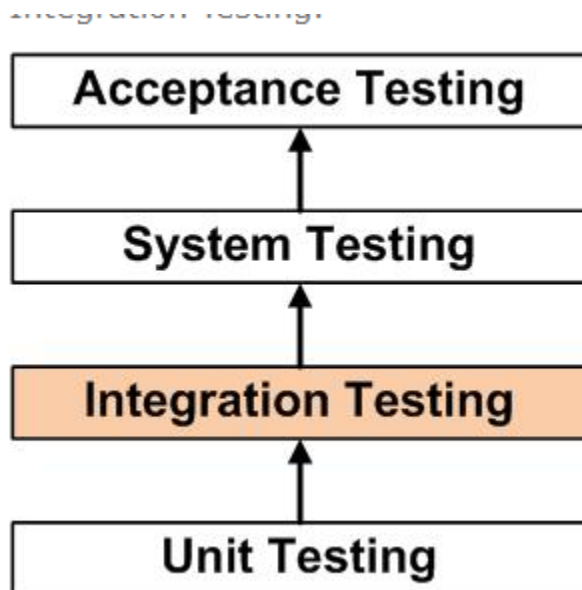


Introduction

The testing for our project involves a full testing regiment defined by ISTQB:



JUNIT was used to carry out Unit testing throughout the development of program.

Part of our testing strategy was to not over test or test inappropriately. Below, highlighted in bold, are methods and classes we've tested that we believe to be the most important in regards to unit testing.

Testing distance between Nodes:

After writing the method we found ourselves perplexed as to why the method wouldn't produce the distance we were looking for. That is the Manhattan distance of two Nodes. (a vertex of 2 dimensions).

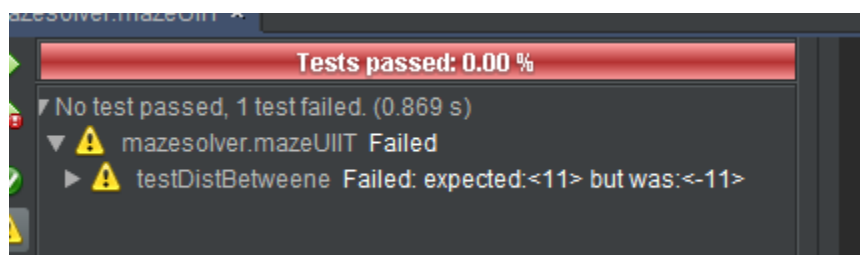
We built a test method to try and figure out why the result was incorrect.

```

/**
 * Test of distBetween method, of class mazeUI.
 * It should produce the manhattan distance between two nodes
 */
@Test
public void testDistBetween() {
    mazeUI test = new mazeUI(0, 0);
    Node node1 = new Node(0, 0);
    Node node2 = new Node(4, 7);

    int result = (int) test.distBetween(node1, node2);
    assertEquals(11, result);
}

```



As you can see it failed, saying that the expected answer was 11 but we were generating -11.

We then realized it requires the absolute addition of the distance in order to achieve it.

Original method:

```

/** Returns the distance between two Nodes ...3 lines */
double distBetween(Node u, Node v) {
    double dist;
    int dx = u.col-v.col;
    int dy = u.row-v.row;

    // calculate the Manhattan distance
    dist = dx+dy;

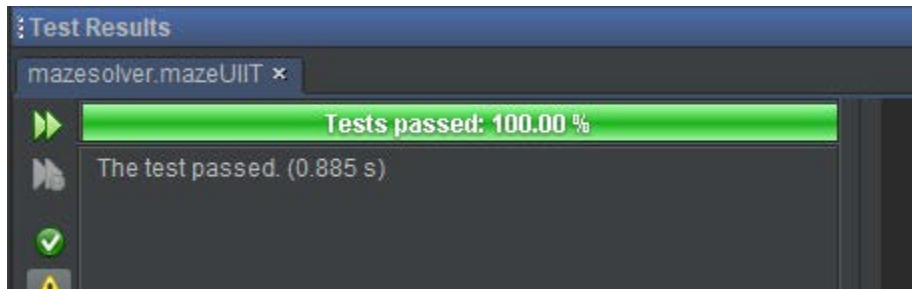
    return dist;
} // end distBetween()

```

Solution:

```
/** Returns the distance between two Nodes ...3 lines */  
public double distBetween(Node u, Node v){  
    double dist;  
    int dx = u.col-v.col;  
    int dy = u.row-v.row;  
  
    // calculate the Manhattan distance  
    dist = Math.abs(dx)+Math.abs(dy);  
  
    return dist;  
} // end distBetween()
```

Now we run the test and it turns out successful:

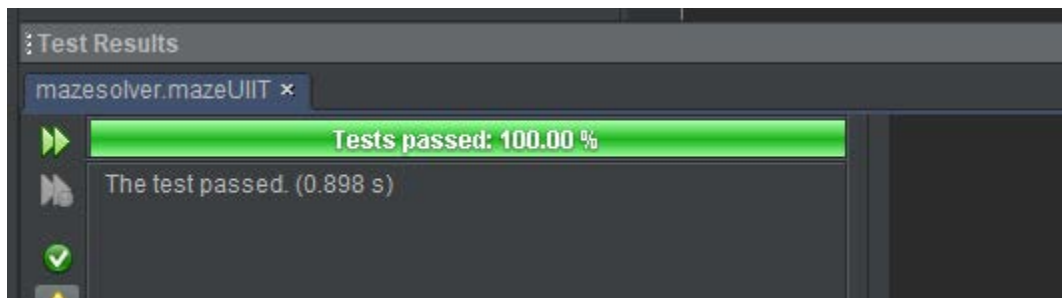


To further validate this, we created more nodes and asserted the results:

```
/**
 * Test of distBetween method, of class mazeUI.
 * It should produce the manhattan distance between two nodes
 */
@Test
public void testDistBetween() {
    mazeUI test = new mazeUI(0, 0);
    Node node1 = new Node(0, 0);
    Node node2 = new Node(4, 7);

    Node node3 = new Node(-1, 4);
    Node node4 = new Node(10, 90);

    int result1 = (int) test.distBetween(node1, node2);
    int result2 = (int) test.distBetween(node3, node4);
    assertEquals(11, result1);
    assertEquals(97, result2);
}
```

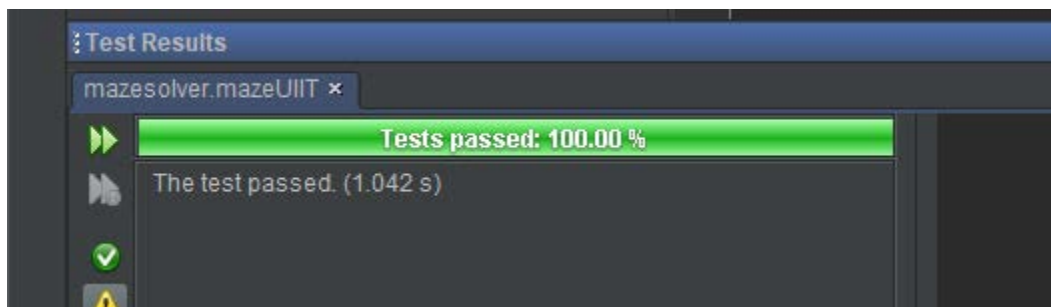


We also tried for 0:

```
@Test
public void testDistBetween() {
    mazeUI test = new mazeUI(0, 0);
    Node node1 = new Node(0, 0);
    Node node2 = new Node(0, 0);

    Node node3 = new Node(-1, 4);
    Node node4 = new Node(10, 90);

    int result1 = (int) test.distBetween(node1, node2);
    int result2 = (int) test.distBetween(node3, node4);
    assertEquals(0, result1);
    assertEquals(97, result2);
}
```



We only had to carry out these 3-unit tests on the method as it covered the entire test search space.

Testing Node Indexes

We also ran a test to validate if `getNodeIndex()` would work.

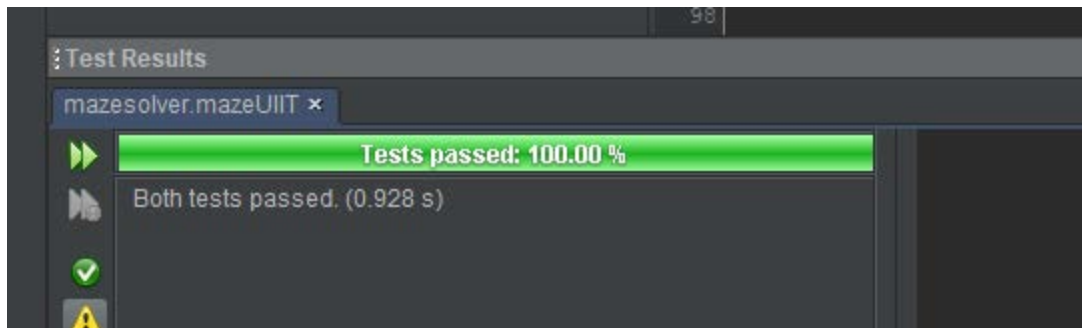
Its job is to return a Node's Index in a list of Nodes. Here we created 10 nodes and added them to a list.

```
/**
 * Test of getNodeIndex method, of class mazeUI.
 * It should produce the index of a Node in a list of Nodes
 */
@Test
public void getNodeIndex(){
    mazeUI test = new mazeUI(0, 0);
    Node node1 = new Node(5, 5);
    Node node2 = new Node(9, 3);
    Node node3 = new Node(3, 2);
    Node node4 = new Node(1, 9);
    Node node5 = new Node(9, 3);
    Node node6 = new Node(3, 5);
    Node node7 = new Node(3, 8);
    Node node8 = new Node(1, 8);
    Node node9 = new Node(3, 7);
    Node node10 = new Node(3, 1);

    ArrayList<Node> list = new ArrayList();
    list.add(node1);
    list.add(node2);
    list.add(node3);
    list.add(node4);
    list.add(node5);
    list.add(node6);
    list.add(node7);
    list.add(node8);
    list.add(node9);
    list.add(node10);

    int result1 = (int) test.getNodeIndex(list, node7);
    assertEquals(6, result1);
}
```

Upon creation of the original method we managed to achieve it working without usage of the test method. However, to validate it we ran the above test regardless. We asked for node7 at index 6 in the above test.



