is due on **April 24, 2020** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment near the top of all files you submit. Failure to do so may result in a penalty.
- **Submission:** Submit a `zip` archive on Canvas containing all your `java` files. This includes every file listed in the "Files Given" subsection below. You are allowed to change or modify your submission, so submit early and often, and *verify that all your files are in the submission.*

  Failure to submit the correct files will result in a score of zero for all missing parts. Late submissions and submissions in an abnormal format (such as `.rar` or `.java`) will be penalized. Only submissions made via Canvas are acceptable.
- **Partners:** You may work alone or with *one* partner. Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero. Ensure all code shared with your partner is private.
- **Code:** You must use the *EXACT* class and method signatures we ask for. This is because we may use a program to evaluate your code. Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 8, which is installed on the CSE Labs computers.
- **Questions:** Questions related to the project can be discussed on Piazza in abstract. This relates to programming in Java, understanding the writeup, and topics covered in lecture and labs. **Do not post any code or solutions on the forum**. Do not e-mail the TAs your questions when they can be asked on Piazza.
- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately* through the TA alias: "csci1933-spring2020tas@umn.edu".
- **README:** Make sure to include a README.txt in your submission that contains the following information:
  - Group member's names and x500s
  - Contributions of each partner (if applicable)
  - Any assumptions
  - Additional features that your project had (if applicable)
  - Any known bugs or defects in the program

IMPORTANT: You are NOT permitted to use ANY built-in libraries, classes, ect... besides java.util.Random. Double check that you have NO import statements in your code, except for those explicitly permitted.

Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.
- Use effective and standard indentation.
- Use descriptive names for variables. Use standard Java style for your names: `ClassName`, `functionName`, `variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments.
  `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.
- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct. In general, functions ought to terminate in under 5 seconds for any reasonable input.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

If you are confused about the style guide, please talk with a TA.

Credits

This wrieup incorporates ideas from the United States Naval Academy: https://www.usna.edu/Users/cs/aviv/classes/ic312/f16/project/01/project.html While this link no longer works, it was used as reference for the creation of this project.

It also implements a search algorithm to create mazes inspired by the following Wikipedia article: https://en.wikipedia.org/wiki/Maze_generation_algorithm#Depth-first_search

While we are pulling ideas from this, do not follow any instructions found on the web links.

1 Introduction

In this project, you will use Stacks and Queues to solve basic mazes. A maze is a network of paths designed so there is at least one path from the entrance of the maze to the exit of the maze. You will be using a queue structure to find this correct path through the maze, and a stack structure to randomly generate new mazes.

1.1 Files Given

Along with the project write-up, you will be given the following files:

- **MyMaze.java** The main java class for this project
- **Cell.java** A helper class for MyMaze
- **NGen.java** - Both the queue and stack structure will utilize this generic node class
- **Q1.java** - An interface for a generic queue
- **Q1Gen.java** - An implementation of a generic queue
- **StackGen.java** - An interface for a generic stack
- **Stack1Gen.java** - An implementation of a basic stack

You will not change any of the files given except for the class called `MyMaze`. All other files are provided with everything you should need.

1.2 Stack and Queue data structures

Both the stack and queue data structures provided have basic functionality. Take a look at both classes to ensure you know how to instantiate and use them properly. They should be similar to the examples you have seen in lecture.

2 Cell

You do not need to change the Cell class, but this section describes how a cell works. The Cell class has the following attributes:

`boolean visited` – true if this cell has been visited, false otherwise

`boolean right` – true if a right boundary, false if an open right side

`boolean down` – true if a lower boundary, false if an open bottom

Each of these attributes have setter and getter functions respectively. The Cell constructor initializes the cell to have walls on the right and bottom. When we say a cell has been visited, assume we are referring to the `visited` attribute unless otherwise noted. The purpose for each attribute will become evident in the MyMaze section below.

