



Department of Computer Science

BSc (Hons) Computer Science

Academic Year 2018 - 2019

**Virtually generating a maze that a robot can find the
shortest path through**

Sebastian Waring

1541110

A report submitted in partial fulfilment of the requirements for the degree of
Bachelor of Science

Brunel University
Department of Computer Science
Uxbridge
Middlesex
UB8 3PH
United Kingdom
T: +44 1895 203397
F: +44 (0) 1895 251686

Abstract

MicroMouse is an event that has been around since the late 20th century, the competition consists of robots the size of mice solving a physical maze, hence the name. The competitors are given a timeframe of 10 minutes to document the maze in the onboard memory. This is then used to solve the maze in the shortest time possible.

There is a long time given to the robot to map out the maze thoroughly. Thus, this project proposes a solution to remove the need for mapping out the physical maze. A camera will be mounted above the maze to provide an aerial view and assist the robot in solving the maze. The secondary objective is to bring portability to the competition, rather than needing a physical maze to test the software a virtual maze will be created and overlaid on the camera, therefore remove the reliance on a physical maze.

In order to develop this solution, the software implemented can generate a maze of differing sizes. Several different algorithms can then be run to find the shortest path through the maze. The design phase provides a simplistic user interface for the user to generate a desired maze of their choosing and the option to solve the maze with any of the algorithms implemented.

Acknowledgements

I would like to thank my supervisor Mahir Arzoky for assisting me through the project with any help that was needed. Regular meetings helped keep the project focused and vitally helped my application throughout.

I certify that the work presented in the dissertation is my own unless referenced.

Signature

Sebastian Waring

Date

05/04/2019

Total Words: 10729 (Not including numbers in tables)

Table of Contents

Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Figures	vi
List of Tables	vii
Chapter 1: Project overview	1
1.1 Introduction	1
1.2 Aims and Objectives	1
1.3 Project scope and research approach	2
1.4 Dissertation outline	2
Chapter 2: Literature review	4
2.1 Introduction	4
2.2 Background	4
2.3 Types of mazes	4
2.4 Maze Solving Algorithms	6
2.4.1 Recursive Backtracker (DFS)	6
2.4.2 Breadth First Search	7
2.4.3 Wall Following	7
2.5 Maze Generation Algorithms	7
2.5.1 Recursive Backtracker (DFS)	7
2.5.2 Kruskal's Algorithm	8
2.5.3 Aldous-Broder Algorithm	9
2.5.4 Eller's Algorithm	9
2.6 Object tracking	9
2.7 Summary	10
Chapter 3: Requirements and approach	11
3.1 Introduction	11
3.2 Research analysis and algorithms chosen	11
3.2.1 Generating algorithms	11
3.2.2 Solving algorithms	12
3.3 Methodology selection	12
3.4 Source control	14
Chapter 4: Design and prototype	15
4.1 Introduction	15

4.2	Existing tools.....	15
4.3	Design documentation	17
4.3.1	Flow diagram.....	17
4.3.2	Software conceptualisation	17
4.4	Initial prototyping.....	17
4.4.1	Java JFrame prototyping	17
4.4.2	Maze display.....	18
4.4.3	Prototype analysis	18
Chapter 5:	Implementation	20
5.1	Introduction	20
5.2	Overview of software implemented.....	20
5.3	Implementing JFrame.....	21
5.3.1	JOptionPane and MazeVariable panel.....	21
5.3.2	Frame Layout and setSize.....	21
5.4	Implementing Kruskal's Algorithm	21
5.4.1	Generate starting set of walls	22
5.4.2	Selecting edges to merge	23
5.4.3	How maze is displayed	23
5.5	Maze solving.....	24
5.5.1	Recursive Backtracker	24
5.5.2	Breadth First Search	25
5.5.3	Wall Follower	26
5.5.4	Maze display.....	27
5.6	Print times.....	28
5.7	Camera Implementation.....	29
5.8	Robot Navigation.....	29
5.9	Testing	31
Chapter 6:	Evaluation	32
6.1	Introduction	32
6.2	Testing Kruskal's Algorithm	32
6.3	Testing Solving Algorithms	34
6.3.1	Testing square maze solving algorithms.....	34
6.3.2	Testing rectangular maze solving algorithms.....	36
6.4	Evaluating the size of the project	38
Chapter 7:	Conclusion.....	39
7.1	Introduction	39

7.2	Objectives of the project.....	39
7.3	Limitations and reflection	39
7.4	Future Work.....	40
References		41
Appendix A	Personal Reflection	43
A.1	Reflection on Project.....	43
A.2	Personal Reflection.....	43
Appendix B	Appendices	44
B.1	Ethics Approval Confirmation.....	44
B.2	Literary review pseudocode and tables.....	44
B.3	Flow diagram	49
B.4	Design mock up.....	50
B.5	Depth First Search Algorithm code.....	51
B.6	Wall Following Algorithm code.....	51
B.7	BFS chart with outliers	51
B.8	Kruskal's Algorithm – Non square mazes	52

List of Figures

Figure 2.1 - Maze shapes (Razimantv, 2017).....	5
Figure 2.2 - Maze represented as graph (Kirupa, 2006)	6
Figure 3.1 - V-Model life cycle	14
Figure 4.1 - Design of generation website (Mazegenerator.net, 2019)	15
Figure 4.2 - Broken maze generation (Feuerborn, 2017).....	16
Figure 4.3 - Broken maze solving (Feuerborn, 2017)	16
Figure 4.4 - User input panel	18
Figure 4.5 - How maze is drawn.....	18
Figure 4.6 - Small generation.....	19
Figure 4.7 - Medium generation	19
Figure 4.8 - Large generation	19
Figure 5.1 – Demonstration of software implemented.....	20
Figure 5.2 - Integer representations.....	22
Figure 5.3 - Maze printed in console	22
Figure 5.4 - Wall selection painted.....	22
Figure 5.5 - Colour representation.....	24
Figure 5.6 - Method to draw maze.....	24
Figure 5.7 - Recursive Backtracker steps	25
Figure 5.8 - Breadth First Search steps	25
Figure 5.9 - Wall follower steps.....	27
Figure 5.10 - Depth first.....	28
Figure 5.11 - Breadth first.....	28
Figure 5.12 - Wall follower	28
Figure 5.13 - Depth first.....	28
Figure 5.14 - Breadth first.....	28
Figure 5.15 - Wall follower	28
Figure 5.16 - Console output.....	29
Figure 5.17 - Overhead camera implemented	29
Figure 5.18 - Robot navigation algorithm	30
Figure 5.19 - Direction check Figure 5.20 - Directional change	30
Figure 5.21 - Maze generated	30
Figure 6.1 - Kruskal's Algorithm - Timed	33
Figure 6.2 - Kruskal's Algorithm plotted exponentially	33
Figure 6.3 - Solving algorithms comparison 1.....	35

Figure 6.4 - Solving algorithms comparison 2	35
Figure 6.5 - Solving algorithms tested to 1000	35
Figure 6.6 - DFS timed by size	36
Figure 6.7 - BFS timed by size	37
Figure 6.8 - Wall follower timed by size.....	37
Figure 6.9 - Lines of code per class	38
Figure 0.1 - Ethics approval	44
Figure 0.2 - Recursive Backtracker pseudocode (theJollySin, 2018).....	44
Figure 0.3 - Breadth First pseudocode (theJollySin, 2018).....	45
Figure 0.4 - Wall Following pseudocode (theJollySin, 2018).....	45
Figure 0.5 - Recursive Backtracker pseudocode (theJollySin, 2018).....	46
Figure 0.6 - Recursive Backtracker pseudocode (theJollySin, 2018).....	46
Figure 0.7 - Kruskal's Algorithm pseudocode (Jeulin Lagarrigue, n.d.)	46
Figure 0.8 - Aldous Broder pseudocode (theJollySing, 2018).....	47
Figure 0.9 - Eller's Algorithm pseudocode (theJollySin, 2018)	47
Figure 0.10 - Table comparing generation algorithms (Pullen, 2015).....	48
Figure 0.11 - Flow diagram	49
Figure 0.12 - Design mock up	50
Figure 0.13 - Camera assisting robot.....	50
Figure 0.14 - Recursive solve code	51
Figure 0.15 - BFS with outliers	51
Figure 0.16 - Kruskal's algorithm plotted	52

List of Tables

Table 2.1 – Perfect maze.....	5
Table 2.2 – Braided maze	5
Table 2.3 – Partially braided maze	5
Table 5.1 - Wall selection printed.....	22
Table 5.2 - Algorithm printed in console	23
Table 5.3 - Kruskal's Algorithm painted	23
Table 5.4 - Matrix painted in JFrame.....	24
Table 5.5 - Recursive Backtracker printed.....	25
Table 5.6 - Breadth First Search printed.....	26
Table 5.7 - Breadth First Search real print.....	26
Table 5.8 - Follow wall algorithm printed.....	27
Table 5.9 - Test cases	31

Chapter 1: Project overview

1.1 Introduction

Mazes have been around since the 5th century, showing up in Greek mythology and Roman art, (National Building Museum, 2014). Around the 1950's the robotics field began using mazes prevalently to pose a challenge to robotic navigation. In the late 1970's a competition called MicroMouse was created to challenge developers to solve a maze in the fastest time with their robot, (Micromouse Online, 2019). This competition uses the onboard detection of the robot to map out the physical maze, using this knowledge to solve the maze in the shortest time.

Instead of a large physical maze, this project aims to bring portability to the competition by using a randomly generated virtual maze that the robot will solve. Without any pre-existing knowledge of the maze the robot is forced to take random directions to gain knowledge about the maze, which it would then use to reconstruct the maze and come up with the optimal solution (Mishra and Pande, 2008). The inability of the robot to see the whole maze means the most time consuming aspect of solving the maze is mapping it out (Dracopoulos, 1998).

This project will implement generating algorithms to generate the virtual maze, then use solving algorithms to solve the maze in the shortest time, transferring the optimal path through the maze to the robot.

This project aims to use visual assistance to help navigate the robot through the maze. This will remove the time wasted by the robot not travelling down paths that are dead ends. Instead the computer will solve the maze and relay the correct path to the robot. The robot will use object detection to position the robot at the beginning of the maze and keep it within the walls while following the path.

1.2 Aims and Objectives

The overall aim of this project is for a maze to be generated virtually, that assists a robot to navigate through the maze and reduce the time taken to solving the maze. To achieve this goal the objectives will have to be met first. These objectives are as follows:

- I. To conduct a thorough literature review on maze generation and solving algorithms identifying which would be best for this project.
- II. Design an interface, which allows the user to generate their own sized maze.