The tests passed.

The open set:

Regarding Unit test, the final formal testing we carried out involved our longest function.

The createSuccessors() method. This involves return the list of nodes in the open set.

Like with most testing this involved adding print statements into the Node class and createsuceessors method.

```
2 package mazesolver;
3
4 public class Node {
5     int row;    // the row number of the grid(row 0 is the top)
6     int col;    // the column number of the grid (Column 0 is the left)
7
8     // f(n) = g(n)+h(n). Used for the A* and greedy algorithms.
9     double g;
10    double h;
11    double f;
12    // f(n) = g(n)+h(n). Used for the A* and greedy algorithms.
13
14    double dist; // the distance of the node from the initial position of the robot
15
16    Node prev;   //variable storing the previous Node
17    public Node(int row, int col){
18        this.row = row;
19        this.col = col;
20        System.out.println("row: " + row);
21        System.out.println("col: " + col);
22    }
23
24 }
```

The added print statements above would allow us to locate the decisions made by the current Node in terms of what the successors are and shows up clearly in the console for us. We then add a test method in our separate java file to test just a specific Node decision rather than every single one that was made:

```

5
6
7
8
9
0
1
2
3
4
5
6
7
8
9
0
1
2
3
4
5
6
7
8
9
0
1

```

```

/**
 * Test of createSuccessors method, of class mazeUI.
 * It should produce the index of a Node in a list of Nodes
 */
@Test
public void testCreateSuccessor(){
    mazeUI test = new mazeUI(0, 0);
    Node node = new Node(0, 0);
    Node testnode1 = new Node(0,1);
    Node testnode2 = new Node(1,0);

    ArrayList<Node> list = new ArrayList<>();
    list.add(testnode1);
    list.add(testnode2);

    ArrayList<Node> result = test.createSuccessors(node, true);

    assertEquals(list, result);
}

```

```

}

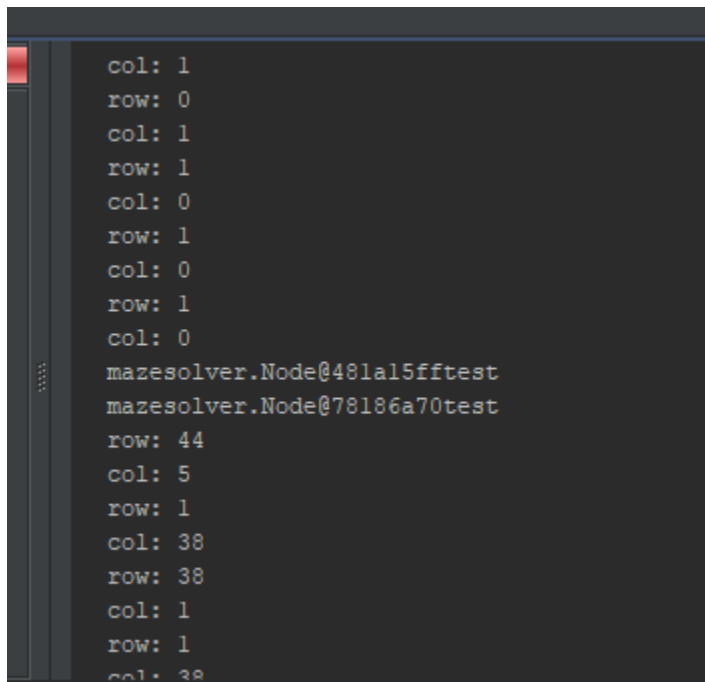
// When DFS algorithm is in use, Nodes are added one by one at the beginning
// OPEN SET list. Because of this, we must reverse the order of successors fo
// so the successor corresponding to the highest priority, to be placed
// the first in the list.
// For the Greedy, A* and Dijkstra's no issue, because the list is sorted
// according to 'f' or 'dist'.
if (dfs.isSelected())
    Collections.reverse(temp);

for (int i = 0; i < temp.size(); i++){
    System.out.println(temp.get(i).toString() + "test");
}

return temp;

```


The console:



```
col: 1
row: 0
col: 1
row: 1
col: 0
row: 1
col: 0
row: 1
col: 0
mazesolver.Node@481a15fftest
mazesolver.Node@78186a70test
row: 44
col: 5
row: 1
col: 38
row: 38
col: 1
row: 1
col: 38
```

The [mazesolver.Node@....test](#) lines indicate the result we're asserting in our test.

However, this is unreadable as it's showing the memory address. So we built a simple toString Method to remedy this:

```
2 package mazesolver;
3
4
5 public class Node {
6     int row; // the row number of the grid(row 0 is the top)
7     int col; // the column number of the grid (Column 0 is the left)
8
9     // f(n) = g(n)+h(n). Used for the A* and greedy algorithms.
10    double g;
11    double h;
12    double f;
13    // f(n) = g(n)+h(n). Used for the A* and greedy algorithms.
14
15    double dist; // the distance of the node from the initial position of the robot
16
17    Node prev; //variable storing the previous Node
18    public Node(int row, int col){
19        this.row = row;
20        this.col = col;
21        System.out.println("row: " + row);
22        System.out.println("col: " + col);
23    }
24
25
26
27    @Override
28    public String toString() {
29        return "successor row: " + row + " " + "successor col: " + col;
30    }
31
32
33
34 }
```

Now the console shows:

```
col: 0
row: 1
col: 0
row: 1
col: 0
successor row: 1 successor col: 0
successor row: 0 successor col: 1
row: 44
col: 5
row: 1
col: 0
```

This test didn't need to be asserted to anything as all that is required to validate its operation is to check if the rows and columns generated never create an out of bounds exception.

Full output:

row: 44

col: 5

row: 1

col: 38

row: 38

col: 1

row: 1

col: 38

row: 0

col: 0

row: 0

col: 0

row: -2

col: 4

row: 10

col: 90

row: 44

col: 5

row: 1

col: 38

row: 38

col: 1

row: 1

col: 38

row: 1

col: 1

row: 1

col: 0

row: 2

col: 1

row: 1

col: 2

row: 0

col: 1

row: 0

col: 1

row: 0

col: 1

row: 0

col: 1

row: 1

col: 2

row: 1

col: 2

row: 1

col: 2

row: 2

col: 1

row: 2

col: 1

row: 2

col: 1

row: 1

col: 0

row: 1

col: 0

row: 1

col: 0

successor row: 1 successor col: 0

successor row: 2 successor col: 1

successor row: 1 successor col: 2

successor row: 0 successor col: 1

row: 44

col: 5

row: 1

col: 38

row: 38

col: 1

row: 1

col: 38

row: 5

col: 5

row: 9

col: 3

row: 3

col: 2

row: 1

col: 9

row: 9

col: 3

row: 3

col: 5

row: 3

col: 8

row: 1

col: 8

row: 3

col: 7

row: 3

col: 1

Strangely, it did. A row with -2 was created which even for an obstacle shouldn't happen. This then brought us back to the createMaze file which was tested (and is not our own code but code we've acknowledged and credited in our documentation and source code).

This file handled that row with -2:

```
        updateGrid();
    }
    // used to get a Cell at x, y; returns null out of bounds
    public Node getNode(int x, int y) {
        try {
            return nodes[x][y];
        } catch (ArrayIndexOutOfBoundsException e) { // catch out of bounds
            return null;
        }
    }
}
```

With an arrayIndexOutOfBoundsException.

Fortunately, we didn't have to worry about testing the createSuccessor() method any further due to this.

Above were the most challenging and crucial methods to unit test as they required a return value. That was white box testing.

All other methods are validated by the above unit testing as if the void methods did not work as required neither would the methods that returned data. Further validation was derived from the subsequent testing methods we carried out.

Once our white box testing was completed we moved forward to our black box testing method. Integration testing, system and acceptance testing.

When it came to testing graphical elements, it was matter of running the main file making sure what is seen followed our requirements.

Initially we had issues with no UI elements appearing at all until we implemented the addition of the .add() method in order to display content:

```

/** super is calling the JPanel class.
 * Here we're adding all our JButtons and JLabels to be printed to screen.
 */
super.add(message);
super.add(mazeButton);
super.add(clearButton);
super.add(CPUTimeButton);
super.add(drawPathButton);
super.add(speedLabel);
super.add(speedController);
super.add(dfs);
super.add(bfs);
super.add(aStar);
super.add(greedy);
super.add(dijkstra);
super.add(algoPanel);

```

Our integration testing included trying multiple positions and sizes of components until they fit the display according to our usability testing results. Initially this was automated which we explained the reason for switching to a manual approach in our blog:

<https://davidproject.tumblr.com/post/171475287323/java-setup>

```

//(dimensionX in Pixels, dimensionY in Pixels, width of actual JLabel)
message.setBounds(58, 850, 400, 75);
dfsMessage.setBounds(40,450, 400, 500);
bfsMessage.setBounds(40,450, 400, 500);
astarMessage.setBounds(40,425, 400, 500);
greedyMessage.setBounds(40,450, 400, 500);
dijkstraMessage.setBounds(40,450, 400, 500);
mazeButton.setBounds(10, 520, 140, 50);
CPUTimeButton.setBounds(10,580, 140, 50);
drawPathButton.setBounds(10, 640, 140, 50);
speedController.setBounds(10, 730, 200, 40);
speedLabel.setBounds(15, 760, 190, 40);
clearButton.setBounds(10,820, 140, 50);
algoPanel.setLocation(250,520);
algoPanel.setSize(220, 150);
dfs.setBounds(255, 545, 50, 20);
bfs.setBounds(255, 575, 70, 25);
aStar.setBounds(255, 610, 70, 25);
greedy.setBounds(360, 545, 85, 25);
dijkstra.setBounds(360, 575, 85,25);
resetUI.setBounds(10,510, 25, 25);

