1. Introduction:

1.1 Overview:

The main idea of this project revolves around various path finding algorithms, such as BFS, DFS, Greedy, A*, and Dijkstra's algorithm. The purpose is to visualize how these algorithms work internally to provide a better understanding for anyone who has difficulty grasping the various algorithms.

The software implements maze generation. The program will allow for a user to generate a grid of a range of sizes, draw their own grid in real-time, determine animation speed and generate random mazes for use of ease. Using Dijkstra's algorithm, the software graphically demonstrates what nodes the robot is looking at, obstacles in the way of the robot and backtracking once the path is found; a highlight of the said algorithm determined. The user will have the option of selecting different algorithms the robot will search the maze with. This will allow us to implement a time complexity feature which displays a graph allowing the user to compare time complexities of different algorithms.

Regarding the graphical side of things, the program will produce a 2D maze which the robot will search on. Java will be used in the project along with NetBeans. The program will be runnable on windows

1.2 Glossary:

- Abstract Data Type (ADT) – an ***abstract data type*** (***ADT***) is a mathematical model for data types, where a data type is defined by its behaviour from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations.
- Algorithm – is a well-defined procedure that allows a computer to solve a problem.
- Traversal – is a form of graph traversal and refers to the process of visiting each node in a tree data structure, exactly once.
- Tree – ***a tree*** is an abstract data type (ADT) that simulates a hierarchical ***tree*** structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
- Breadth-First Search (BFS) – is an algorithm for traversing or searching graph data structure. It starts at the tree root and explores the neighbour nodes first, before moving to the next level neighbour.
- Depth-First Search (DFS) – is an algorithm for traversing or searching a tree or graph data structures. One starts at the root, selecting some arbitrary node as the root in the case of a graph, and explores as far as possible along each branch before backtracking.

- Heuristic – is used in artificial intelligence as a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximation when classic methods fail to find an exact solution.
- Local Optimum – is the best solution to a problem within a small neighbourhood of possible solutions.

- Greedy Algorithm – is an algorithm that follows the problem-solving heuristic of making the locally optimum choice at each stage with the hope of finding a global optimum.

- A* Algorithm – is an algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently directed path between multiple points, called nodes.

- Dijkstra's Algorithm – is an algorithm for finding the shortest paths between nodes in a graph.

- Real Time - the actual time during which a process occurs.

- Time Complexity – is a measure of the time that it takes for an algorithm to complete its tasks.