- III. Randomly generate mazes, with the purpose of testing the speed of algorithms. The mazes will need several difficulties to thoroughly test the different algorithms and find out which is best. There should also be an easier difficulty to test the robot.
- IV. Implement different algorithms that can solve the generated mazes. The metric for efficiency will be the route in and out of the maze with the least blocks used.
- V. Image detection will need to be capable of tracking the robot as it progresses through the maze, and able to detect whether the robot has hit a wall or not.
- VI. Write up a detailed report on the project explaining the choices made throughout and any difficulties encountered.
- VII. To evaluate the project against the initial objectives to find out whether the overall aim is met.

1.3 Project scope and research approach

The aims of this project span many different areas of computer science, from creating and visualising algorithms, to tracking moving objects, collision detection and robotic navigation. Limitations will also be introduced to help shorten the project and make it achievable to complete in the timeframe given.

One limitation will be to use a virtual maze rather than a physical maze. A virtual maze will allow for randomly generated mazes to be displayed quickly, using a projector to display the maze will make it portable and allow the size of the maze to change with little effort.

The approach for this project will be to use the V-Model methodology. This model was chosen as it benefits the projects with a fixed timeframe that can't be delayed or pushed back. With regular deadlines and milestones in place it is easy to tell if the project is falling behind. The requirements of the project should be set early on and should not change throughout the course of the project. The technology isn't going to change during my project as robotics around maze development has been around for decades and has stayed the same with changes only affecting the robots, as the technology improves.

1.4 Dissertation outline

The dissertation consists of 7 chapters as explained below:

Chapter 2 – Literature review: This chapter will explain all the important information found out during researching generating and solving algorithms as well as visual tracking and robot navigation.

Chapter 3 – Requirements and approach: This chapter explains the reasoning behind choices made about the project. It will also discuss which methodology was chosen for this project and why, as well as the ethics needed for the project.

Chapter 4 – Design and prototype: This chapter will explain the design choices made and why, as well as evaluating the first prototype developed.

Chapter 5 – Implementation: This chapter will go over in detail everything implemented during this project, expanding on the prototype and explaining where difficulties were found and if they were overcome then how.

Chapter 6 – Evaluation: This chapter tests the generation algorithm implemented, the solving algorithms used on the mazes.

Chapter 7 – Conclusion: This chapter details whether the aims and objectives of the project have been met and if so why. Limitations and future work are discussed once the project has been completed.

Chapter 2: Literature review

2.1 Introduction

This chapter aims to provide a detailed report on the literature reviewed and the knowledge gained as well as a brief background on mazes. The research will be conducted to gain knowledge on the following topics: mazes, algorithms for generating a maze, algorithms for solving a maze, visual detection of objects and robotics.

2.2 Background

It's impossible to pinpoint when mazes were first invented, as the earliest evidence points back to the 5th century where coins with labyrinths carved into them were discovered (National Building Museum, 2014). Greek mythology also has references to mazes as the tale of the Minotaur involves the beast being locked in a huge maze and volunteers would enter the maze to fight the monster. In the story Theseus would enter the maze with a long piece of string, attaching this string to the entrance of the maze allowed him to retrace his steps out of the maze once he had slain the monster (Garcia, 2013). While this was a myth, some of the algorithms covered at a later stage still use this concept.

Similarly, the shortest path problem was inherently a primitive problem as the quickest route to food or water would be found (Schrijver, 2012). Maze solving has been around in mathematics since before the 20th century with maze solving being similar to the 'Traveling Salesman Problem' and other path finding problems. In the 1950's the robotics field turned their attention towards the maze solving problem.

In 1977 a new competition was announced 'Amazing Micromouse Competition'. It took place in 1979 in New York, there were 6000 robots had to find the exit of a 10 by 10 maze, (Micromouse Online, 2019). Consequently, there was a flaw with the competition as a high-speed but 'dumb' wall following algorithm ended up winning. Next year the competition was held in London, the rules were changed slightly so the robot had to find the end point located in the centre of the maze. Loops were also added to the maze to prevent wall following algorithms from succeeding. The record in 1981 was 47 seconds, now the competition has changed even more with the maze being 21 by 21 and the fastest as of 2018 was in 5 seconds, (Micromouse Online, 2019).

2.3 Types of mazes

Mazes come in many different varieties, the most well-known are the tourist attractions that commonly use hedges as the walls and the participant tries to find the centre of maze or a way out. As the volunteer cannot see above the hedges, they will make their way towards the goal by using randomised back tracking (trial and error) or they can try using the follow the wall method

of keeping one hand on the hedge. These mazes normally only have one route to solving the maze all other turnings lead to dead ends. Labyrinths share similarities to mazes but have one long snake like passage with no wrong turns (Pullen, 2015). There are smaller physical mazes like in the MicroMouse competition which has multiple different paths to the destination and loops throughout. A maze that contains only loops and no dead ends are referred to as 'braids', these mazes would be the hardest to solve as they have many more choices to make throughout (Buck, 2015). A 'perfect' maze would be a maze that has only dead ends without any inaccessible areas, any two points on the maze will have only one possible path between them. A mixture of these two types of mazes will be called a 'partial braid', for example the hedge maze will be a perfect maze, while Micromouse is a partial braid.

1	1	1	1	1	1	1
1	0	0	0	1	0	1
1	1	1	0	1	0	1
1	0	1	0	0	0	1
1	0	1	0	1	1	1
1	0	0	0	0	0	1
1	1	1	1	1	1	1

Table 2.1 – Perfect maze

1	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	1	1	1	0	1
1	0	0	0	1	0	1
1	0	1	0	1	0	1
1	0	0	0	0	0	1
1	1	1	1	1	1	1

Table 2.2 – Braided maze

1	1	1	1	1	1	1
1	0	0	1	0	0	1
1	1	0	1	1	0	1
1	0	0	0	0	0	1
1	0	1	0	1	0	1
1	0	0	0	1	0	1
1	1	1	1	1	1	1

Table 2.3 – Partially braided maze

Tables 2.1, 2.2, 2.3 show the differences between the mazes. As the project uses a virtual maze instead of physical maze the rest of this research will be focused on 2d mazes where an overview of the maze is visible, this allows for a more structured approach to solving the maze. Mazes come in many different shapes, circular, honeycomb, hexagonal or rectangular as show in figure 2.4. Circular mazes are often used in robotics as it makes the movement and detection more challenging, without straight paths.

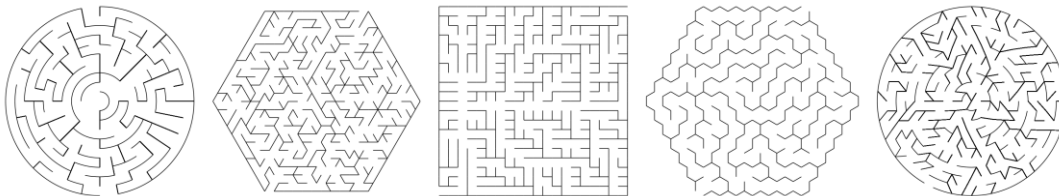


Figure 2.1 - Maze shapes (Razimantv, 2017)

However, this project will aim to use rectangular mazes due to there being a wealth more research on these types of mazes, only some generation and solving algorithms would work on a maze that aren't rectangular. Furthermore, a rectangular maze would effectively use the space of a rectangular projection.

2.4 Maze Solving Algorithms

There are many different solving algorithms which solve many different types of mazes. This section will only cover the algorithms that are most likely to be implemented. The literacy for this project will focus on the most popular, the fastest or whether it finds the shortest path.

2.4.1 Recursive Backtracker (DFS)

Recursive Backtracker (RB) is a variant of a Depth First Search (DFS), RS is the most known as it is the best implementation in terms of maze solving, however, there are hundreds of different applications for DFS, (Algolist.net, n.d.). The DFS algorithm was initially created for searching a tree or graph data structures as shown in Figure 2.2.

Perfect mazes can be represented as trees or graphs as every dead end is just a branch from the main path. While braid or partial braid cannot be represented in this way as there will be loops in the branches causing multiple different solution.

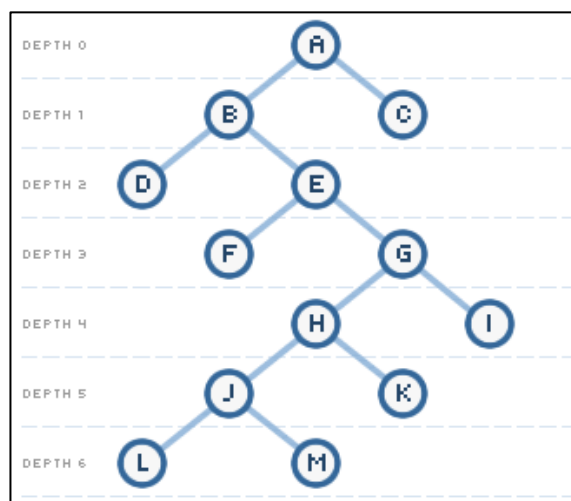


Figure 2.2 - Maze represented as graph (Kirupa, 2006)

The pseudo code for this algorithm can be found in the Appendix (B.2 – 1). 7.4B.2This algorithm works by selecting a branch and exploring all possible nodes, if it reaches a dead-end, back track until another branch is available. This algorithm will find a solution very quickly and is the most implemented algorithm due to its speed and ease of implementation. Due to the way the algorithm searches until the exit is found. This means if there are multiple solutions to the maze it may not find the shortest. However, if the maze is perfect, meaning only one solution then it will find the shortest path.

2.4.2 Breadth First Search

Breadth First Search (BFS) also referred to as shortest path finder or flood fill, is described very well by its many names. Breadth first refers to exploring all available branches at the same time, similar to how water would flow through a maze.

The pseudo code for this algorithm can be found in the Appendix (B.2 – 2). The algorithm will search all paths and once reaching a dead-end will stop searching that path. All other paths will carry on searching until the end is found. This algorithm is simple to implement yet slow. If the algorithm goes from one end of the maze to the other it will cover 99% of the maze as it explores every possibility. As it explores every possibility it will find the shortest path of any maze, perfect or braided (Opendatastructures, n.d.).

2.4.3 Wall Following

Wall following is the simplest algorithm, not in terms of implementation but so simple that a human can do it. Put your hand on the side of a wall, keep it there walk forwards and you should end up at the exit. Similarly, with a robot it is extremely easy to implement as all it must do is turn one way. Both left and right versions of this algorithm are the same, with the variation being on which wall is followed.

The pseudocode for this algorithm will be in the Appendix (B.2 – 3). Step 3 of the pseudocode is not always included, this step tracks to see if it is stuck in a loop by returning to the same place more than once. Thus, the algorithm will only work for perfect mazes. While this algorithm is regarded as a bad algorithm due to its cheat like behaviour, it can be fast as in the first Micromouse competition it did come first. This algorithm will find one long path to the exit, it won't find the shortest path or anywhere near, unless there are no dead-ends. A chart created by Walter Pullen, 2015 can be found in the Appendix (B.2 – 4), that compares many different solving algorithms.