```

Another crucial test was the constant review of the colourizing method paintComponent:

```

@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    // Fills the background color.
    g.setColor(Color.DARK_GRAY);

    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < columns; c++) {
            if (grid[r][c] == EMPTY) {
                g.setColor(Color.WHITE);
            } else if (grid[r][c] == ROBOT) {
                g.setColor(Color.RED);
            } else if (grid[r][c] == TARGET) {
                g.setColor(Color.GREEN);
            } else if (grid[r][c] == OBST) {
                g.setColor(Color.BLACK);
            } else if (grid[r][c] == OPEN) {
                g.setColor(Color.BLUE);
            } else if (grid[r][c] == CLOSED) {
                g.setColor(Color.MAGENTA);
            } else if (grid[r][c] == ROUTE) {
                g.setColor(Color.YELLOW);
            }
            // fillRect(position x, position y, size width, size height)
            // gridSize to eliminate lines. -1 is used in order to offset the color position.
            g.fillRect(11 + c*gridSize, 11 + r*gridSize, gridSize, gridSize);
        }
    }

    // end paintComponent()
}

// end MazuUI Panel

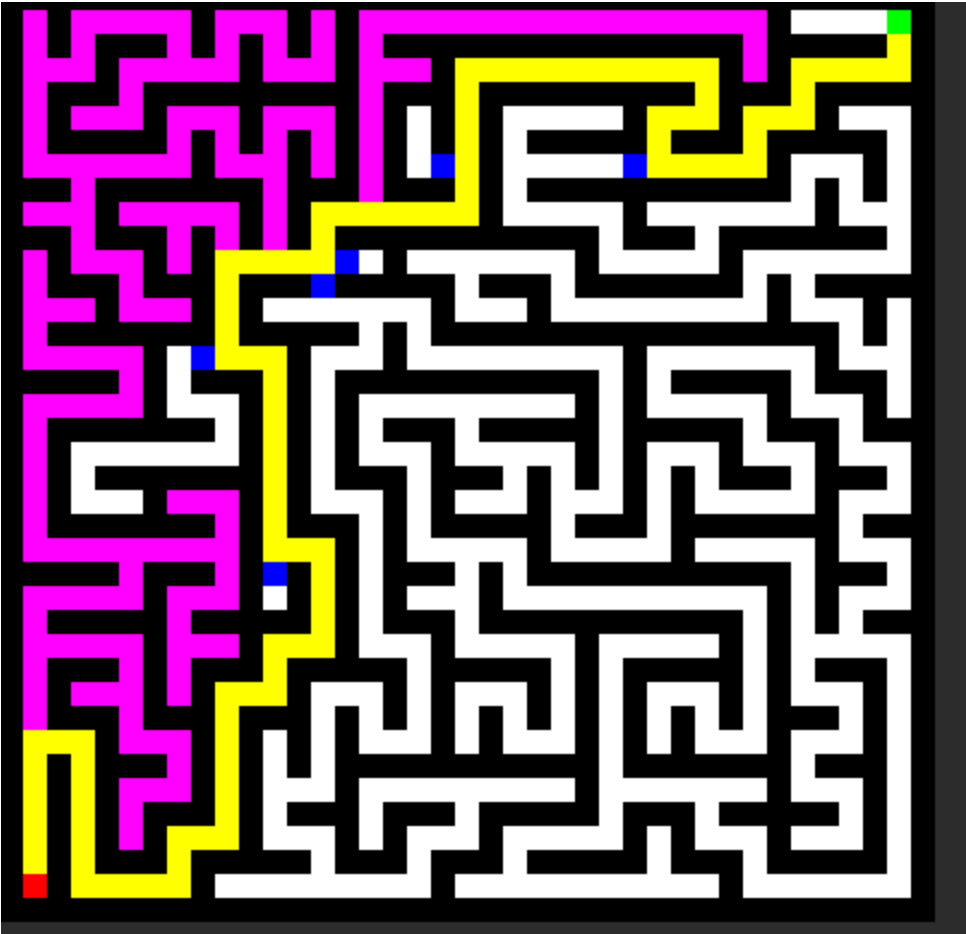
```

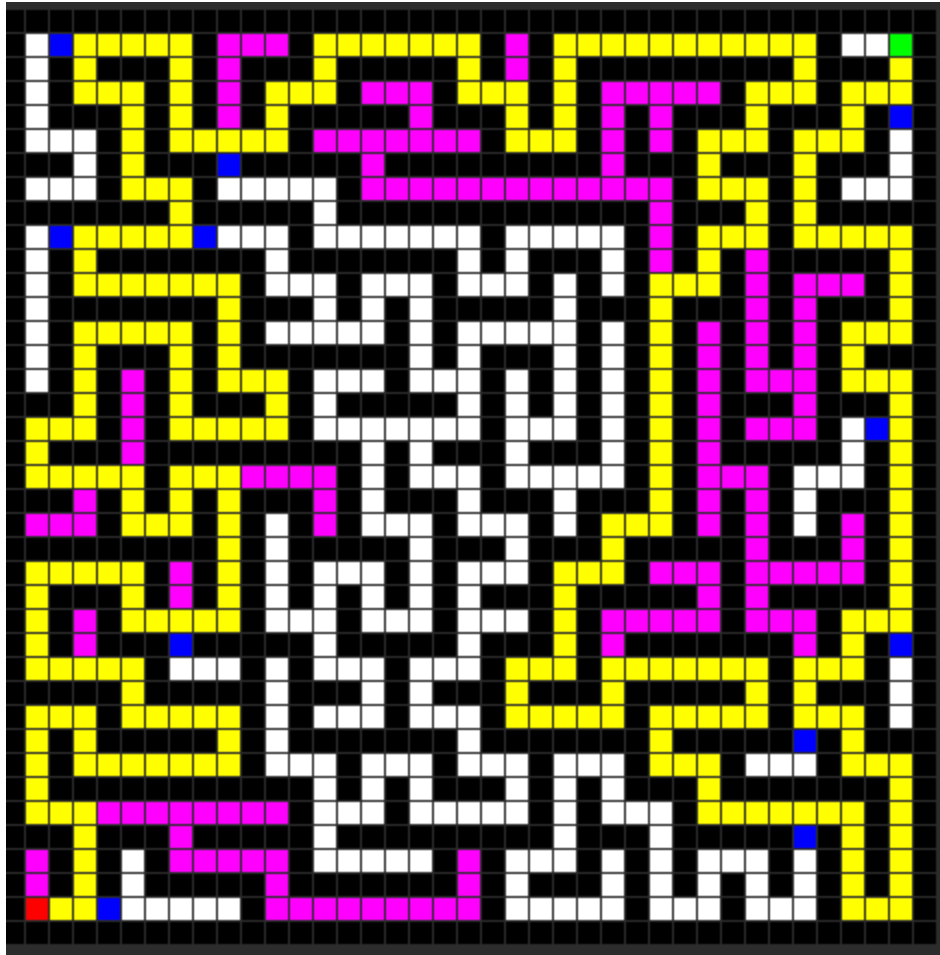
Like said before, this was a matter of running main.java over and over assuring the colours were the colours they should be.

We also struggled to decide if the grid should display the individual squares or not and how many rows and columns we should use. So, we compiled two versions. This was done in order to please results we obtained from our usability testing (mentioned later in this document):

`g.fillRect(11 + c*gridSize, 11 + r*gridSize, gridSize, gridSize);` generates without the grid.

`g.fillRect(11 + c*gridSize, 11 + r*gridSize, gridSize - 1, gridSize - 1);` generates with the grid by offsetting the black colour paint.

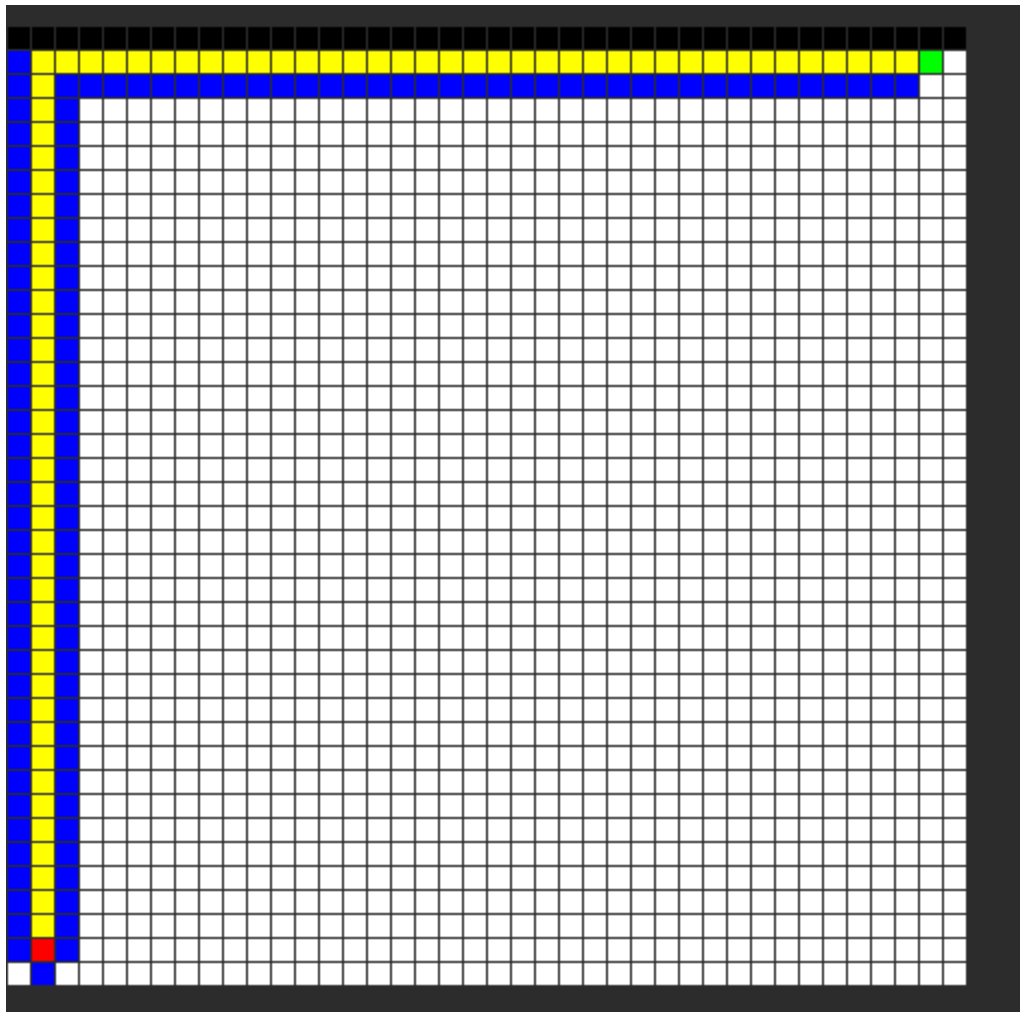




We then moved to our System testing which was running the program and testing each and every single component and how they interacted with each other.

