TCSS 342 - Data Structures
Extra Assignment 10 - Maze Generator

Version 1 (May 26th, 2022)

Time Estimate: 8-12 hours

Learning Outcomes

The purpose of this assignment is to:

- Build your own graph data structure.
- Decide on the best implementation choices for your purpose.
- Implement a depth first search of a graph.
- Visualize your graph.

Assignment

Many gaming applications require 2-dimensional n by m mazes with no cycles (the no cycles requirement creates a class of tough mazes with a single unique solution). These mazes are then the target of maze solving games or used by maze solving algorithms. One problem is generating a large library of mazes to be used by the games or to test the algorithms. Of course, algorithms exist to generate random mazes. It will be our goal to implement one of these algorithms.

To generate a 2-dimensional n by m maze with no cycles it is helpful to think of a graph with n*m nodes arranged in n rows of m nodes. Each node is then connected to the nodes above, below, to the left and to the right, if they exist, and no other nodes. Let's call this special type of graph G(n,m) for grid. (These types of graphs are often called "grids" or more generally "lattices".) A 2-dimensional n by m maze with no cycles is a *spanning tree* on this graph. Our goal, then, is to generate a *random spanning tree* of G(n,m). To do so we have several options.

We can use a depth-first search, a breadth-first search, Prim's algorithm, Kruskal's algorithm, or come up with our own method. You can find these methods discussed here. My recommendation is depth-first search (since you use this to solve the maze too) or Kruskal's algorithm (which makes nice random mazes).

I have a JavaScript demo of this in my reinforcement learning demo here. Press "New Maze" to see the new maze generated. Press "New Agent" to watch a robot try to solve the maze (this is beyond the scope of this class, but it's fun to watch).

To complete this you will implement one public class:

- Maze

Formal Specifications

Maze
-width : int -height : int
+Maze(width : int, height : int): -buildGraph() -buildMaze() -solveMaze() +toString() : String

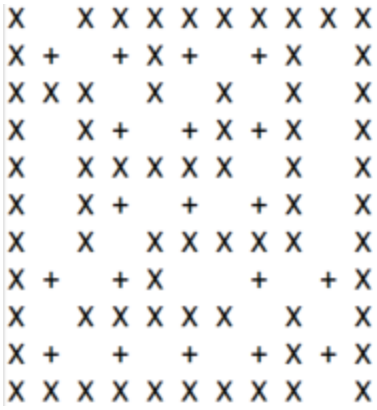
Field summary

- `width` - The width of the maze.
- `height` - The height of the maze.

You will need other fields in your maze class, but those details are left up to you. You may implement an array based graph (adjacency matrix) or a linked graph (adjacency lists). Or you may use an entirely different implementation. For instance, I used an array list of strings.

Method summary

- `buildGraph` - This method builds the initial graph G(width,height).
    - The implementation details are up to you.
- `buildMaze` - This method builds the maze from the graph G(width, height).
    - The choice of algorithm is up to you.
    - My recommendation is depth-first search as it can solve the maze at the same time as building it.
- `solveMaze` - This method solves the maze once built by finding the single path from the entrance to the exit.
    - Note: I hard code the entrance in the top-left and exit in the bottom-right. You may do the same or use another method to place the entrance and exit on opposite sides of the maze.
    - Depth-first searching for the exit from the entrance is the best technique to solve the maze.
    - If you solved the maze while you built the maze you can omit this method or make it empty or trivial.
- `toString` - Displays the maze in a readable way
    - I use Xs to represent walls and +s to represent the solution path. Here is what my text display looks like for a 5x5 maze:

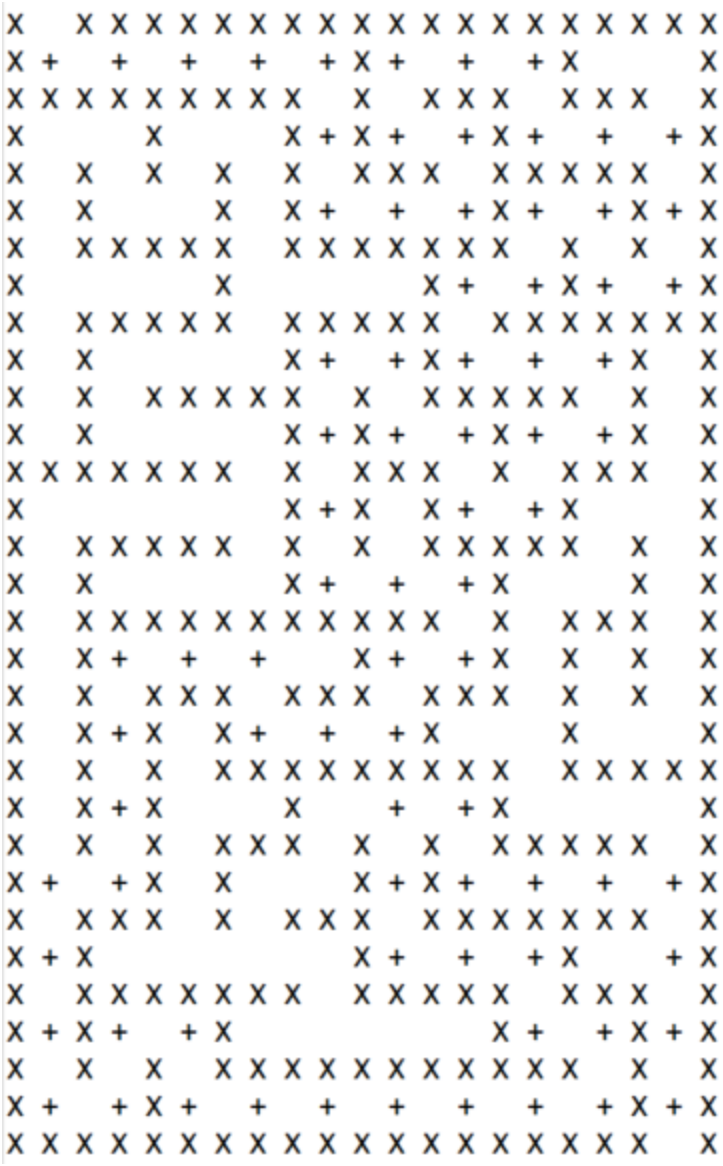


- **(Optional)** Instead of a text display you can implement a graphical display. This display should open a window and draw a maze and display the solution.

Testing

It is important that you test your code to ensure it works on all potential inputs. Please do this in a separate Main class, and without using additional tools (like JUnit). You will not submit your test code. Instead, your data structures will be tested by code developed by the instructor and/or grader. This code will not be provided to you, as you should test your code before submitting it. If your code fails our tests we will let you know which tests it fails so you can find and correct your errors.

Here is the output from my solution on a 10x15 maze:



Submission

You will submit a .zip file containing:

- Maze.java - Your implementation of Maze.
- Any other classes necessary for your solution (i.e. MyLinkedList.java or MyArrayList.java or MyHashTable.java).

Grading Rubric

In order to count as complete your submission must:

1. Generate a random maze of the given size according to the specifications above.
2. Solve the maze by finding the unique solution.
3. Display the maze and the solution.

Reminder: Incomplete assignments can always be corrected and resubmitted. If they are completed within 7 days of the due date they will count as late and after that period they will count as missed. Please review the grading matrix for the number of permitted late and missed assignments.