2.5 Maze Generation Algorithms

There are many different maze generation algorithms which generate all different types of mazes, the for generation has covered over 35 different algorithms, but this section will only cover the most useful.

2.5.1 Recursive Backtracker (DFS)

The DFS can be adapted to also work as a generation algorithm. Similarly, to generation it will only work for generating perfect mazes. DFS has different variations, iterative and recursive back

tracker, (Reichert, 2019) the only difference between the two is what type of queueing is used. Recursive backtracker is the go-to algorithm as it is easy to implement and fast to generate, on the other hand, significant memory will be needed for large mazes as it must store the entirety of the maze, (Buck, 2010).

The pseudocode for this algorithm can be found in the Appendix (B.2 – 5), the pseudocode is similar to the solving algorithm, with differences being that instead of picking from branches to explore, it randomly chooses which cells to create a passage through and recursively backtracking until back at the start. The look of the maze is very aesthetically pleasing with no bias; however, it creates fewer dead ends as they tend to weave longer passages. An interactive display of the algorithm is available (Buck, 2011). This algorithm produces the smallest amount dead ends and is the 5th quickest to generate, Appendix (B.2 – 9). Breath first is a deviation from this algorithm which instead of creating the passages separately, will create them all at once, however this is slower and harder to implement.

2.5.2 Kruskal's Algorithm

Kruskal's Algorithm was developed by the mathematician and computer scientist Joseph Kruskal in 1956 (Buck, 2015). This algorithm will create a minimum spanning tree which is another way of representing a perfect maze, other variants like Prim's and Boruvka will not be covered as they are very similar but are much harder to implement (Reichert, 2019). These may be looked at later on when the easier algorithms have been implemented. There are two different versions to the algorithm, there is weighted and randomised.

The pseudocode for this algorithm will be in the Appendix (B.2 – 6). The algorithm works by starting off with a set of edges and merging the edges to create the maze. However, there can be a difference between the way the edges are chosen, either randomly chosen or weighted. Jamis Buck referred to this algorithm as "second in difficulty" when it comes to implementation (Buck, 2011). Looking at the research in (Buck, 2011), (Reichert, 2019), (theJollySin, 2018), (Dreibholz, n.d.) and (Jeulin-Lagarrigue, n.d.) they all implemented the randomised version rather than the weighted version, as (Buck, 2011) says "Making that change, the algorithm now produces a fairly convincing maze". An interactive display of the algorithm is available (Buck, 2011). Looking at Pullen's table Appendix (B.2 – 9), this algorithm on average creates triple the number of dead ends as the recursive backtracker and will not be restricted to one style of generation. The memory and time to generate are very similar when compared with the recursive backtracker.

2.5.3 Aldous-Broder Algorithm

The Aldous-Broder algorithm is one of the simplest algorithms to implement. D. Aldous and A. Broder worked independently on uniform spanning trees and independently arrived at this algorithm (Buck, 2011).

The pseudocode for this algorithm will be in the Appendix (B.2 – 7). This algorithm starts at an edge and randomly chooses a neighbouring cell and visits it, however it will not consider already visited cells. Hence, the algorithm will revisit the same cell many times. An interactive display of the algorithm is available (Buck, 2011). As there is no memory stored of where it has been the algorithm takes very little memory and is quick to code. Pullen did not include this algorithm in his table of comparisons due to it being so inefficient and random.

2.5.4 Eller's Algorithm

Eller's algorithm is the most difficult algorithm to implement, as it runs very quickly, the speed of creation does not change massively depend on the size of the maze, meaning it is able to generate mazes of infinite sizes in linear time (Buck, 2015).

This pseudocode is the hardest to understand and will need to be covered in greater detail if implemented. This algorithm is faster than all the other generation algorithms covered, it also can change how it is generated based variables. This can be very helpful as normally two different algorithms would be implemented to generate differing mazes.

2.6 Object tracking

Object tracking will be used for detecting whether the robot is following the correct path and not running into any walls. The robot is large and can be brightly coloured to help with the visual tracking. When researching image capture, object tracking and recognition there were many open source libraries providing these features. The main libraries found were Open CV (Opencv.org, 2019), SimpleCV (Simplecv.org, 2019) and BoofCV (Boofcv.org, 2019). OpenCV is the biggest library able to be implemented in C++, Python and Java, the library is an “open source computer vision and machine learning software library” however, this project will only be using the computer vision aspect. SimpleCV is an “open source framework for building computer vision applications”, this library builds on OpenCV's computer vision and simplifies the code for new programmers. Unfortunately, the library is only documented and coded in Python. BoofCV is a library that would be used for higher level programming, as it boasts faster algorithms with the drawback of being harder to implement and less documentation.

2.7 Summary

This chapter focuses on researching all the different types of mazes that could be used. As well as choosing the best generating and solving algorithms could be implemented. The different libraries what would be need for image detection and object tracking.

Chapter 3: Requirements and approach

3.1 Introduction

This chapter will break down all the research in Chapter 2 and make decisions on the direction the project will take. The methodology chosen and ethics will also be discussed as these are needed before the project can begin.

3.2 Research analysis and algorithms chosen

After concluding a thorough literature review, decisions were made on how this project will be implemented moving forwards, these decisions will be explained in this subsection.

This project will focus on only perfect mazes as there is a wealth of literature on the topic of solving and generating perfect mazes. A perfect maze can be represented as a tree in graph theory as there are only branches from the main path with no loops (Ioannidis, 2016). Braid generation can generate on top of a perfect maze (Buck, 2015), this will save time instead of creating a whole new algorithm but should only be implemented if necessary. MircoMaze started with a perfect maze Section 2.3 and changed to a partially braided maze after a simple wall following algorithm took first place. As this project aims to implement multiple solving algorithms, one algorithm should be implemented that is faster than a wall follower rather than changing the maze to stop these algorithms from succeeding. As the robot will be entering the maze and exiting the maze at the end the generating algorithm should be capable of achieving this.

3.2.1 Generating algorithms

From the research conducted, the best algorithms to be implement would be Kruskal's Algorithm and Recursive Backtracker, these are fast algorithms which provide differing perfect mazes with more and less dead ends respectively, both are aesthetically pleasing with no bias. These algorithms would provide differing levels of difficulties for solving algorithms. Kruskal's algorithm is harder to implement so it may be best to implement the Recursive Backtracker first.

If there is time after finishing the main stages of the project, then implementing Aldous-Broder will be a good idea as it is easy and makes evaluating the algorithms better than just comparing the two. The final algorithm Dead-end Culling is an algorithm that changes perfect mazes to braid (Buck, 2015). There aren't many algorithms for braid implementation this is the only one I could find. Otherwise, randomly taking down walls will also work and is much easier to implement, however, this will only work for partial braids.

Generating algorithms to be implemented:

- 1) Kruskal's Algorithm
- 2) Recursive Backtracker

If there is time:

- 1) Aldous-Broder
- 2) Dead-end culling

3.2.2 Solving algorithms

As the two generating algorithms to be implemented will generate perfect mazes, then the first two solving algorithms to be implemented will work for perfect mazes. The first algorithm implemented will be the recursive backtracker as it is fast and easy to implement. This algorithm should be faster than the second one implemented, a wall follower.

The last two solving algorithms will be implemented if there is time after finishing the main part of the project. These would be algorithms that can solve braided, or partially braided mazes, as the dead-end filling algorithm should have been implemented and the current algorithms won't be able to find the shortest path. The last algorithm would need more research done into it. However, two of those algorithms have been implemented in the Traveling Salesman Problem from the second-year module Algorithms and their Application.

After researching into the multitude of libraries available, the library that will be implemented will be OpenCV, this is because there is greater functionality and documentation with being integrated with Java. SimpleCV is very useful and would be used if there was an easy way to implement it into Java.

Solving algorithms to implement:

- 1) Recursive Backtracker
- 2) Wall follower

If there is time:

- 1) Breadth first (shortest path finder)
- 2) A* Search / Dijkstra's Algorithm / Hill Climbing

3.3 Methodology selection

Choosing a software development life cycle can be difficult as there are many different approaches, providing different strengths and weaknesses to the project. There are three main methodologies when it comes to managing software, Waterfall, Iterative development (RAD) and

Agile (Williams, 2017). Waterfall methodologies set clear sequential phases and milestones where no phase begins until the previous phase has ended. This is very simplistic and easy to understand, with the downside being poor choices made early on will be hard to rectify other than starting over (Software Integrity Blog, 2017). Agile and Iterative development are both newer methodologies created to handle very rapid development. This allows for flexibility of requirements and multiple design phases that adapt over time.

This project will not benefit from an iterative life cycle, as the algorithms that will be implemented have been implemented many hundreds of times over. The last phase of the project, robot navigation could benefit from an iterative style to constantly improve the navigation to reduce the time it takes to solve the maze. However, this will be towards the end of the project and time to iterate will be extremely limited. The Waterfall methodology that best suited this project is the V-Model. This model was chosen as the requirements and technology aren't going to be changed during the project, this project won't be heavily design focused and the designs won't be likely to change. This project is heavily implementation driven with strict deadlines for phases to be completed by, with the overall deadline being to be completed before the 5th of April, which is what V-Model is designed to benefit. The disadvantages of this methodology are negated by my project, the requirements are set out in advance and will not be changed. Testing will be performed throughout the implementation as phases are completed, however, the project will be tested again once it has finished. The project is short term with a 6-month window, so won't suffer from a lengthy life cycle, (Powell-Morse, 2016).

In Figure 3.1 there has been adaptations to the normal structure of V-Model, this is because the normal methodology is created to be used by large companies with multiple developers or teams. A deviation made was changing "Detailed Design" to "Design and prototyping", this is because the project won't have a lengthy design phase, due to the algorithmic, visual tracking and robot navigation phases not needing to be designed, only how the user may interact with the software. The prototyping/learning phase is necessary due to inexperience in the Java coding language. Secondly changing the "System Verification and Validation" to "Writing documentation" as the dissertation will be written up towards the end and there won't be a use for a verification phase.