2. System Architecture

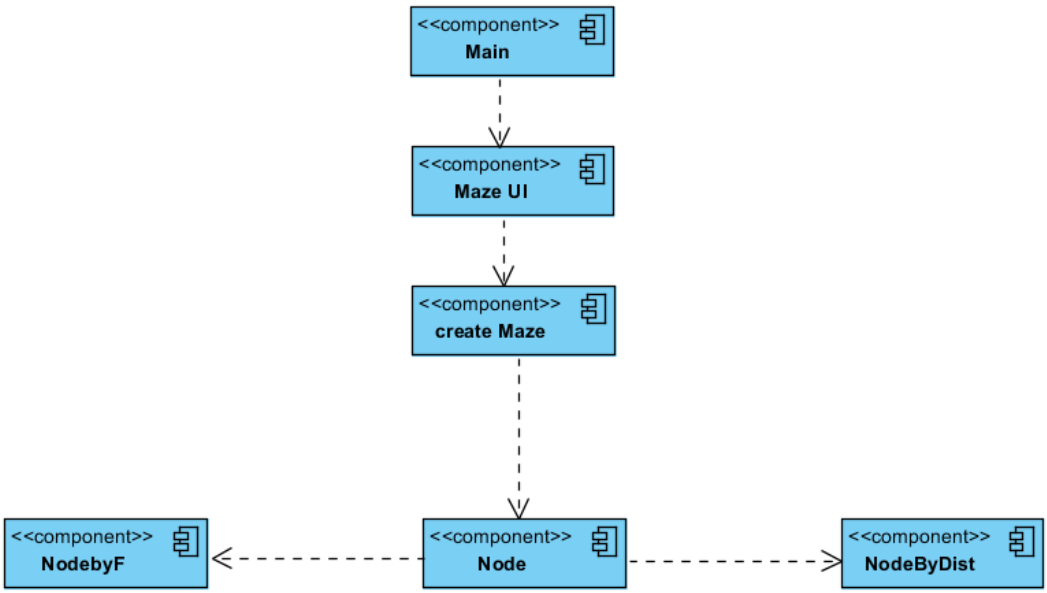


Figure 2.

2.1 Main

Responsible for making the program runnable, creating the window and initializing size. This includes a content pane “Maze UI” which houses all UI elements.

2.1 Maze UI

Houses all UI elements and all functions required for the program to work.

2.2 Create Maze

Handles the creation of the maze.

2.3 Node

This component is responsible for making each individual cell(node) in the grid possess certain attributes and be accessed by the robot for traversal.

2.4 Node By F

The Node is dependent on this as a function (F) needs to generate a value for heuristic search. Specifically, it compares F values between Nodes.

2.5 Node by Distance

The Node is dependent on this as a distance is needed to calculate the Manhattan heuristic and to create successor nodes. Specifically, it compares Distance values between Nodes.

3. High level Design

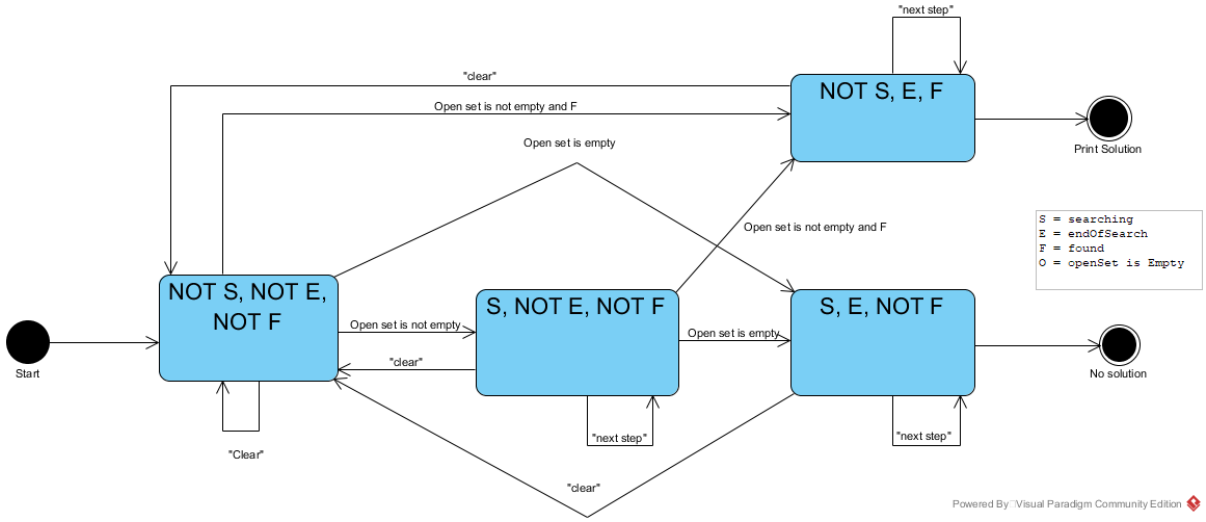
To describe the design at a high level we found it best to do so with a state transition diagram. This is due to the fact our system is best described as being in one of several possible states.

The maze solver's current state when run is a combination of three parameters. Which we use as flags in our code:

1. Whether the search is in progress(searching).
2. Whether the search has reached the end (endOfSearch).
3. Whether the target has been found (found).