3 MyMaze

The `MyMaze` class is where the majority of the work will take place. There will be three main functions for the maze class:

```
public static MyMaze makeMaze(int rows, int cols);
public void printMaze(boolean path);
public void solveMaze();
```

More detail about each function will be found in their respective sections. The `MyMaze` class will have one main attribute: `Cell[][] maze`. The `maze` attribute will be how the maze is represented. This attribute will have the following properties:

- The start of the maze will always be the upper left: `maze[0][0]` (open on the left)
- The end of the maze will always be in the lower right: `maze[rows-1][cols-1]` (open on the right)
- All Cells on the "top" (row 0) have an implicit top boundary
- All cells of the "left" (column 0) have an implicit left boundary
- Once initialized, the maze should have a path from the beginning to the end of the maze.

For example, a maze of size 5x20 that could be generated is shown in Figure 1.

3.1 Constructor

The `MyMaze` constructor should have the following signature: `public MyMaze(int rows, int cols);` It should just instantiate the `maze` attribute: `maze = new Cell[rows][cols]` and create a new cell object for each index: `maze[i][j] = new Cell()`.

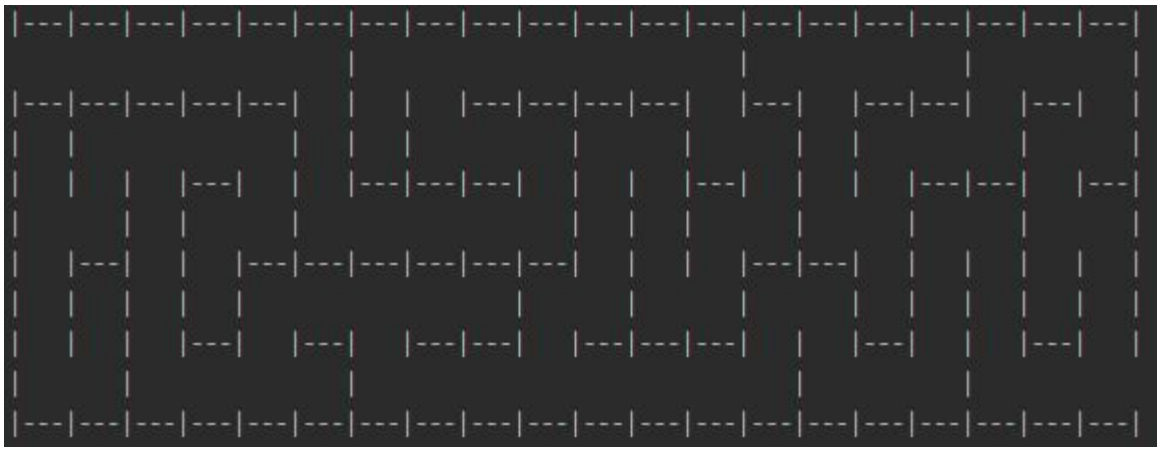

Figure 1: 5x20 Maze; Path = False

3.2 makeMaze

This function should instantiate a new `MyMaze` object, generate the maze, and then return the new `MyMaze` object. This will build a maze from scratch by utilizing a stack. By choosing randomly which direction to go from a particular cell, and visiting every cell, there should be a path from the entrance to every cell, including the exit. You can do this by implementing the following search algorithm:

- Initialize a stack with the start index {0, 0}. Mark this cell (`maze[0][0]`) as visited.
- Loop until the stack is empty:
  - Get the top element off the stack but do not remove it.
  - Choose a random neighbor to the corresponding cell that has not been visited and do the following:
    - Add the neighbor's index to the stack.
    - Mark the neighbor as visited.
    - Remove the wall between the current cell and the neighbor cell.
  - If the current cell does not have any un-visited neighbors, then it is a dead end. Pop the corresponding index from the top of the stack.

where a neighbor is defined as a cell that is either horizontally or vertically next to reference cell. The last thing the `makeMaze` function should do before returning is reset the `visited` attribute of maze cells to `false`.