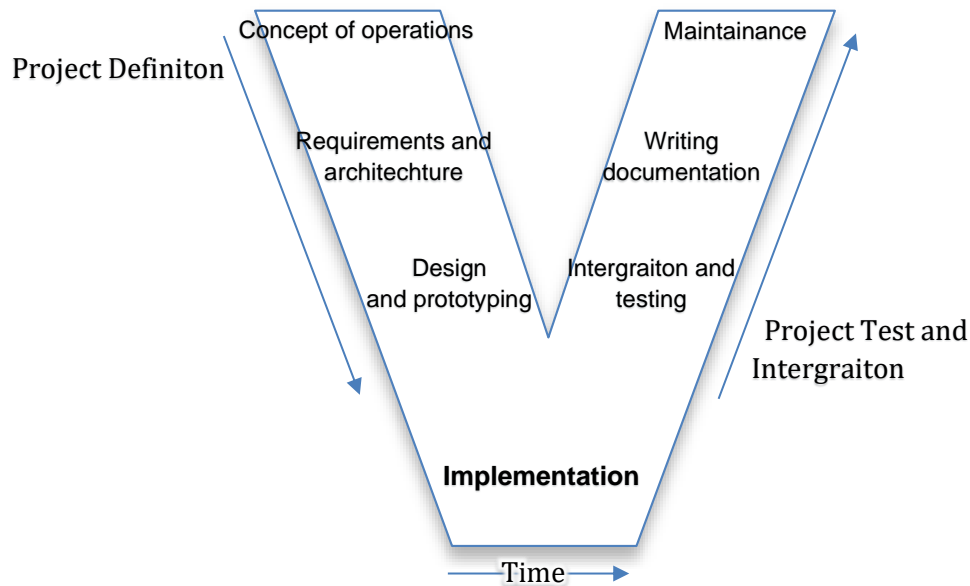


Figure 3.1 - V-Model life cycle

This project will not need human participation to evaluate the efficiency of the project. This limits the need for ethical guidance, nevertheless all research conducted will abide by the guidelines given and a conscious effort will be made to keep the work area tidy and keep it safe for others to use. The conformation for this project not requiring research ethics approval is screenshotted in the Appendix, (7.4B.1).

3.4 Source control

GitHub will be used as the source control management system, it provides version control by having a running history of code development. GitHub also allows the project to be worked on by many different devices, this is beneficial when working from different locations. This allows code to be sent to relevant parties without the need for downloading and importing. The project will be uploaded to github.com/sebmins/MazeGenerator, when milestones of development are achieved (Sebmins, 2019).

Chapter 4: Design and prototype

4.1 Introduction

This chapter aims to research projects that are similar and analyse their designs, use this knowledge to create a design for the software, then prototype the design to improve knowledge in Java before starting implementation.

4.2 Existing tools

The idea of the project is to improve the time it takes to solves mazes with a robot by removing the use of its onboard memory. When designing the software for this project, it must be designed around the requirements. Having a look at already developed software gave an insight into what is commonly expected in the layout. Firstly, looking at online generators these had a slightly different layout than software. Figure 4.1 displays the user input is above the generation; this website allowed many different shapes and styles but failed to explain them well, as they didn't change anything, it may be because these were only meant to work with certain shapes.

The screenshot shows the 'Maze Generator' website interface. At the top, the title 'Maze Generator' is displayed. Below it, there are several configuration options: 'Shape' set to 'Rectangular', 'Style' set to 'Orthogonal (Square cells)', 'Width' set to 5, 'Height' set to 5, 'Inner width' set to 0, 'Inner height' set to 0, 'Starts at' set to 'Bottom or inner room', and 'Advanced' settings for 'E' and 'R' both set to 100. There are 'Like' and 'Share' buttons, and a note that '3.6K people like this'. Below these are links for 'About', 'Help', 'Examples', 'Donate', 'Commercial use', and 'How tos'. A 'Generate new' button is also present. The generated maze is titled '5 by 5 orthogonal maze' and has options for 'Solution', 'As lines', and 'PDF (A4 size)', along with a 'Download' button. The maze itself is a 5x5 grid with a complex path. A mouse cursor is visible over the maze. At the bottom, there is a copyright notice: 'Copyright © 2019 JGB Service'.

Figure 4.1 - Design of generation website (Mazegenerator.net, 2019)

The second website researched had a similar design with the input being at the top and the generation being below, however the generation for this maze was animated, this is always best as it shows the algorithm it used to generate. From observing the generation; the developer most likely implemented the recursive backtracker algorithm. The colours were also very visually pleasing, as well as the blocky maze rather than the generic white and black with thin white walls.

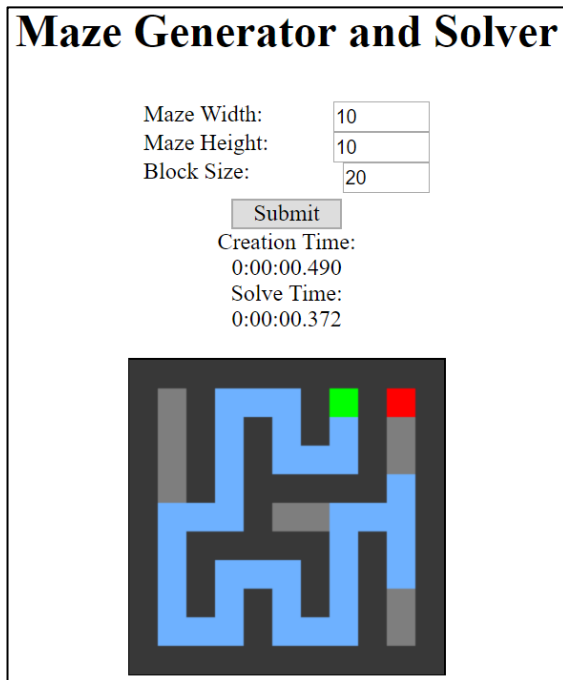


Figure 4.3 - Broken maze solving (Feuerborn, 2017)

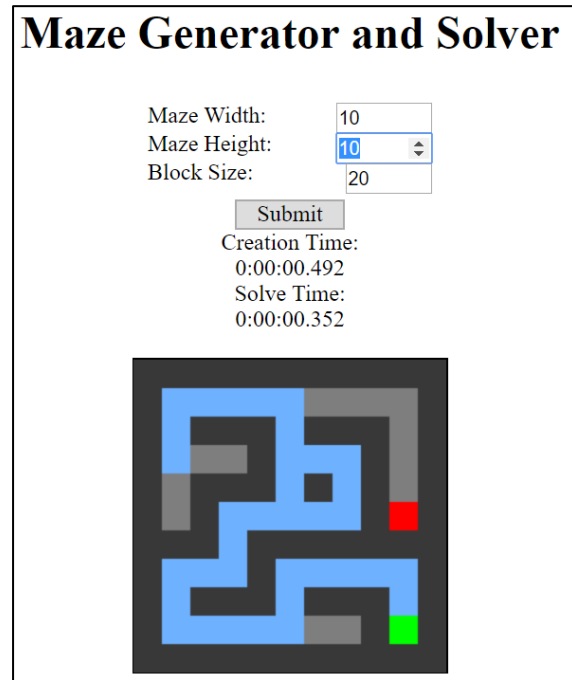


Figure 4.2 - Broken maze generation (Feuerborn, 2017)

However, there were a few issues with the website, as represented in Figure 4.2 the generator managed to create a loop which shouldn't be the case with a recursive backtracker, this ended up breaking the solving algorithm. Notably, in Figure 4.3 even without a loop in the generation the algorithm still got stuck, this shows that implementing a bug free program will need extensive testing, as this wasn't reproduceable once refreshing the page. Furthermore, the user input for width and height would round up if it was even. This is noticeable in either of the figures as the input was 10 by 10 while it printed a 11 by 11.

The next two looked at were software, developed in Java and Win32. Looking at the layouts in (Cylog.org, 2004) and (Ics.uci.edu, 2012) the user input is along the sides rather than on top. This could be due to websites needing to adjust the layout for mobile users. Both have used check boxes to give the user more choice, the check boxes have been used to turn on or off the animation or solve the maze after generation and to show the visited cells for the solving algorithm. If there is time to implement these features, it will give the software greater customisability.

4.3 Design documentation

This Sub-Section covers the documentation created in the design phase. The flow diagram and the software conceptualisation are listed in the Appendix B.3 and B.4, these were created and then implemented in the prototyping phase.

4.3.1 Flow diagram

From the research conducted a rough flow diagram was created to set out the steps that the user can take, this can be found in the Appendix (7.4B.3). The user should first choose what generating and solving algorithm they would like, then choose the difficulty. The difficulty will have pre-set values the first difficulty will be extremely easy with usually only one wall, this will be to test out the robot. Medium will be 21 by 21 which is the same as the MicroMouse tournament. Then 100 by 100 will be used to test algorithms as it provides a greater challenge, hence taking longer. The user will also be able to decide whether they want animations or not, this is for the sake of development as well as being a useful feature. The software will in the background run all other algorithms and display if they succeeded and how long they took. Finally, the user can decide if they want the robot to use the path found to solve the maze or try generating a new maze.

4.3.2 Software conceptualisation

Using the flow diagram created in Section 4.2.1 a mock-up was created, this can be found in the Appendix 7.4B.4). This will be the general structure aimed to be implemented, though this will be limited to coding ability and priority will be to implement the requirements as quickly as possible. When designing the user input panel was placed on the right-hand side as if the maze is going to square (equal number of rows and columns) most of the time, this will effectively utilise the resolution of a computer screens.

4.4 Initial prototyping

As stated in Section 3.3, the methodology has set aside time to learn and prototype in Java. Java was chosen as a programming language as it works with all the robots provided by the University, there are also many libraries that work with image detection in Java. However, during the research not many developers used Java opting for Ruby, Python or C#, this could be due to Javas' GUI being limited. Without any experience in Java for over a year an initial learning phase will be useful to gain back knowledge.

4.4.1 Java JFrame prototyping

In the design mock-up there are two sections to the software, the user input and the maze. The maze will take up much of the development time, so starting with user input made sense, leaving a placement area for where the maze can be integrated at a later stage. Learning from several

YouTube tutorials a very basic side panel was created, creating a new JFrame in the main class, then in another class that extends JFrame using the BorderLayout and a Container to separate each section and place the Maze Variables on the east side of the maze. This allowed the maze and variables to be created in separate classes. Finally, I created a Variables class which used a GridBagLayout to neatly position all the JComponents.

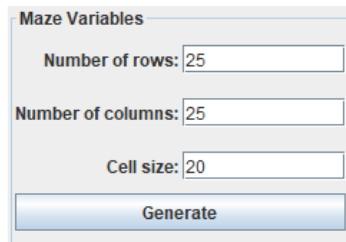


Figure 4.4 - User input panel

4.4.2 Maze display

Since the frame was designed and the rest of the functionality will be added depending on the maze. As this prototype was meant to focus on improving and developing Java skills and not to develop further understanding of generation algorithms, the Aldous-Broder algorithm was used. The reasoning for this was to draw the maze Java uses a paintComponent method that will be the focus of the prototype not the algorithm. However, despite the algorithm meant to be the easiest to implement it took a significant while to get working. To paint the maze into the Container a method was created that implemented Graphics2D and Path2D. The paint method worked by drawing a line of length which the user could choose, getting the information from the algorithm seen in Figure 4.5. This was difficult to implement as the visualisation of the maze will only appear once the algorithm had been completed, meaning if there was an error nothing will be displayed, and it will be hard to find out what went wrong.

```
if (m1.cells[i][j].walls[0] == 1){  
    mazeShape.moveTo(x, y);  
    mazeShape.lineTo(x + cellSize, y);  
    g2d.setColor(Color.BLACK);  
    g2d.drawLine(x, y, x + cellSize, y);  
}
```

Figure 4.5 - How maze is drawn

4.4.3 Prototype analysis

This prototype for creating mazes turned out better than expected. The rows and columns the user could input was unlimited, only restraint being size of screen, as higher numbers than 2000 by 1000 at cellsize 2 would go off the screen. There isn't any way of getting around this other than getting a bigger screen size, however, that number of rows and columns is more than enough. This was easy to implement and negates the need for a difficulty slider, as it gives the user more

freedom. One issue that was not able to be fixed, due to the short time constraint of the prototype was when generating a new maze, a new JFrame would be created. This was because the method to initialise the JFrame would be recalled on generate, to stop this from happening the JFrame would have to be in separate class and be a global variable. This wouldn't work with the system as changing the size would break the maze, this would have to be implemented again with the global variable in mind.