This means there are a total of eight combinations of values(teach value can be true or false so 2^3) that can hold the above three parameters. However, out of these eight combinations only four of them allow for possible search states and this

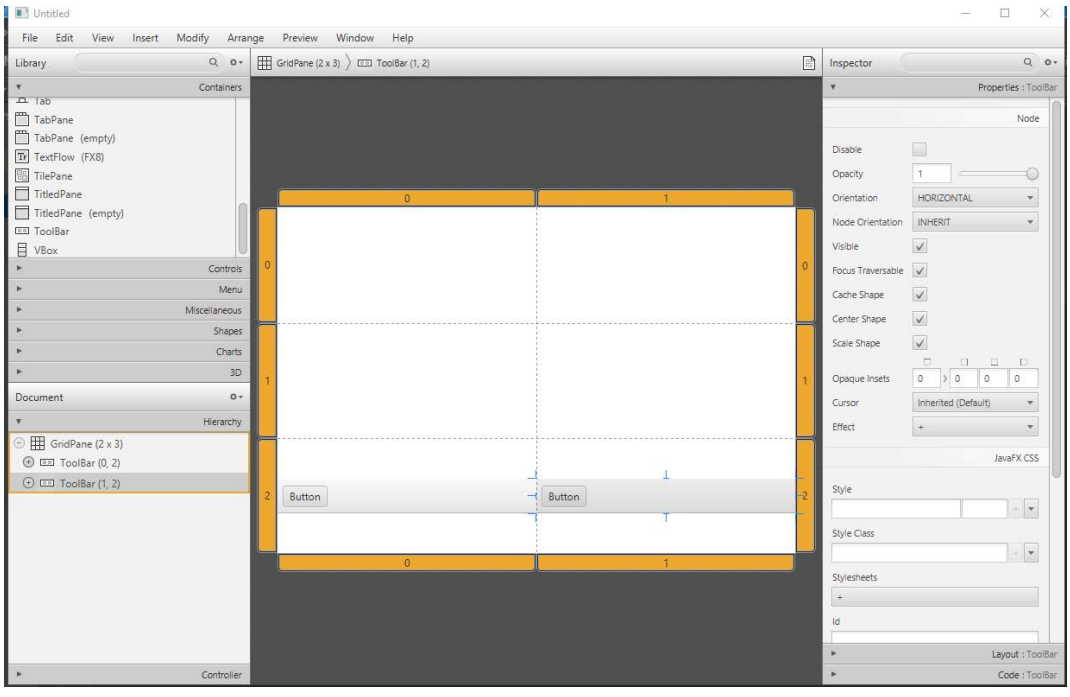
is shown in the below state diagram (figure 3):