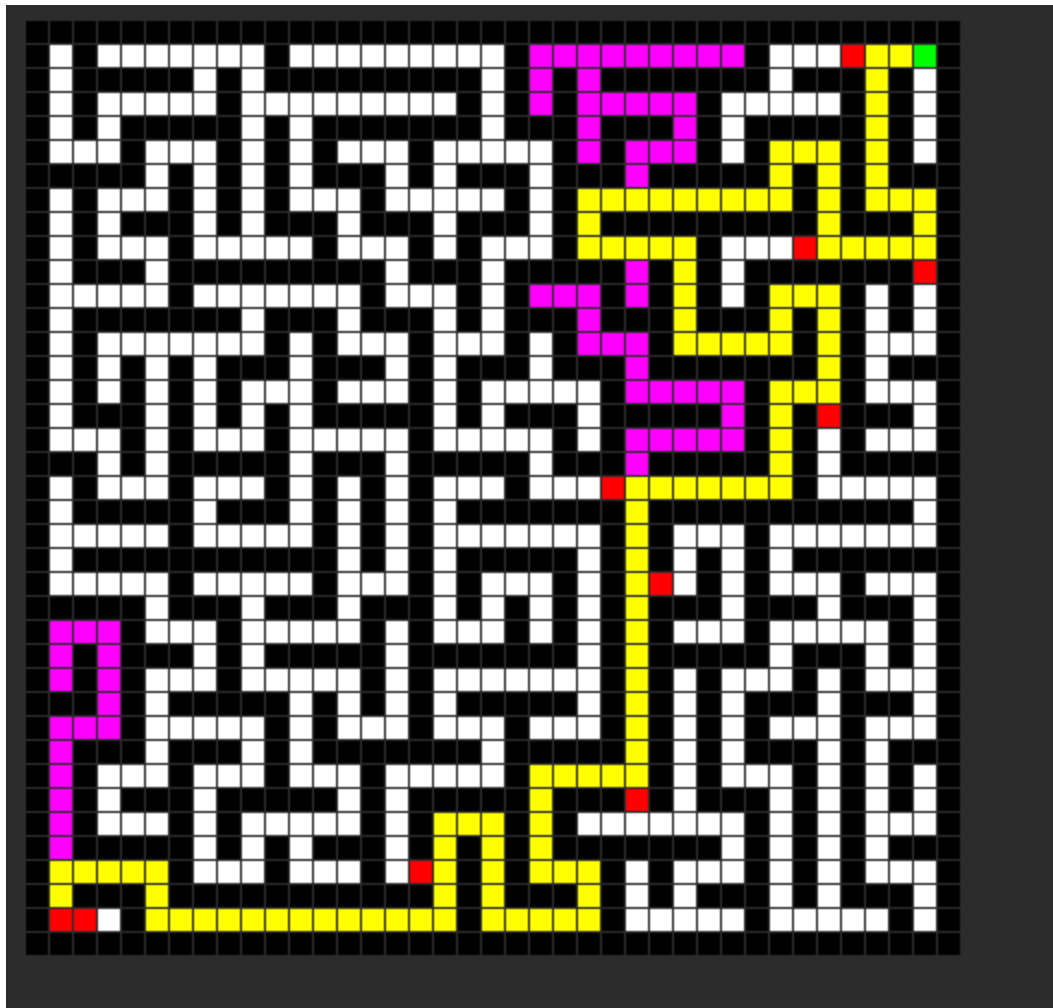
We found a handful off issues here.

To begin, the maze wasn't being created at one point. We came to the conclusion that the grid must have an odd amount of rows and columns to allow space for the black maze. This is the result of not doing that:



Another issue we uncovered from system testing was that the robot was replicating the action of the open set:

```
} // end constructor
private final static int
    INFINITY = Integer.MAX_VALUE, // Infinity
    EMPTY    = 0, // empty cell
    OBST      = 1, // cell with obstacle
    ROBOT     = 7, // the position of the robot
    TARGET    = 3, // the position of the target
    OPEN      = 7, // open set
    CLOSED    = 5, // closed set
    ROUTE     = 6; // cells that form the robot-to-target path
```



The red ends that extend from the yellow path should be blue. We found out that the robot was using the same final int value as the Open set. This they were constantly overriding each other.

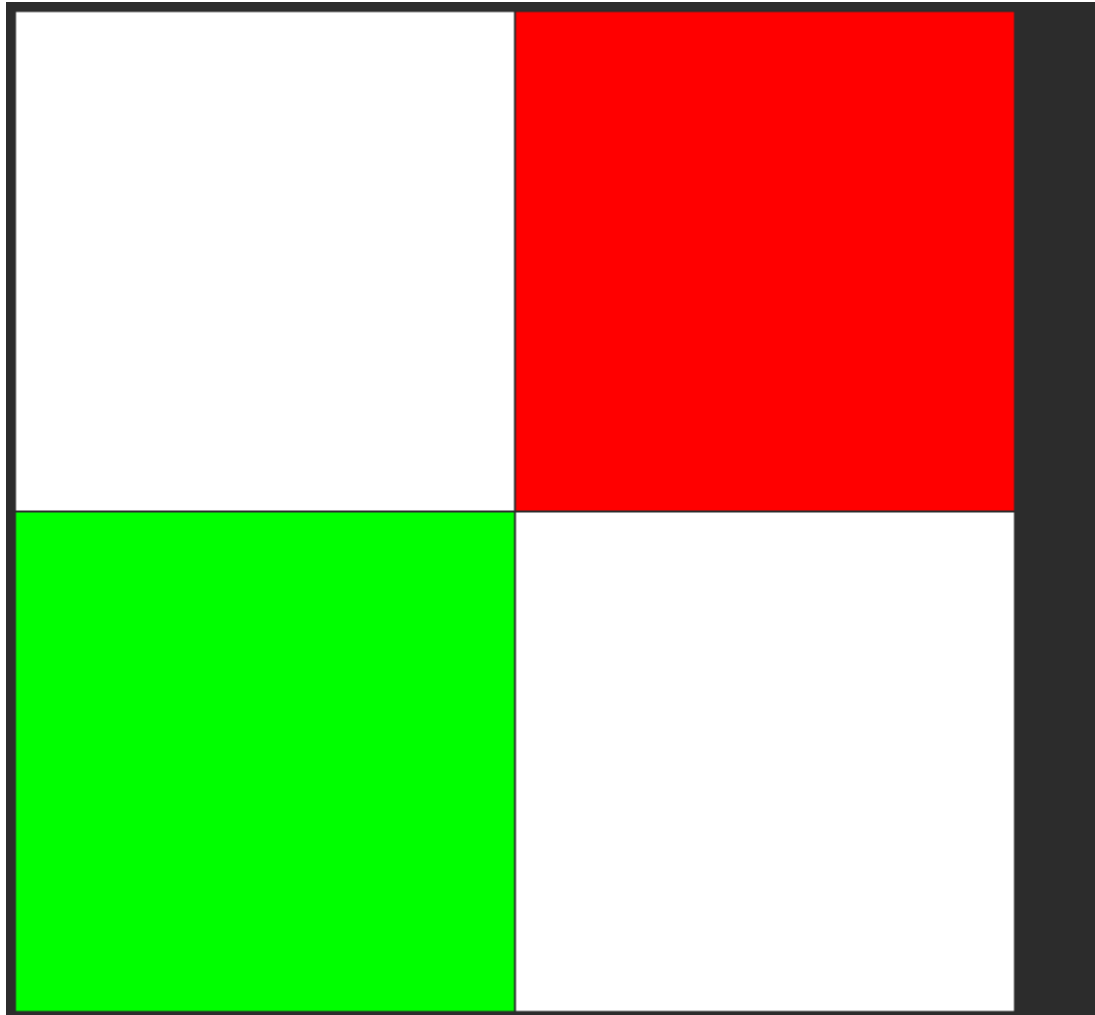
The solution was to change the robot to a value of 4 (this value doesn't matter, it just has to be different from the other final ints).

We also pushed the size of the grid to its lower and upper limits.

We start with a 1x1, 1x2 and 2x1 maze: Which failed (as expected).