3.3 printMaze

This function will print a visual representation of the maze to the terminal. There are different ways that you can represent a maze, but one of the simplest is to use vertical bars | as vertical

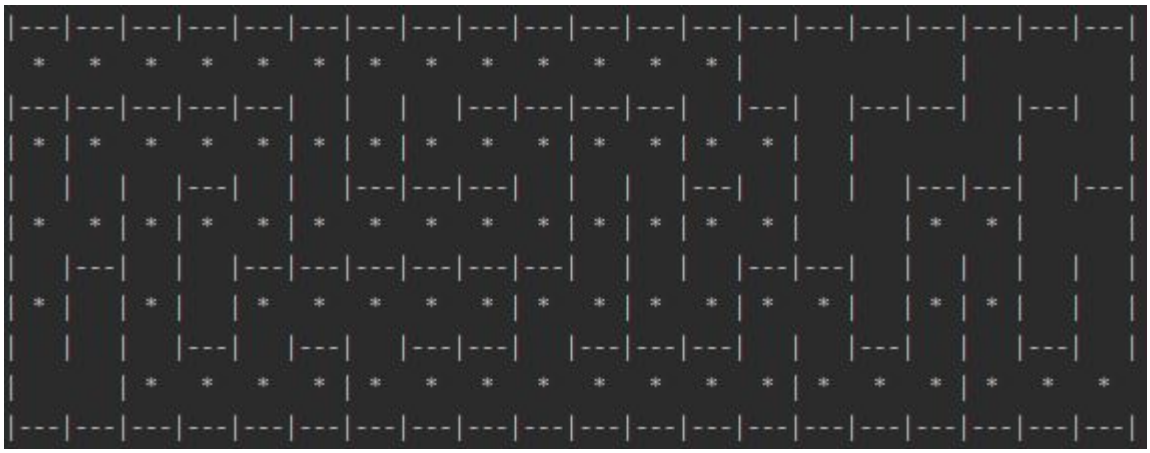

Figure 2: Path = True

borders, and dashes $---$ as horizontal borders. You can use whatever characters you want, but make sure the output is reasonable. You can reference figure 1 as a good example.

If the `path` parameter is `true`, then you must print an asterisk '$*$' in every cell where the `visited` attribute of the cell is `true`. Otherwise, if the `path` parameter is `false`, then do not print an asterisk '$*$' in the center of the cell. The reason for this is so we can visualize the path that the `printMaze()` function took when solving the maze.

Figure 1 is what may be printed to the terminal for a maze of size 5x20 when the parameter `path =false` is passed in. Figure 2 is what is what may be printed to the terminal for a maze of size 5x20 when the parameter `path=true` is passed in.

You are not restricted to how you print off the maze, but it may be easiest to print off the maze by going though each row one-at-a-time. There may be two special cases depending on how you implement the algorithm. You may need to remove the walls for the entrance and exit on the border of the maze separate from the loop.

By the end of this function, you should have printed the entire representation of the maze. Do **not** reset the `visited` attribute of the cells.

3.4 solveMaze

To solve a maze, we will use a queue to test all possible paths. The algorithm should work as follows.:

- Initialize a queue with the start index {0, 0}
- Loop until the queue is empty:
  - Dequeue the front index of the queue and mark the corresponding cell's `visited` attribute as `true`.

- If the current cell is the finish point (i.e. the index {rows-1, columns-1}), then break. The maze has been solved.
- Enqueue all reachable neighbors that are un-visited.

At the end of the function, call the printMaze function with `path=true`. An example of a solved maze is figure 2.

3.5 Testing

You are free to do any testing you see fit. You will know the functions are being generated correctly if there exists a path from the entrance cell to every cell of the maze, including the exit cell.

4 Grading Rubric

The following weight will be given to each category:

- Code Style: 10%
- a README with the required information: 5%
- The stack and queue structures are used in the correct functions: 5%
- `makeMaze` is working and implements the given algorithm: 30%
- `printMaze` is correctly outputting a representation of the maze: 25%
- `solveMaze` is working and implements the given algorithm: 25%