4. Problems and Resolution

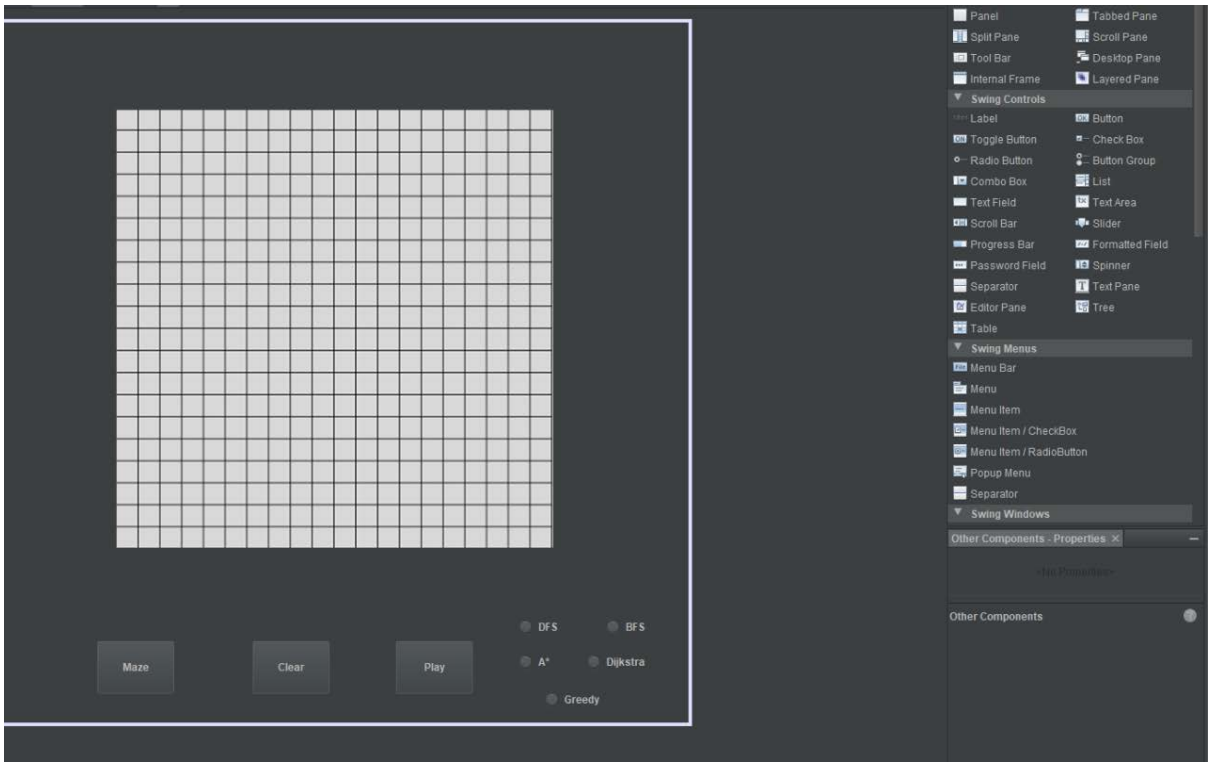
UI problem:

Our original design featured a “spinner” which allowed the user to decide the size of the grid but we later removed this problem. However, before we made this decision we used Java scene builder to build the UI for us (and generate the code automatically):



However, upon building the UI we found out it didn't have a "spinner" so we forced to find another solution.

This how we came across Java Swing:



We found this to be much more intuitive because the scene builder aspect was built into the IDE rather than having to install a separate application to handle it.

This was the reason we switched from using IntelliJ to Netbeans.