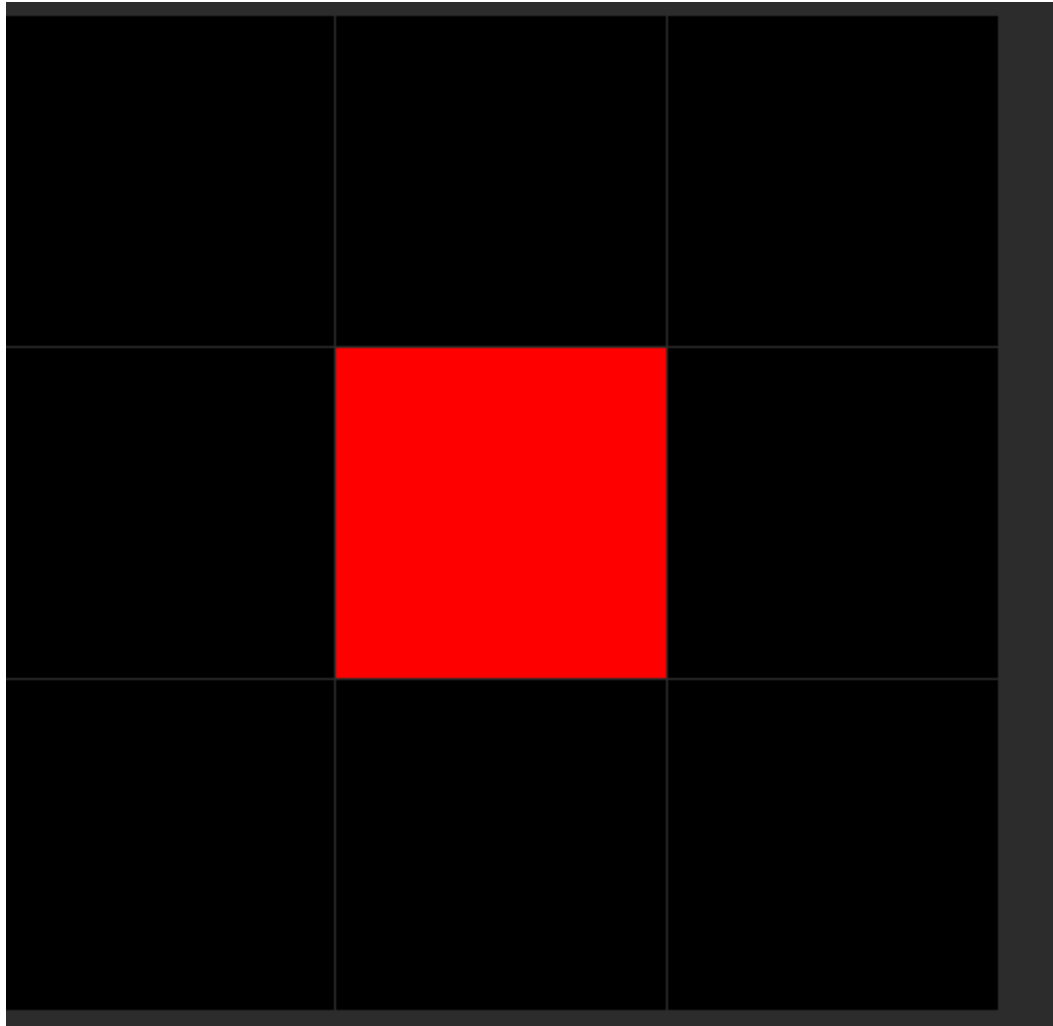
```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
    at mazesolver.mazeUI.initializeGrid(mazeUI.java:421)
    at mazesolver.mazeUI.initializeMaze(mazeUI.java:367)
    at mazesolver.mazeUI.<init>(mazeUI.java:327)
    at mazesolver.Main.main(Main.java:28)
```

Since there has to be at least one square available for the generation of a maze that sets its lower limits to 2x2:



However, maze creation is still impossible here. This is the lower limit for grid creation.

We now try 3x3:



Both grid and maze creation are viable however. Path calculation is now not possible. The red square appears to be pushed to the center but is actually on the bottom left in terms of free grid space. (Remember, one square is required for maze generation).

We then try 4x4:



Strangely it converts to the 3x3 grid when we attempt to create a maze. This would be concerning if our goal wasn't to show robot motion planning. As the effects are only visible when the rows and columns are of a larger number. So, we acknowledge this bug and ignore it.

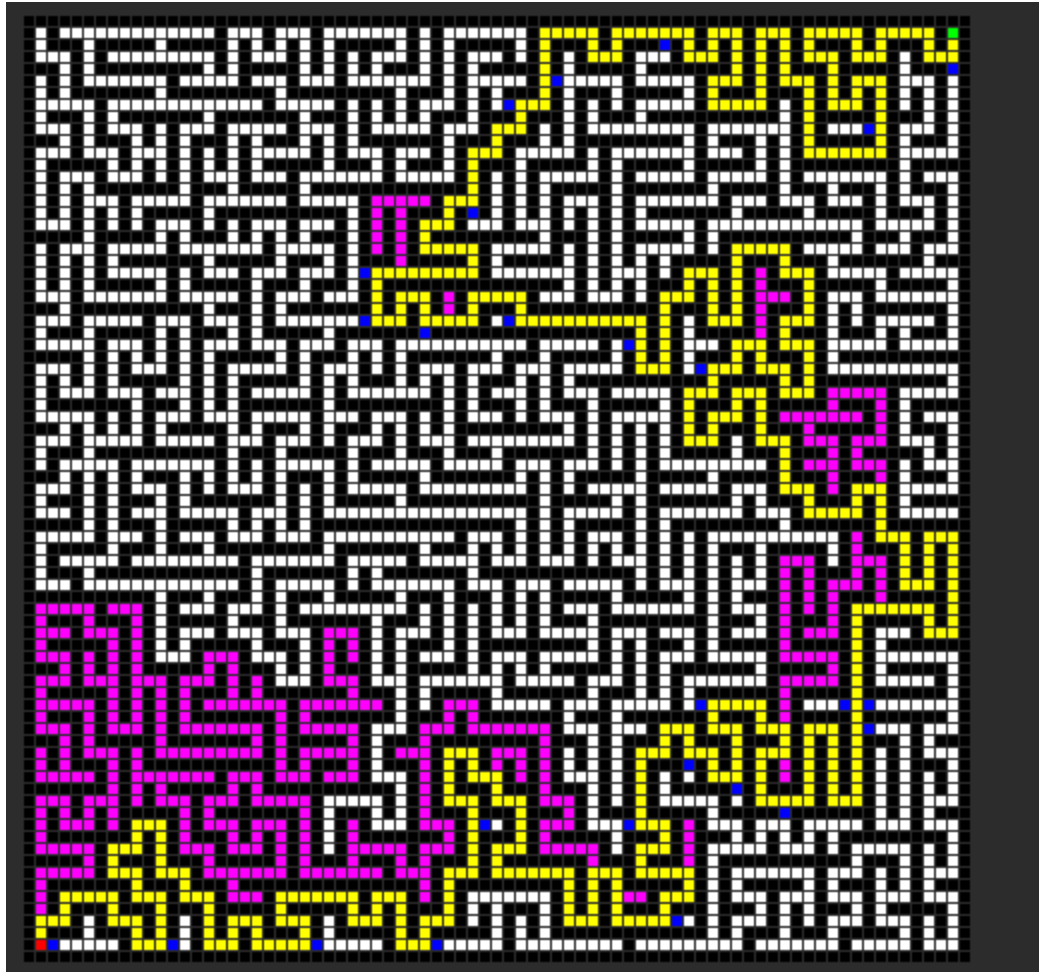
And now we try 5x5:



This is the lowest possible grid size that can be used to allow the program to run all functions.

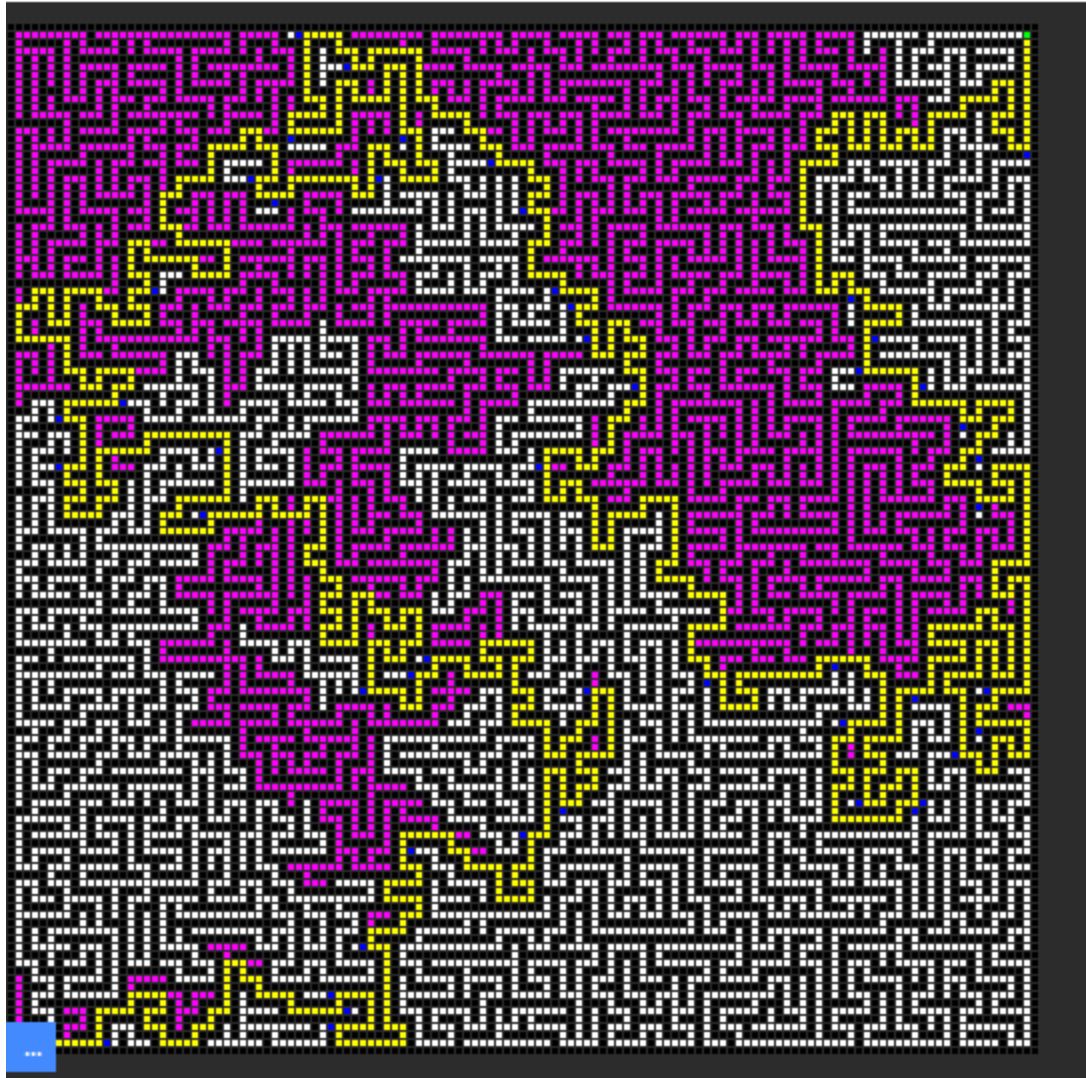
We know move forward and attempt to discover the upper limits.