The biggest oversight was the maze wasn't built to be solved easily, as the maze was just drawings of lines, solving the maze would only be possible by having the solving path not overlap a line. This is very unnecessary and there are much easier ways to display a maze. This prototype was not meant to cover the algorithm implemented, this will be covered in the implementation phase. However, the algorithm is meant to be extremely slow, yet generating the maze for below 200 rows and columns wouldn't result in any noticeable delay. Figures 4.6, 4.7, 4.8 show the prototype generating different sized mazes.

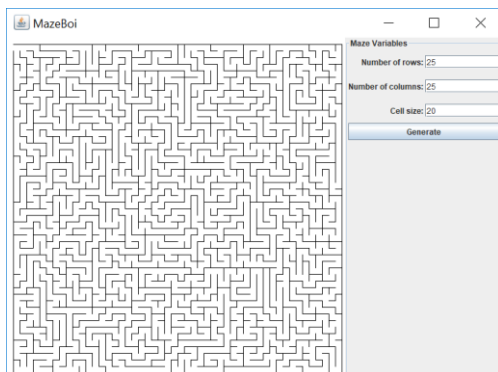


Figure 4.7 - Medium generation

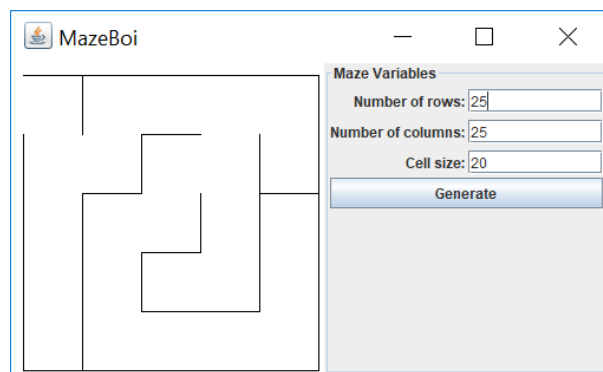


Figure 4.6 - Small generation

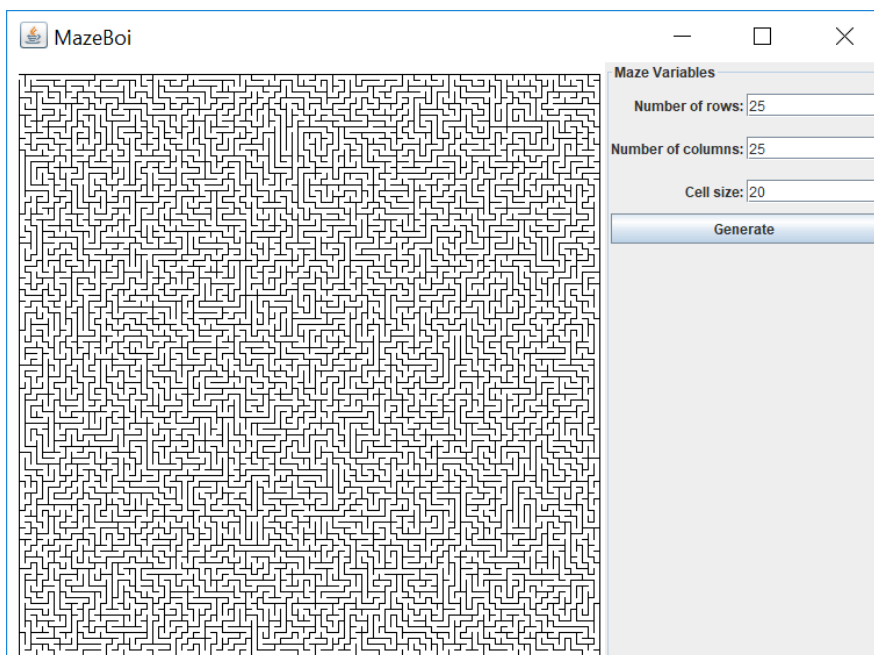


Figure 4.8 - Large generation

Chapter 5: Implementation

5.1 Introduction

This chapter will break down the implementation of each phase of the project, the full source code of this project is available on GitHub as this was the source control chosen, (Sebmins, 2019).

5.2 Overview of software implemented

The software was developed in Java using the Eclipse IDE. The project consisted of 9 classes which will be broken down and explained in this Chapter. All the code for this project was uploaded to GitHub after completion of a milestone, (Sebmins, 2019). In Figure 5.1, a demonstration of the software is shown. Displayed on screen, is a 7 by 7 maze that has been generated using Kruskal's Algorithm, then solved using Depth First Search (highlighted in green). The camera was mounted on the ceiling of the Wilfred Brown building, however, as seen in the Figure 5.1, the Finch robot still overlapped slightly despite using the biggest possible cell size.

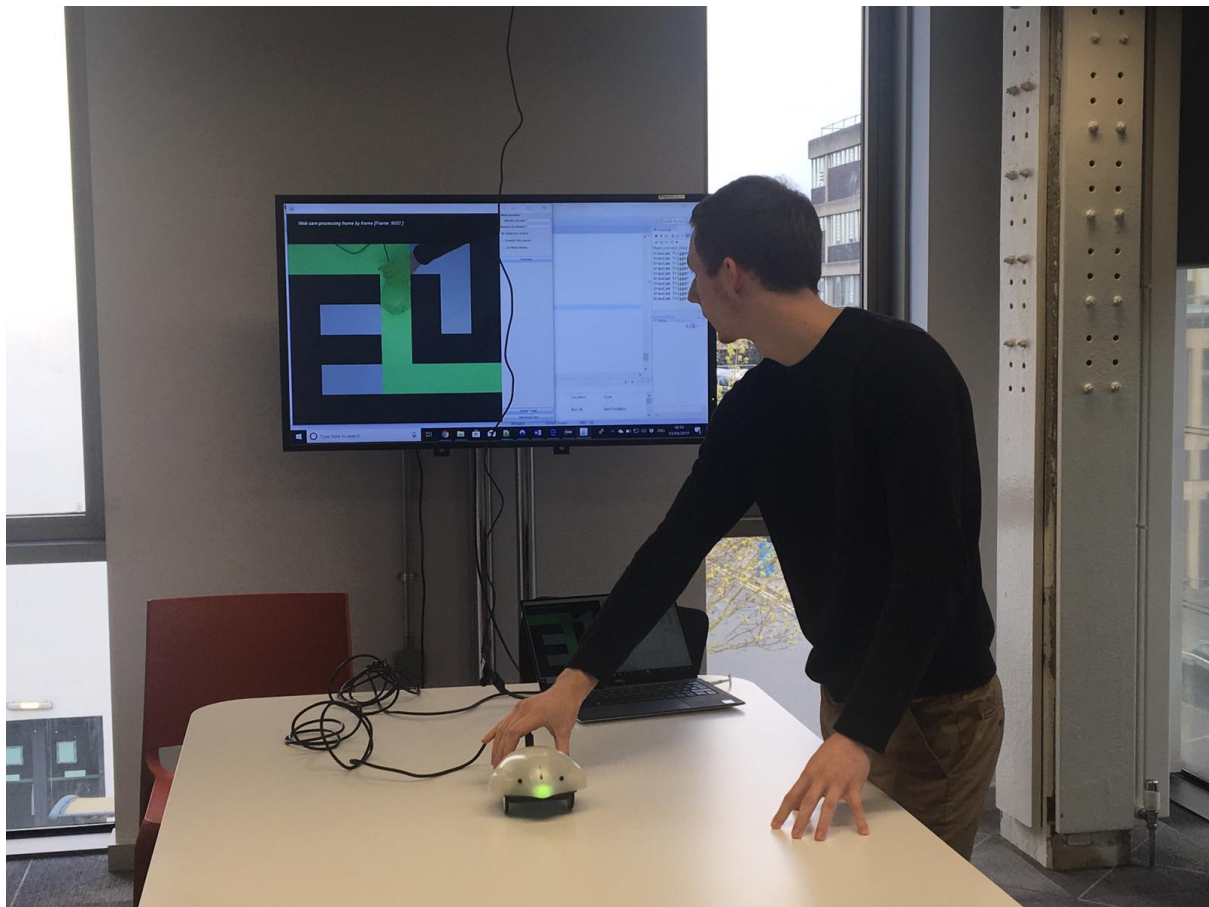


Figure 5.1 – Demonstration of software implemented

5.3 Implementing JFrame

This Sub-Section explains what was implemented in the frame class and how it differs from the prototype in Section 4.4.1.

5.3.1 JOptionBox and MazeVariable panel

One of the problems in the prototype was that the user had to wait for the default maze to load each time before being able to customise the maze, therefore a pop-up box was implemented using JOptionBox. In Figures 5.1 and 5.2 the panels share the same JTextFields and JLabels, taken from the prototype. Moreover, any integers entered in the initial box are automatically transferred over to the side panel to save the user time. Cellsize was removed due to the frame repainting in accordance to the width and height of the frame. JRadioButtons were grouped in a way to prevent the user from selecting more than one algorithm. The values are also checked so the user can't input invalid integers.

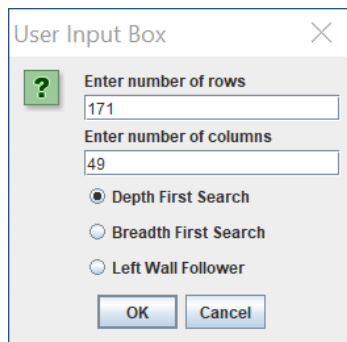


Figure 5.2 - Initial pop up box

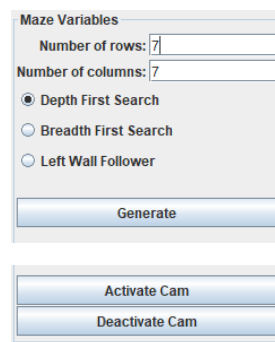


Figure 5.1 - Side panel

5.3.2 Frame Layout and setSize

The layouts implemented in the prototype were reused to great effect, this saved substantial development time. The error in Section 4.4.3 where a new frame was created on generation was fixed. As the maze was developed around a private globally accessible JFrame a public method could be called to resize the frame.