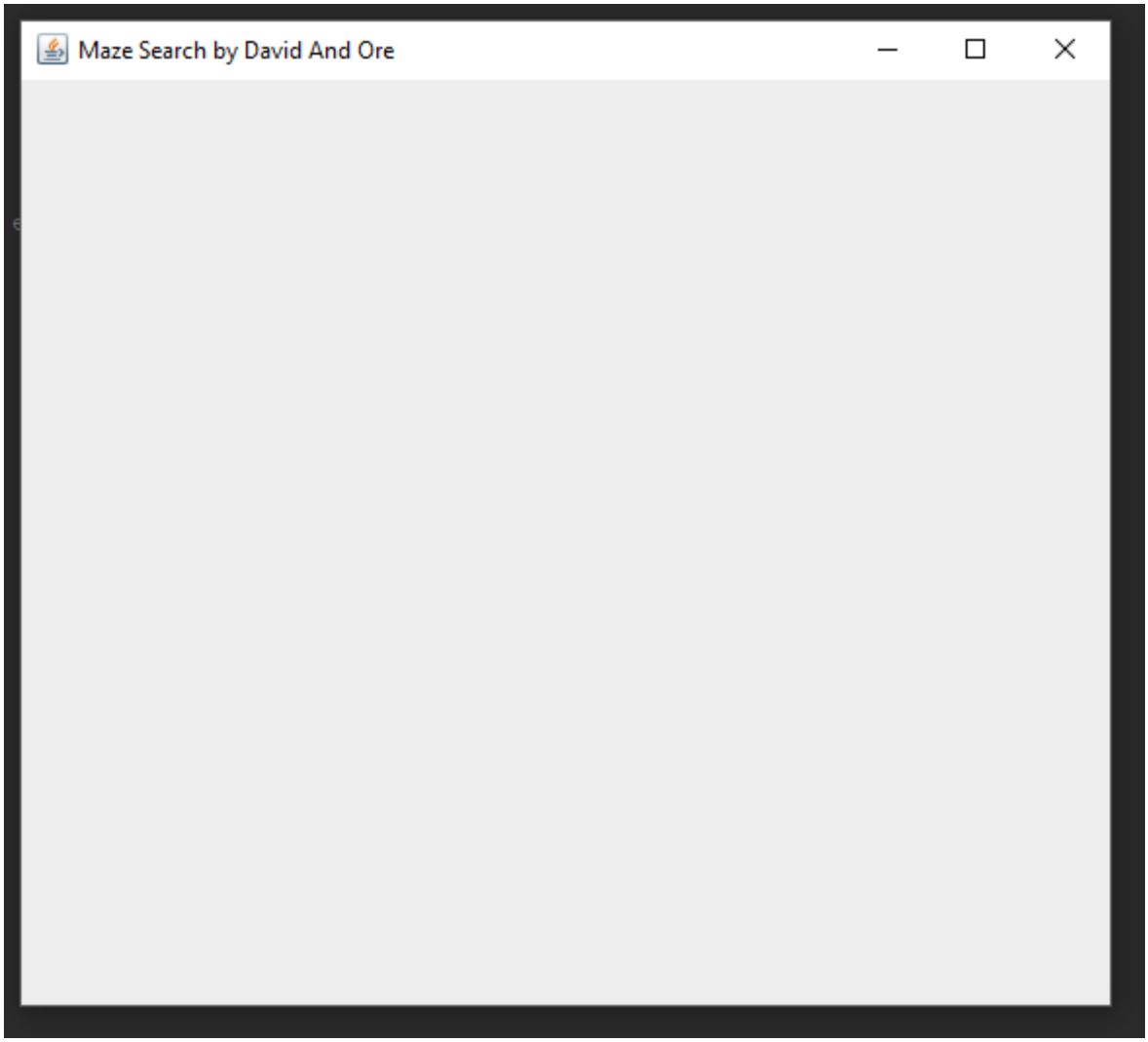
However, there were still further problems.
Even though it made it incredibly easy to design the UI as it auto generated code for us.

It made it difficult to learn how to implement beyond that as tutorials online would only detail how to build the UI manually.

So we followed this
series: https://youtu.be/jUEOWVjnIR8,https://youtu.be/svM0SBFqp4s
And this site for aid: https://www.javatpoint.com/java-swing

Below is the start of our UI implementation. We started by created a simple 500x500 window.

```java
public static void main(String[] args) {

    mazeFrame = new JFrame("Maze Search by David And Ore");
    mazeFrame.setSize(500, 500);
    mazeFrame.pack();
    //it loads on the centre of the screen
    mazeFrame.setLocationRelativeTo(null);
    mazeFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mazeFrame.setVisible(true);

} // end main()
```



Maze creation Problem:

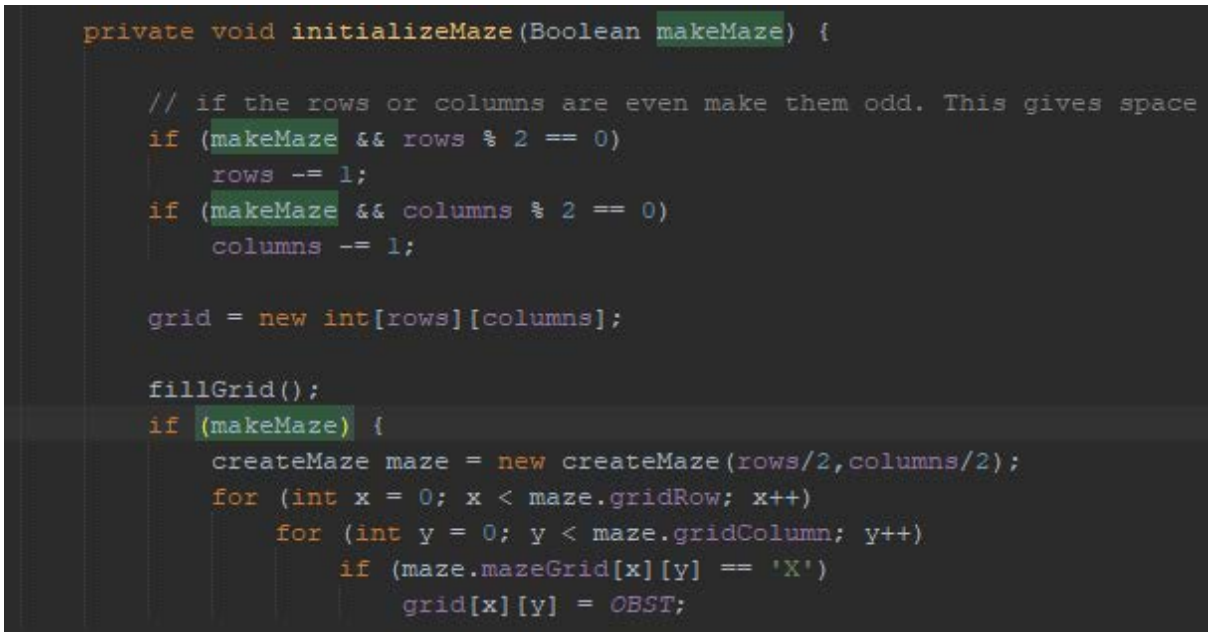
The major issue here was we simply didn't know how to create the exact maze we required.

Fortunately, we found exactly what we needed from a stackoverflow thread.
we created a "createMaze" class and all code in this class has been credited to this link:

https://stackoverflow.com/questions/18396364/maze-generation-arrayindexoutofboundsexception

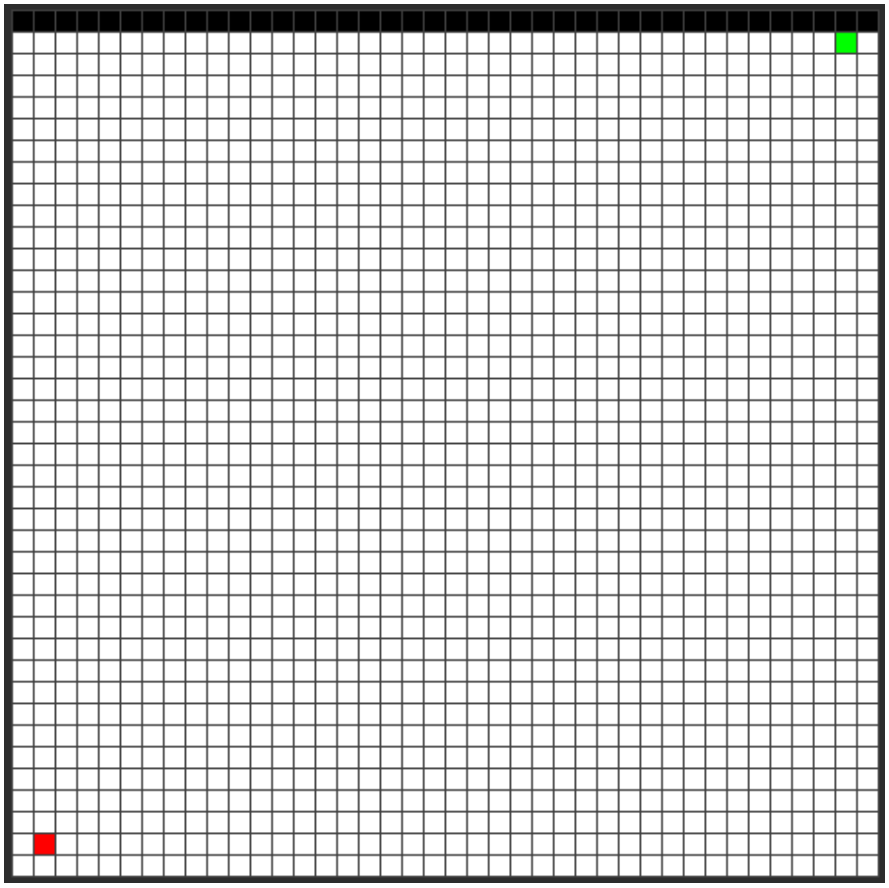
The code in that link managed to generate a maze that protects against an ArrayIndexOutOfBoundsException which is exactly what we needed.