We start with 80x80:



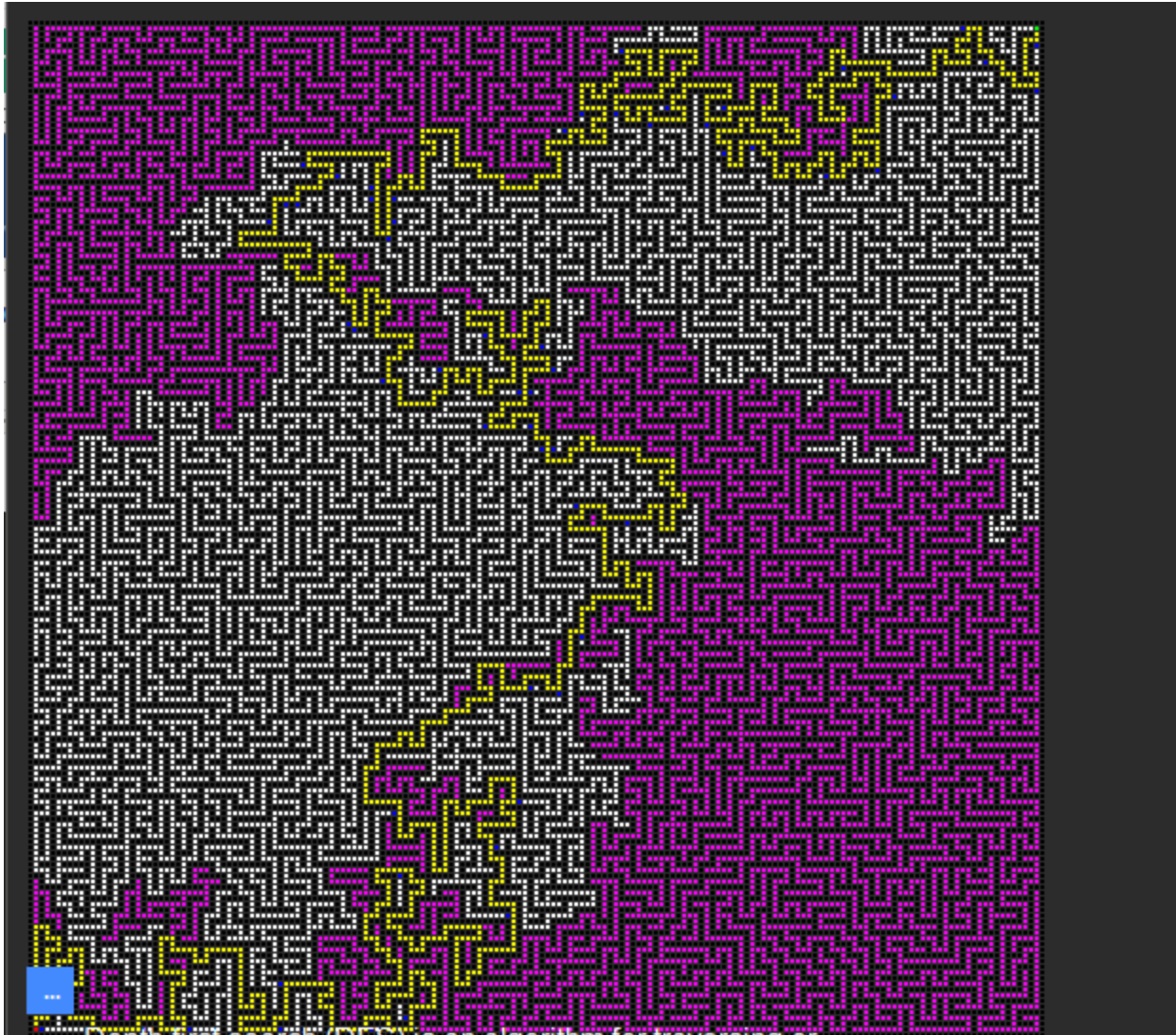
Still working.

130x130:



Still working.

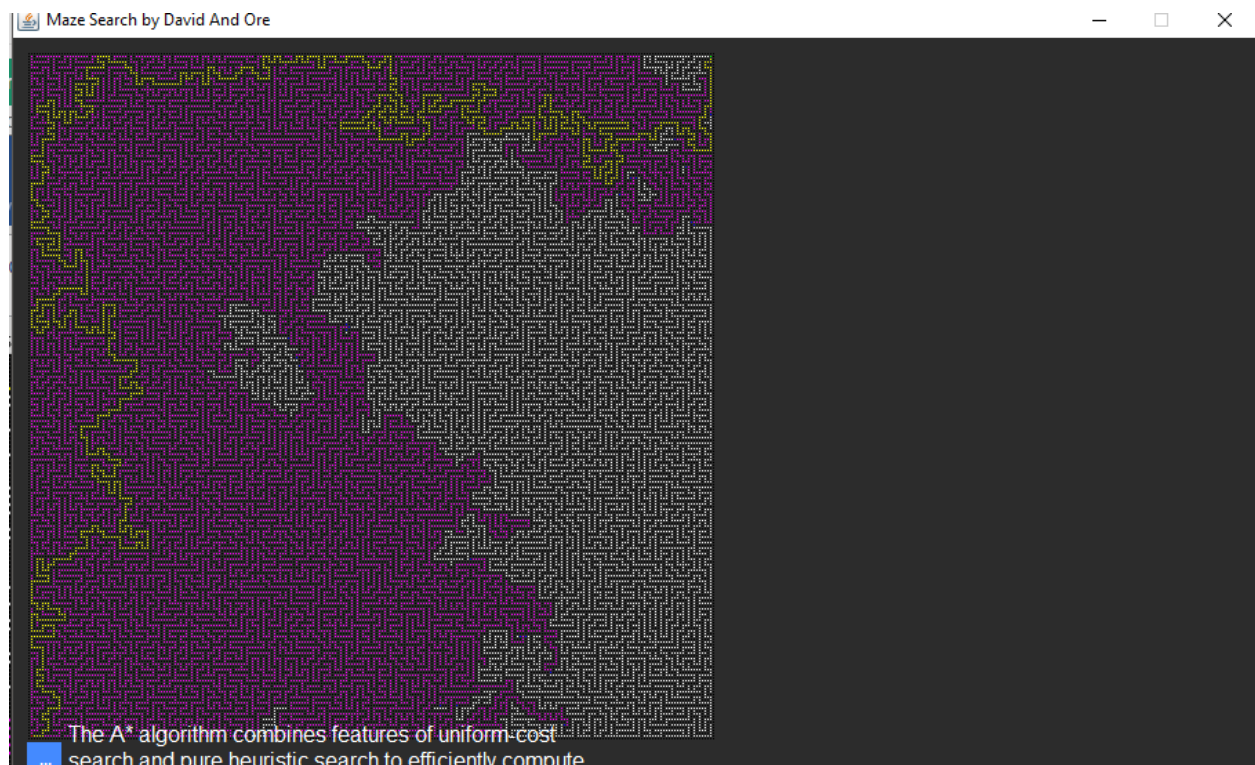
180x180:



Overlapping of UI occurs but we know that's only due to us accommodating a 1920x1080 display.

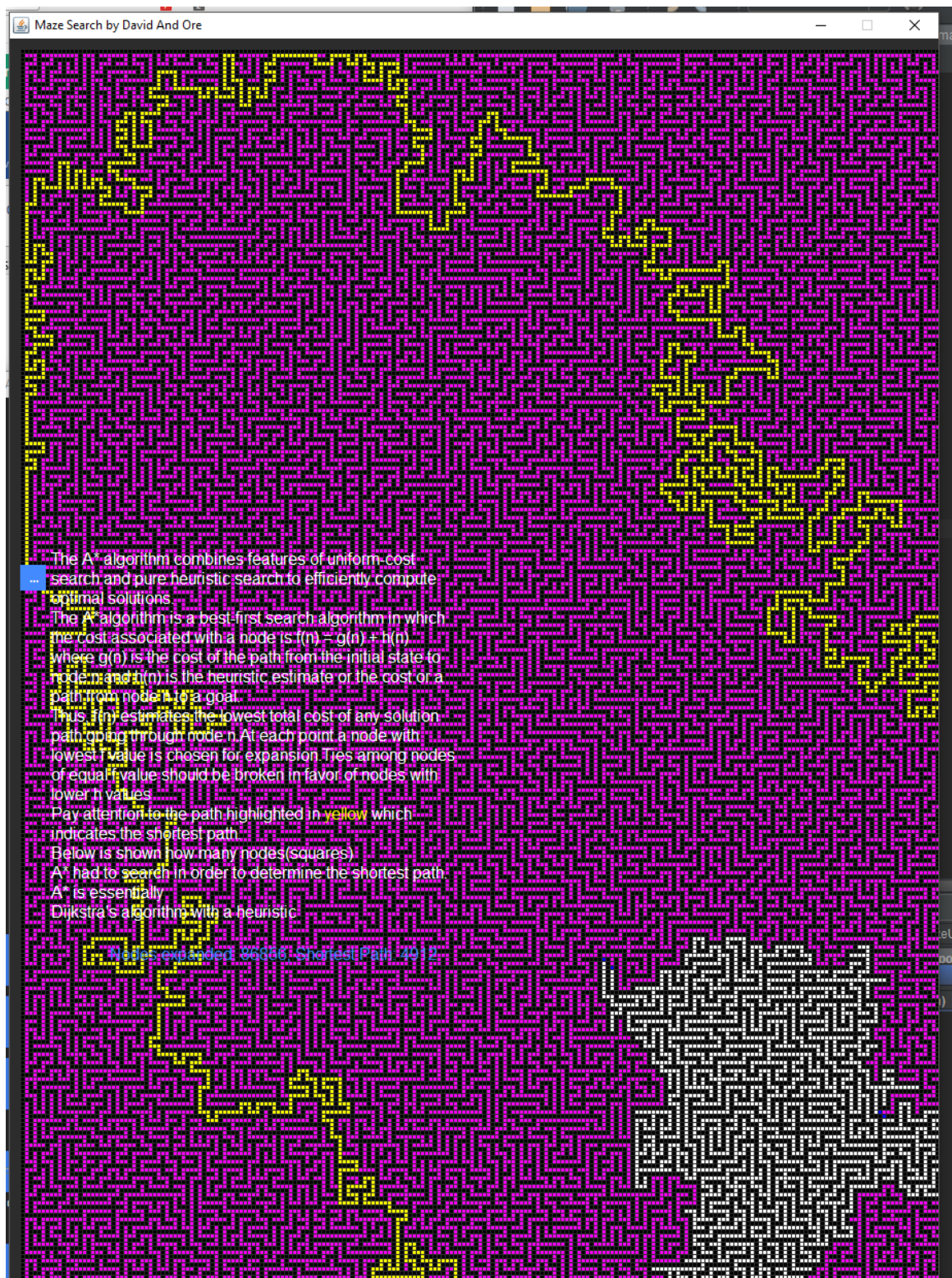
Still working.