5.4 Implementing Kruskal's Algorithm

Due to the research in Section 2.5.2, it was decided that Kruskal's Algorithm should be implemented first, due to it producing a greater number of dead-ends meaning it was more challenging for the solving algorithms. As Jamis Buck said this algorithm did indeed prove hard to implement. The way the maze is displayed has been changed, because of the difficulties in displaying the Aldous-Broder algorithm in the prototyping phase. Firstly, the maze will be represented as a matrix of rows by columns, making it simpler to debug without the need for visualisation, therefore simplifying the display as explained later in Section 5.3.3. This type of

maze is called a block-based maze, rather than a maze that has thin walls between cells. A print method was created for the matrix, allowing the maze to be printed out in the console as shown in Figure 5.3. To see what the integers represent, refer to Figure 5.2.

```

MainGenerator [Java Application] C:\Program Files\Java\jdk1.8.0_191\bin\java.exe
java.awt.Dimension[width=1215,height=100]
0 0 0 0 0 0 0
1 1 1 1 0 1 0
0 1 0 0 0 1 0
0 1 1 1 1 1 0
0 0 0 1 0 0 0
0 1 1 1 1 1 1
0 0 0 0 0 0 0

```

Figure 5.3 - Maze printed in console

```

26 final static int wallCode = 0;
27 final static int emptyCode = 1;
28 final static int pathCode = 2;
29 final static int visitedCode = 3;

```

Figure 5.2 - Integer representations

5.4.1 Generate starting set of walls

This Sub-Section will go into detail on how Kruskal's Algorithm was implemented. Four methods were created that was inspired by "Chapter 10 – Kruskal's Algorithm" from Jamis Bucks' book (Buck, 2015). This particular implementation used is here is the randomised version as discussed in Section 2.5.2. The choice was made due to the maze being blocky and with no walls separating the cells. In Table 5.1 the matrix has been printed out showing the initial walls selected, the border around the maze has been taken out to simplify it. In Figure 5.4 the maze painted onto the JPanel is shown.

1	0	1	0	1
0	0	0	0	0
1	0	1	0	1
0	0	0	0	0
1	0	1	0	1

Table 5.1 - Wall selection printed

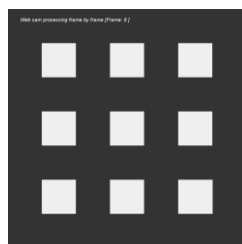


Figure 5.4 - Wall selection painted

This algorithm works by changing every even row and column as an empty cell (Green '1' in Figure 5.1), and every odd row and column as a wall cell (Red '0' in Figure 5.1). Furthermore, every empty cell needs to be uniquely identified allowing it to be picked at random. This was achieved by giving each cell a sequential number and multiplying it by -1 to avoid using the same numbers as in Figure 5.2, shown in the first cell of Table 5.2.

5.4.2 Selecting edges to merge

All the initial walls were uniquely identified and stored. The next step shown in the pseudocode of Section 2.5.2, works by putting every edge in a set and pulling one out at random. When an edge is chosen the algorithm determines whether the wall is above or to the right of an empty cell. Once this is determined it checks that if there will be a loop if the wall is taken down, if not, then the cell is changed to an empty cell and all empty cells connected will be changed to the lowest number. This is repeated until the set is empty, and a maze is created. The shortened process is shown in Table 5.2, and the way it is painted in Table 5.3.

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

Table 5.2 - Algorithm printed in console

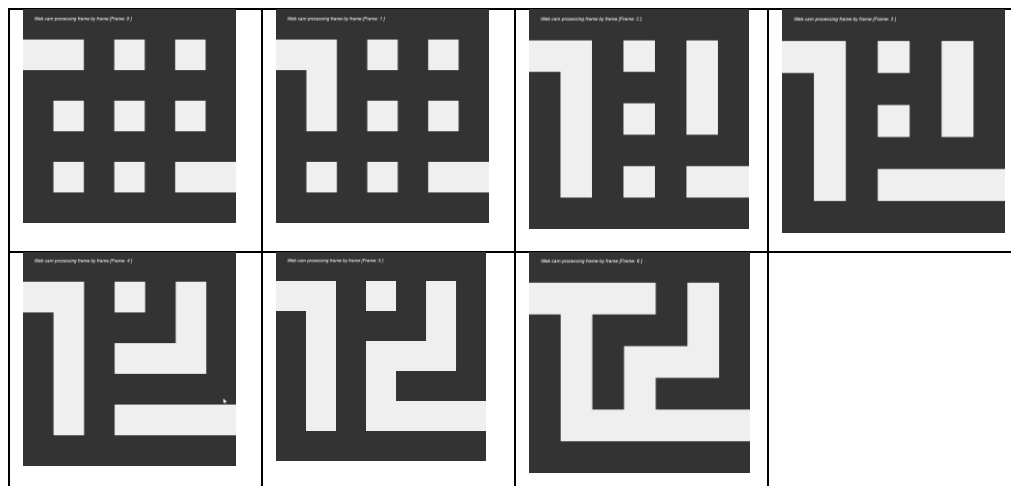


Table 5.3 - Kruskal's Algorithm painted

An animated Gif has been created to watch the generation happen for a 101 by 101 maze, <https://gyazo.com/19e496331276a36f37ec4e17e8f0f242>.

5.4.3 How maze is displayed

When the generation algorithm is created, the user needs a way to observe the maze to make sure the robot is following along correctly. To allow the user to see the difference between the choices of algorithms the maze generation needs to be visible. Previously in the prototyping phase this had been hard, however, with the matrix and block-based maze this was much easier. An extremely simple method was created that painted the maze onto the panel snippet (5.8). This method works by painting every cell in the matrix as a square rectangle that is coloured

depending on the integer of that cell. The colour depending on the integer can be found in Figure 5.7. The colours were created with transparency to allow the video image to be displayed through the maze.

```
void redrawMaze(Graphics g) {
    if (mazeExists = true) {
        int w = (int) Math.round(getWidth() / (double) columns);
        int h = (int) Math.round(getHeight() / (double) rows);
        for (int j=0; j<columns; j++)
            for (int i=0; i<rows; i++) {
                if (maze[i][j] < 0)
                    g.setColor(colour[emptyCode]);
                else
                    g.setColor(colour[maze[i][j]]);
                g.fillRect( (j * w), (i * h), w, h);
            }
    }
}
```

Figure 5.6 - Method to draw maze

```
// r g b a
public MazeArea() {
    colour = new Color[] {
        new Color(0,0,0,200), //Black // Maze walls
        new Color(255,255,255,20), //White // Maze paths
        new Color(0,255,0,80), //Green // Solving path
        new Color(255,0,0,80), //Red // Visited cells
    };
}
```

Figure 5.5 - Colour representation

The below Table 5.4 shows the result of painting a matrix.

0	0	0	0	0	0	0
1	1	0	1	1	1	0
0	1	0	1	0	0	0
0	1	1	1	0	1	0
0	1	0	0	0	1	0
0	1	1	1	1	1	1
0	0	0	0	0	0	0




Table 5.4 - Matrix painted in JFrame

This method searches through the current matrix and paints it on the canvas. Every time a new value is added to the matrix the method can be called, which makes it incredibly efficient in painting the generation happening. Then a method that delayed the progress of the algorithm allowed the generation to slowly appear.

5.5 Maze solving

This Sub-Section will explain all the algorithms implemented for solving the maze that was generated.

5.5.1 Recursive Backtracker

With the matrix and visualisation set up in Section 3.1.1, the only changes made were to add extra integer values and colours to the matrix shown in Figure (5. Error! Reference source not found.2) to indicate visited and solved cells.

The steps for this implemented algorithm can be found in Figure 5.7, the code for this method can be seen in the Appendix - B.5.

1. Check to see if current cell is unvisited
2. If at end of maze; stop
3. Check all 4 neighbouring cells to find any unvisited paths
4. If unvisited cell is found recall method on with new cell
5. If no unvisited direction is found return to previous cell

Figure 5.7 - Recursive Backtracker steps

The initial test can be seen in Table 5.5. The first column shows that algorithm (highlighted in green) has reached a dead-end. It recursively backtracks until another path is available, changing the path code (2) to visited code (3), this is displayed in column two. This is continued in columns three and four until it reaches the exit of the maze.

0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0
2 2 2 2 1 1 0	2 2 2 2 1 1 0	2 2 2 2 1 1 0	2 2 2 2 2 2 0
0 0 0 2 0 1 0	0 0 0 2 0 1 0	0 0 0 2 0 1 0	0 0 0 3 0 2 0
0 1 1 2 0 1 0	0 2 2 2 0 1 0	0 3 3 2 0 1 0	0 3 3 3 0 2 0
0 1 0 2 0 1 0	0 2 0 3 0 1 0	0 3 0 3 0 1 0	0 3 0 3 0 2 0
0 1 0 2 0 1 1	0 2 0 3 0 1 1	0 3 0 3 0 1 1	0 3 0 3 0 2 2
0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0

Table 5.5 - Recursive Backtracker printed

An animated Gif has been created to watch the algorithm for a 101 by 101 maze,
<http://recordit.co/Kf8Pr60lP1>.

5.5.2 Breadth First Search

As the implementation of Depth First Search (DFS) was straightforward, the decision was made to implement Bread First Search (BFS) next. Using the pseudocode from Section 2.4.2, the algorithm was implemented with one method. The steps for this algorithm can be found in Figure 5.8.

1. Add starting cell to queue
2. While queue is not empty
3. Check first cell in queue for unvisited neighbouring cells
4. If unvisited cell is found add to back of queue
5. Pop cell out of queue that has just been checked

Figure 5.8 - Breadth First Search steps

1. If at end of maze; stop
2. If left turn is available; turn left
3. Otherwise if able to move forward; move forward
4. Else turn around
5. Recall Method

Figure 5.9 - Wall follower steps

The reason why the algorithm will differ for a matrix maze is because directional sense is dependent on the current direction of the algorithm traversing the maze. Figure 5.9 shows the steps that the implemented algorithm will take, from column 2 to column 3 the algorithm fails both left and forwards checks, hence it turns around and then makes a left turn as it is available. However, step two had to be coded for all four directions, which increases the method length, this method is displayed in the Appendix (B.6). Table 5.8 shows how the algorithm would traverse a maze.