This allowed us to couple that code with an InitiliazeMaze class that allowed us to have it fit the maze into our specified grid:

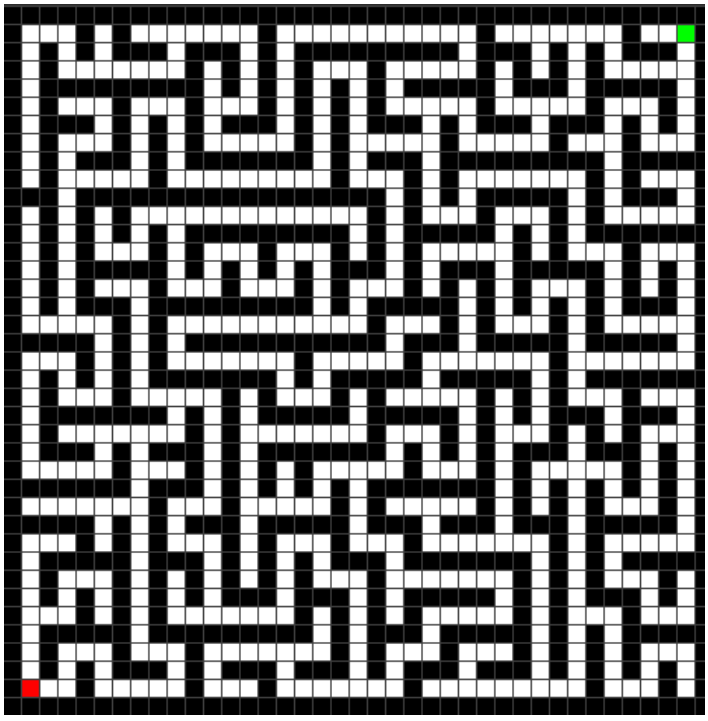


```
private void initializeMaze(Boolean makeMaze) {

    // if the rows or columns are even make them odd. This gives space
    if (makeMaze && rows % 2 == 0)
        rows -= 1;
    if (makeMaze && columns % 2 == 0)
        columns -= 1;

    grid = new int[rows][columns];

    fillGrid();
    if (makeMaze) {
        createMaze maze = new createMaze(rows/2,columns/2);
        for (int x = 0; x < maze.gridRow; x++)
            for (int y = 0; y < maze.gridColumn; y++)
                if (maze.mazeGrid[x][y] == 'X')
                    grid[x][y] = OBST;
```

We also took notice that the rows and columns had to be odd when generated in order for the maze to fit. Below is the problem not doing that created:



When you force odd number of columns and rows to fix the problem:



Time Problem:

Detailed in our blog post here:
https://davidproject.tumblr.com/post/171475390078/whats-possible

The major issue we had was time and feature implementation. Our proposal included features that would require more time (due to exams covering all of January and a few weeks leading up to exams for study).

Our solution was to eliminate a few features that didn't affect the overall purpose of the program.

5. Installation Guide

List of requirements:

- 32/64-bit Windows 7, 8, or 10 with Intel Core Intel Core™ i3, i5, or i7
- 4GB of RAM or more
- 1920x1080 display
- Java™ Runtime Environment 8+

Installation Steps:

1. Download the file maze_70x70.jar or maze_70x70_clean.jar from our gitlab repo.
2. Double click on the file when installed.