250x250:



Still working.

We know really push it with 500x500:



As you can see we had to increase the size of the grid to 2000 pixels. We also found even with such a large maze it solved it in only a few seconds.

Below is the hardest we pushed it. Pay close attention to the CPU and RAM usage. At even such a high resolution the program remains using very little computational power (however the pc we are using is very powerful).

https://drive.google.com/open?id=1FzqaQI2OzLNGP_Umh2NA0xwhoYsRuG6K

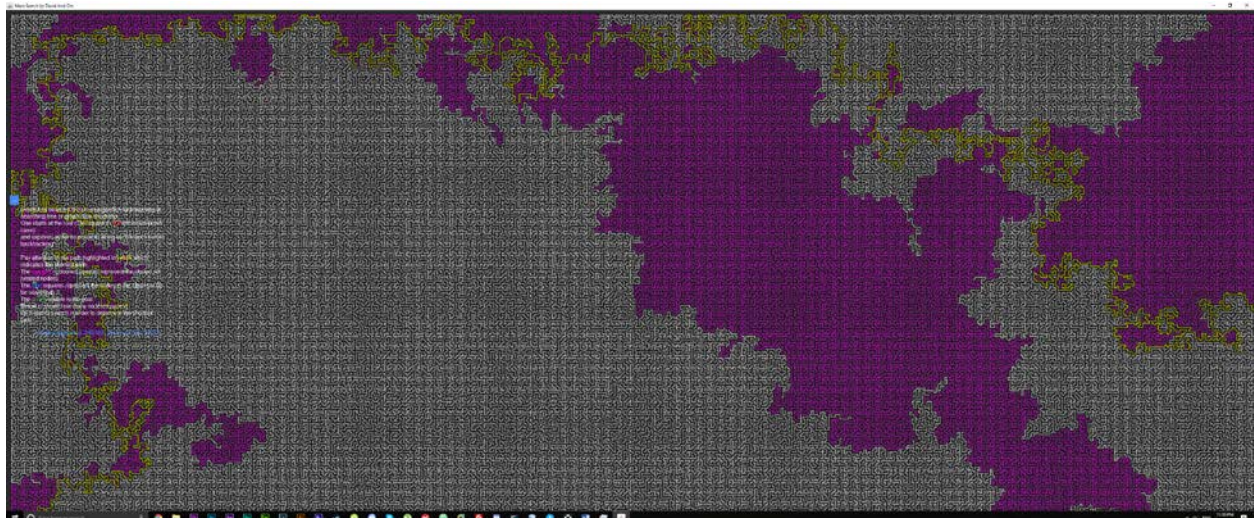
We generate a 3440x1440 maze. As you can see it only uses less than 20% of our CPU power and very little RAM. It can generate a maze but takes too long to calculate the path.

So, we end the test early. As you can see from the console the program does not halt and actually calculates the nodes as required. Below is a video showing that node calculation takes place even with a large maze.

<https://drive.google.com/open?id=1drsSHvFjxS4rw0EZfUNf-CgYetvWSuwa>

However, we acknowledge that very little people run a 3440x1440 display but it was a way to validate the upper limits.

We now try to replicate the labs monitor with a 1920x1080 grid (upscaled to our 3440x1440):



This approximately took us 3 minutes to achieve the result.

So, even though we have no definite upper limit we can safely assume it's far beyond what any normal use case would require (ie lab usage).

Finally, we approach acceptance testing. We do this in the form of usability testing.

Usability Testing

Product to test:

- To carry out usability testing we used what we believed would be the final version of our program. We planned to select a certain number of people to use the program.

Target users:

- Our target users were fellow CASE class mates and students from other disciplines who have some understanding of search algorithms.

Test plan:

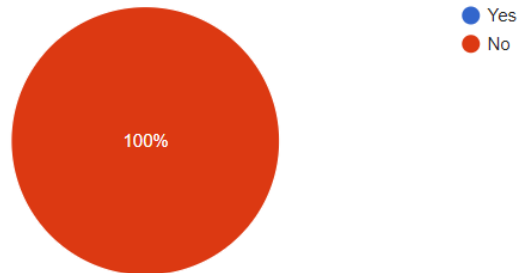
- Our test plan was to get the users to play around with the UI a bit and ask them to fill out a questionnaire. These questions included:
 - If each user understood the concepts of each algorithm.
 - Did they have trouble installing the program?
 - Did they find the interface difficult to use?
 - Was the text readable?
 - Did they like the colours used?
 - Did the frame of the program load correctly?
- After getting feedback from the users we would then look at what steps were required to make improvements to the program; if there were any improvements to be made.

Findings:

- When we obtained the results from the questionnaire we found out that nobody had trouble installing the program.

Did you have any problems installing the program?

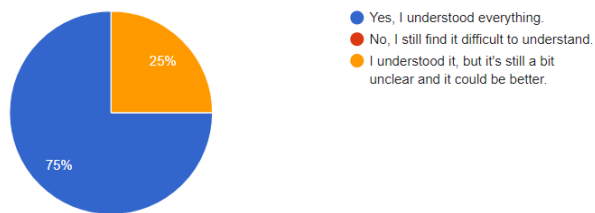
60 responses



- We found out that people could understand the concepts, but some thought it was unclear and that it could be improved.

Did you understand the concept of each algorithm when used?

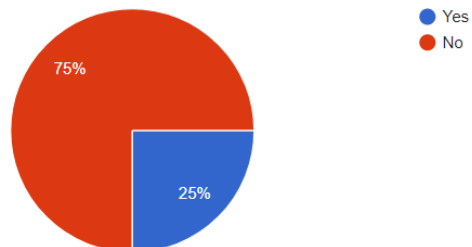
60 responses



- The screen size was correct for some users but not all. So, we made some changes to the UI.

Was the screen size correct?

60 responses

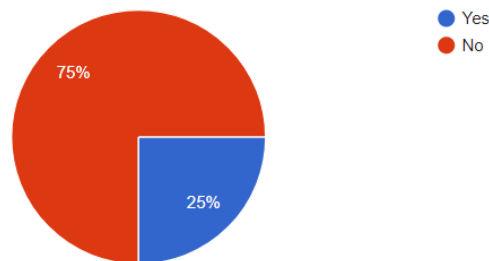


- Very few people found using the interface difficult. Which was good news, so we left that the same.

However comments regarding this question included repositioning the buttons to be closer together.

Did you find using the interface difficult?

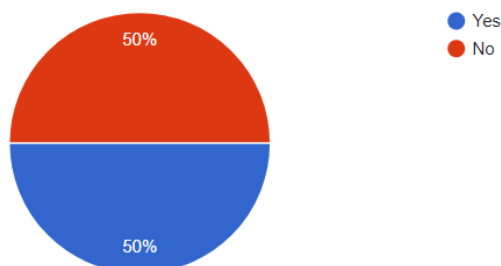
60 responses



- Quite a number of people felt that the size of font used was too small, so we made it larger.

Was the text readable?

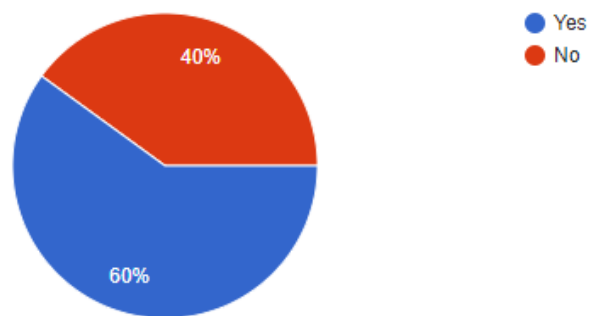
60 responses



- Less people liked the colour than anticipated. The comments mentioned how it looked like an old style of design but they were in favor of the colorus used when the robot traverses through the maze. (We presented a light and dark mode and comments on this question skewed to people preferring the dark mode).

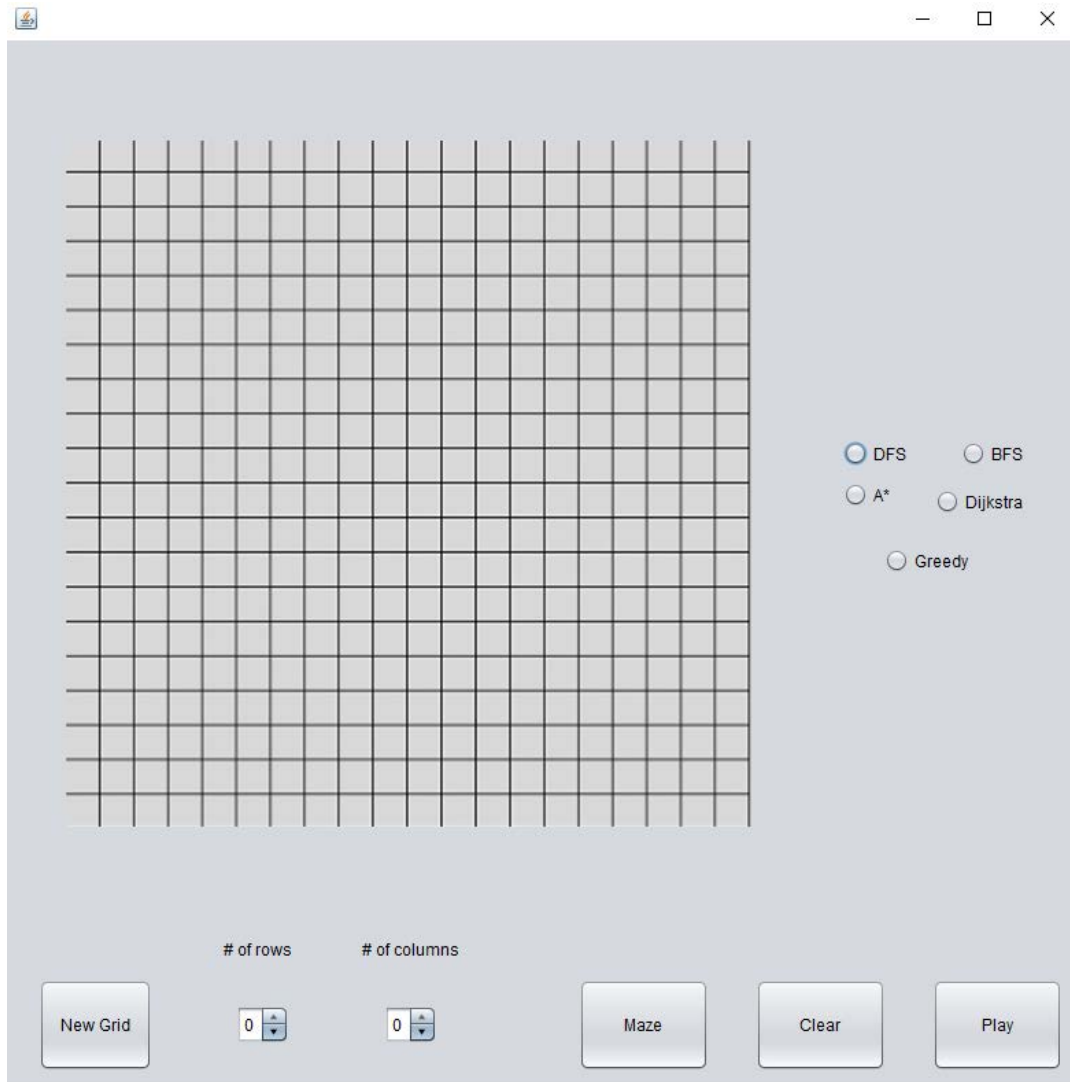
So you like the colours used?