0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0
2 2 0 1 1 1 0	2 2 0 2 1 1 0	2 2 0 2 2 0 0	2 2 0 2 2 2 0
0 2 0 1 0 0 0	0 2 0 2 0 0 0	0 2 0 2 0 0 0	0 2 0 2 0 0 0
0 2 2 0 1 0	0 2 2 2 0 1 0	0 2 2 2 0 1 0	0 2 2 2 0 1 0
0 0 0 1 0 1 0	0 0 0 1 0 1 0	0 0 0 1 0 1 0	0 0 0 1 0 1 0
0 1 1 1 1 1 1	0 1 1 1 1 1 1	0 1 1 1 1 1 1	0 1 1 1 1 1 1
0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0

Table 5.8 - Follow wall algorithm printed

5.5.4 Maze display

Figure 5.10 through 5.15 show how the different algorithms are displayed in the software, green shows the final path found, while red shows the cells visited. Depth first solves by travelling through the maze and backtracking each time it reaches a dead end. Breadth first will expand through the maze, searching all paths then retrace the correct path. Wall follower was altered to show the user the current cell being checked.

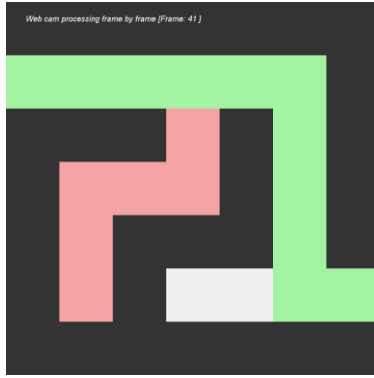


Figure 5.10 - Depth first

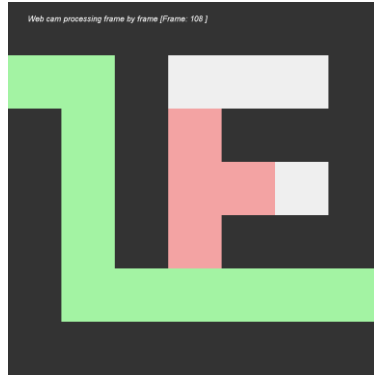


Figure 5.11 - Breadth first

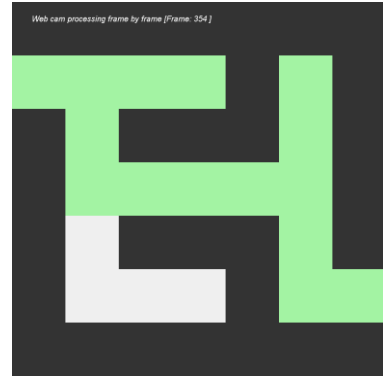


Figure 5.12 - Wall follower

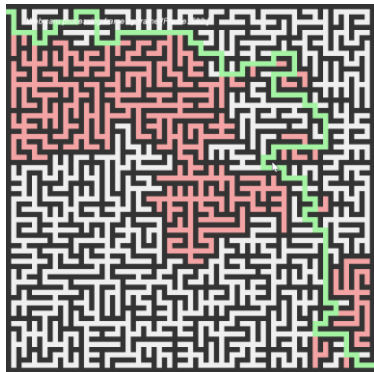


Figure 5.13 - Depth first

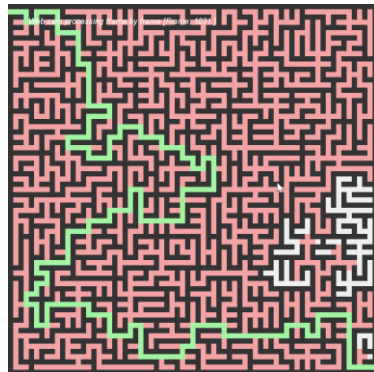


Figure 5.14 - Breadth first

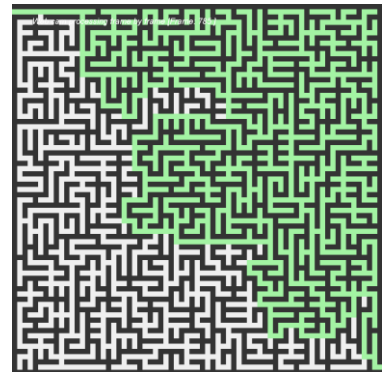


Figure 5.15 - Wall follower

5.6 Print times

Having implemented all the algorithms the user needed a way of comparing the algorithms to effectively choose which one to pick for the robot to use. To do this each of the implemented algorithms are paused for a few milliseconds to allow the user to see the generation. This will provide inaccurate results on how long the algorithms took to complete.

For this reason, the algorithm code was duplicated into a new Print class and the sleep thread methods were taken out. Although it is bad practice to have duplicate code in the project, because if changes are made to one algorithm it will have to be updated in this class, (Refactoring.guru, 2019), it was necessary to provide accurate results. The algorithms are tested for, “Time (MS)”, “Visited”, “Path” and “Coverage”, as seen in Figure 5.16. The ‘Visited’ column shows how many cells the algorithm has visited, not including the path found. The ‘Path’ column counts the number of cells in the final path found, both DFS and BFS should find the shortest path, whilst the wall follower doesn’t. Coverage is the percentage of the maze covered in visited or path cells, $(Visited + Path / Total) * 100$.

Name	Visited	Path	Coverage	Time (MS)
Depth First	92	55	73%	0.080193
Wall Follower	0	163	81%	0.319743
Breadth First	131	55	92%	2.376484
N/A	-	-	-	-

Figure 5.16 - Console output

5.7 Camera Implementation

As shown in the Section 2.6, for video capture and visual tracking openCV would be used. Version 2.4.6 had all the necessary features with the most documentation. A framerate counter was added in the top left as implementing the capture method in a different class decreased the frames per second (fps) of the video. However, moving it into the paint class increased the fps providing for a smoother experience. The lower framerates would impact the ability to track the robot in real time. In Figure 5.17 the image is shown behind the maze, the camera was clipped onto the ceiling to provide an overhead view of the maze.

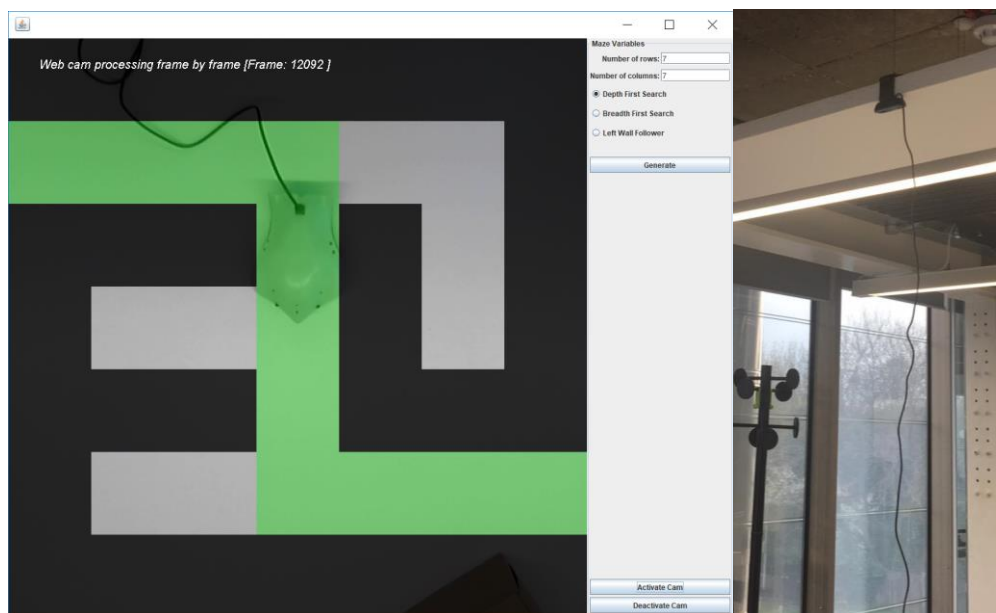


Figure 5.17 - Overhead camera implemented

5.8 Robot Navigation

The next phase would be to apply the path found into directions the robot could understand. An already created algorithm will not provide this, a new one had to be created. Copying the shortest path found from a Depth First Search allowed a simplistic path with no dead-ends. DFS was used as it was the quickest and worked on all perfect mazes. The steps of the algorithm implemented are shown in Figure 5.18.

1. If at end of maze; stop
2. If direction of next cell is the same as previous; add "forward" to queue
3. Otherwise if the next cell is on the left; add "turn left" to queue
4. Else add "turn right" to the queue
5. Recall method

Figure 5.18 - Robot navigation algorithm

The if statements in Figure 5.18 were created for all directions, the first direction is shown in Figures 5.19 and 5.20, these are replicated for the other directions.

```

65 static void move2(int row, int col, int[][] maze)
66 {
67     if(col==columns-1 && row==rows-2) {
68     }
69     else {
70         if(maze[row][col+1]==2){
71             if(currentDirection!=Direction.Right)
72                 changeDirectionRight();
73             q.add("Forward");
74             maze[row][col]=3;
75             move2(row,col+1,maze);
76         }

```

Figure 5.19 - Direction check

```

101 private static void changeDirectionRight() {
102     if(currentDirection==Direction.Up) {
103         q.add("Turn Right");
104     }
105     if(currentDirection==Direction.Down) {
106         q.add("Turn Left");
107     }
108     setCurrentDirection(Direction.Right);
109 }

```

Figure 5.20 - Directional change

The maze in Figure 5.21 was randomly generated and output from console is displayed in Table 5.9. This algorithm presumes that the robot will start at the middle of cell [0,1].

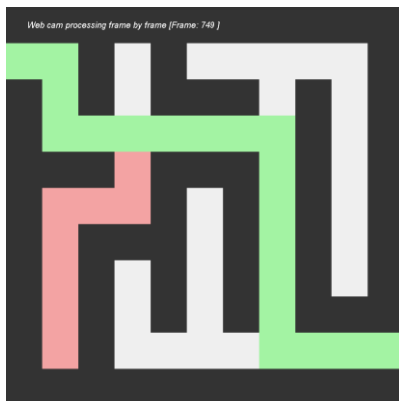


Figure 5.21 - Maze generated

```

Forwardx1
Turn Right
Forwardx2
Turn Left
Forwardx6
Turn Right
Forwardx6
Turn Left
Forwardx3

```

The robot was not able to be implemented in time due to the time restraints. The path outputted will work for a robot to follow, this does complete the objective of providing a faster solution to solve a maze.