60 responses

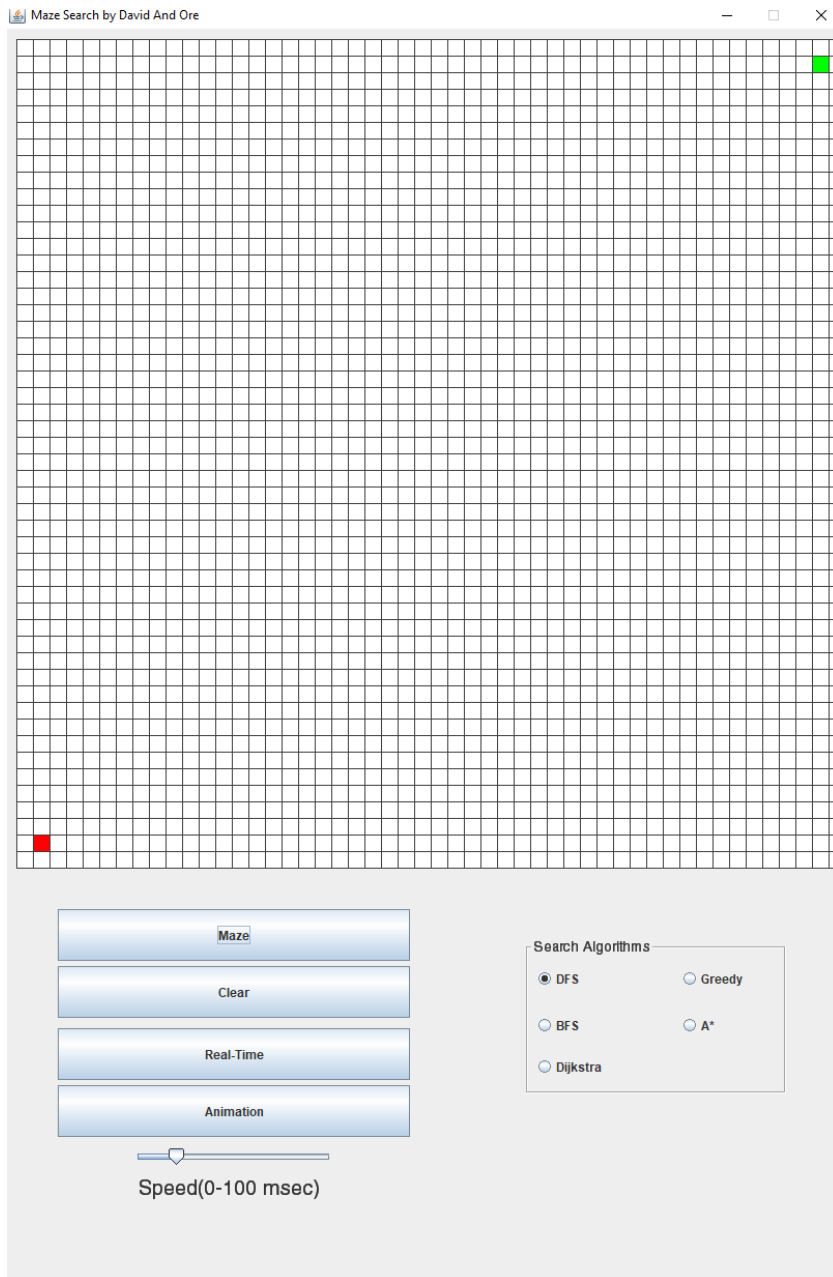


So

This resulted in us going from:



To:



And finally:

Create Maze

Real Time(CPU Time)

Draw Path

Speed(0-1 second)

Clear

Search Algorithms

- ☐ DFS
- ☐ Greedy
- ☐ BFS
- ☐ Dijkstra
- ☒ A*

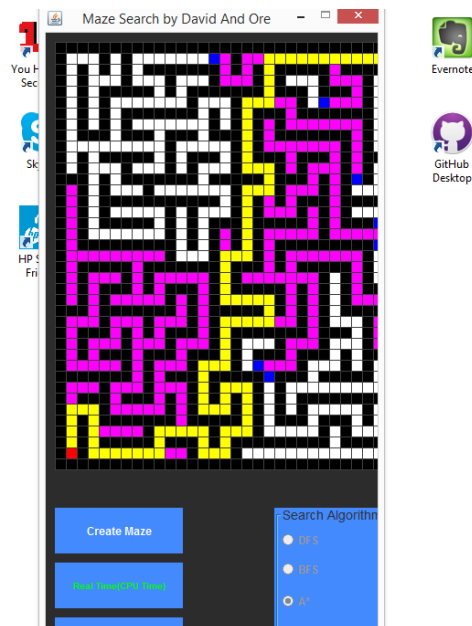
Nodes expanded: 1346, Shortest Path: 604

The final questions

Final Usability Testing:

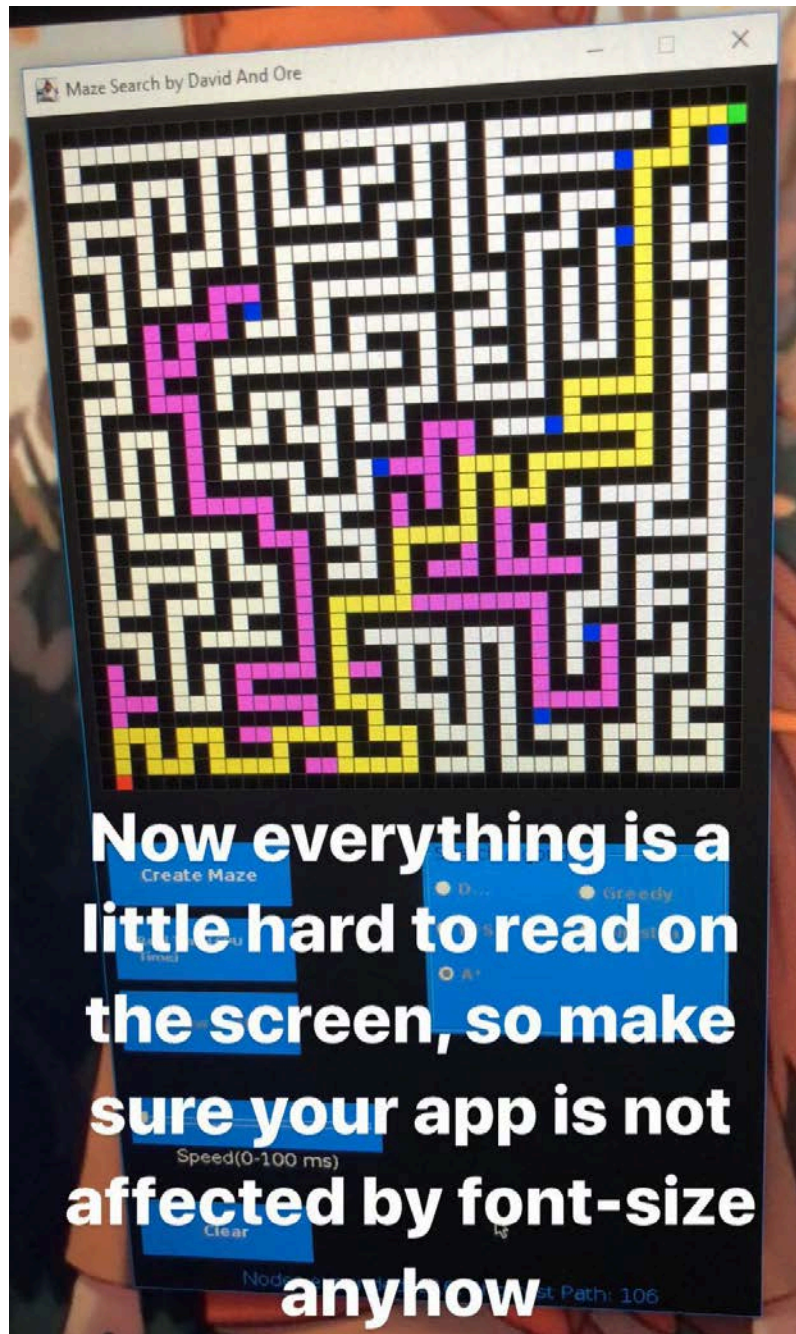
Late into the project we realized that the labs' monitors run on a 1920x1080 resolution.

The final UI design you saw previously is the result of testing on 1920x1080 and scaling down from our original 3440x1080 button layout.



Above is one user running a 1920x1080 display and not being able to see the buttons to run the program. This is because we did not implement a responsive design and thus worked from our native 3440x1400.

Once we found the right positioning, we had an issue with font size. The same user reported:



We adapted and fixed the font issue.

Testing concluded.