5.9 Testing

Test #	Button tested	Test data	Algorithm selected	Expected result	Result
1	Generate button	7 / 7	DFS / BFS / Wall follower	Maze is produced and solved	Pass / Pass / Pass
2	Generate button	21 / 21	DFS / BFS / Wall follower	Maze is produced and solved	Pass / Pass / Pass
3	Generate button	141 / 141	DFS / BFS / Wall follower	Maze is produced and solved	Pass / Pass / Fail
4	Generate button	31 / 51	DFS / BFS / Wall follower	Maze is produced and solved	Pass / Pass / Pass
5	Generate button	100 / 100	DFS / BFS / Wall follower	Console rejects even input	Pass / Pass / Pass
6	Generate button	@ / 100	DFS / BFS / Wall follower	Console rejects invalid input	Fail / Fail / Fail
7	Activate Cam	Pressed	DFS / BFS / Wall follower	Camera to be displayed	Pass / Pass / Pass
8	Deactivate Cam	Pressed	DFS / BFS / Wall follower	Camera to stop being displayed	Pass / Pass / Pass
9	Exit	Exit pressed while camera is open	DFS / BFS / Wall follower	Program closes	Fail / Fail / Fail

Table 5.9 - Test cases

Test cases were carried out at the end of implementation as shown in Table 5.10, these tested the functionality of the program. Testing the Wall Following algorithm for a 141 by 141 maze revealed that the algorithm error due to not enough memory. These large mazes are not necessary to the project and due to time limitations, the algorithm will be kept. The user input is checked for invalid integers that the user might try to enter, as shown in test 6 it is not checked for non-integer values, this should be only inputted by accident. The final test 9, revealed that closing while the camera is still active will close the program but still run in the background. This needed to be fixed, as the only way of closing the program would be to close it from the debug menu. Another two methods were implemented that prevented the program from closing while the camera is active, this is turned off when the camera is de-activated. Two more tests were run to check the error had been fixed.

Exit	Exit pressed while camera is open	DFS / BFS / Wall follower	Program does nothing	Pass / Pass / Pass
Exit	De-active is pressed then exit	DFS / BFS / Wall follower	Program closes	Pass / Pass / Pass

Table 5.10 - Second test case

Chapter 6: Evaluation

6.1 Introduction

This chapter will evaluate each part of the project and then test the project overall. To time the algorithms, the methods to pause the algorithms have been removed, the number of rows and columns are changed via a for loop as shown in Figure 6.1. The data gathered was exported into notepad, separated with commas, then transferred into excel and put into understandable graphs.

```
123 PrintStream out = new PrintStream(new FileOutputStream
124     ("C:\\Users\\Seb\\Desktop\\Kruskals_1-300.txt"));
125 for(int i = 5; i<=300; i=i+2) {
126     for(int j = 5; j<=300; j=j+2) {
127         double begTime = System.nanoTime();
128         makeMaze();
129         generateWalls();
130         double endTime = System.nanoTime();
131         double duration = (endTime - begTime);
132         out.println();
133         out.print(duration/1000000+", "+i+", "+j+", ");
134
135         clear();
136         rows = i;
137         columns = j;
```

Figure 6.1 - Testing Kruskal's algorithm code

6.2 Testing Kruskal's Algorithm

The first test conducted was to the time taken for Kruskal's Algorithm to generate for square maze generation. Previously the only way of telling if the algorithm had generated was to look at the Java panel. After entering greater than 600 by 600 the Java panel would only display black due there not being enough space on screen. Deciding to test from 5 by 5 to 1000 by 1000, 5 is the smallest the maze will go with there being greater than one branch created. The system will print out 0 if it fails to generate the maze. Looking at the results below Figure 6.1, all the tests succeeded in creating the maze. The graph shows that the algoirthm will work very well for a low number of rows and columns, however, the graph takes on an exponential look. Meaning the algorithm will take an increasing amount of time extra, the greater the size of the maze. At 1000 by 1000 the algorithm was taking around 64 seconds to generate. The deviation of the generation from a line of best fit is minimal, similar results should be expected each time. As the project focuses on smaller mazes this will be looked into in the next graph.

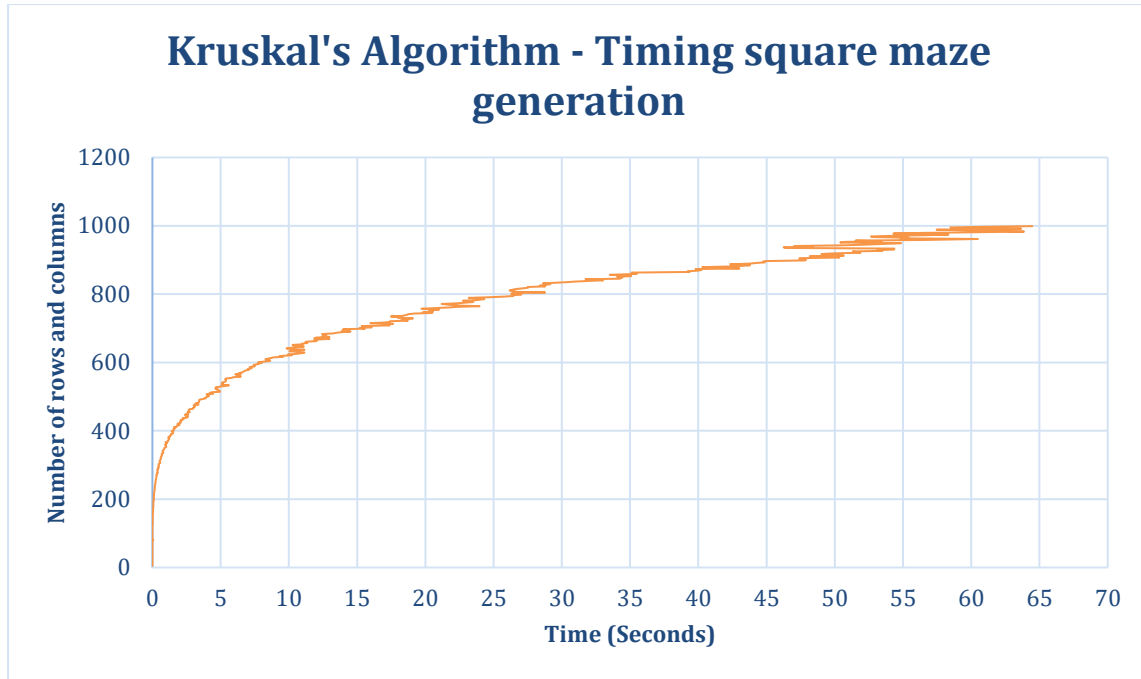


Figure 6.1 - Kruskal's Algorithm - Timed

The graph in Figure 6.2, uses the same data taken from the first test but plots the graph exponentially, this better shows the beginning of the graph which is of more interest. The first 15 mazes of low size generate in less than a millisecond, which is incredibly quick. As the graph is plotted exponentially the graph crosses the y-axis at $x = 1$. This means generating a maze below 350 by 350 takes less than a second, this is acceptable as the user would hardly notice and has no need to go greater than 350. There is one result that different from the plotted line, as there is only one this could be an anomaly.

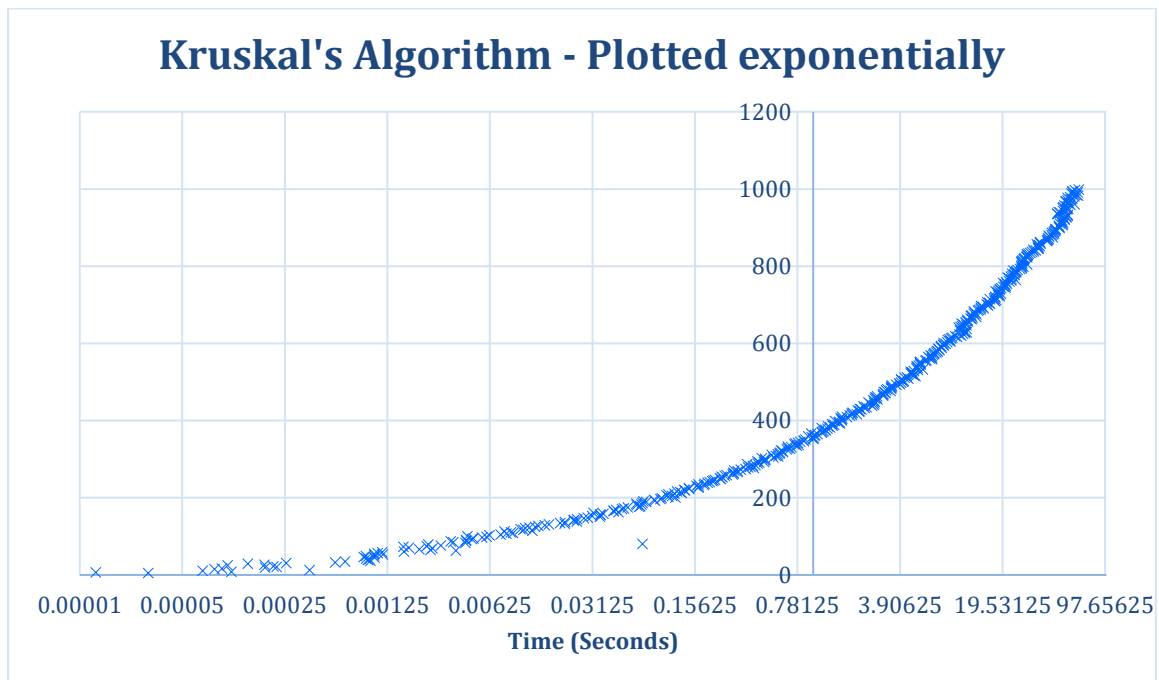


Figure 6.2 - Kruskal's Algorithm plotted exponentially

The next test was to see if there was a difference between generating a maze with low numbers of rows and high numbers of columns and visa versa. The graph can be viewed in the Appendix (7.4B.8). The number of columns is fixed going from 5 to 300 and each different line is a different number of rows, from 5 to 300. Due to the complexity of the graph a larger gap was left between lines to avoid the graph being solid colour. The furthestest line to the right would be 299 rows, which generates quickly when using a low number of columns, the same is true for the opposite side of the graph. This shows that it will take the same amount of time as there is no difference.

6.3 Testing Solving Algorithms

This sub-section will test all the different solving algorithms that were implemented, then evaluated on the results.

6.3.1 Testing square maze solving algorithms

The next algorithms to be tested were the solving algorithms, as the algorithms were created with no random the same results should be gathered each time, the only variable is the maze generation as it may generate a maze that is more lenient towards following the wall. This random chance was removed by testing the algorithms multiple times. All three algorithms were tested against each other from 7 to 200 with equal rows and columns.

After trying to test the algorithms in a for loop the follow the wall algorithm kept erroring, investigating the error showed that there was not enough memory for the algorithm to work as solving mazes with large numbers of rows and columns lead to the method being called many hundreds of times. Unable to fix this algorithm in time, the algorithm will produce 0 if it fails.

Referring to the Figure 6.3, the Depth first search algorithm performed the best out of all three algorithms, this was very noticeable. Breadth first algorithm was slower than the wall follower for low numbers of rows and columns, but the higher it goes the better it started performing. To make sure this was accurate a second test was conducted as shown in Figure 6.4, this still showed BFS being slower to begin with and then wall follower becoming slower for larger mazes. Being able to test these two algorithms against themselves for larger mazes would have been good to see if this trend continues. As BFS is meant to be slower than the Wall Following algorithm, there might be a shorter way to implement the algorithm which this project was unable to do in time.

The horizontal axis for the below charts was removed as it the numbers overlapped.

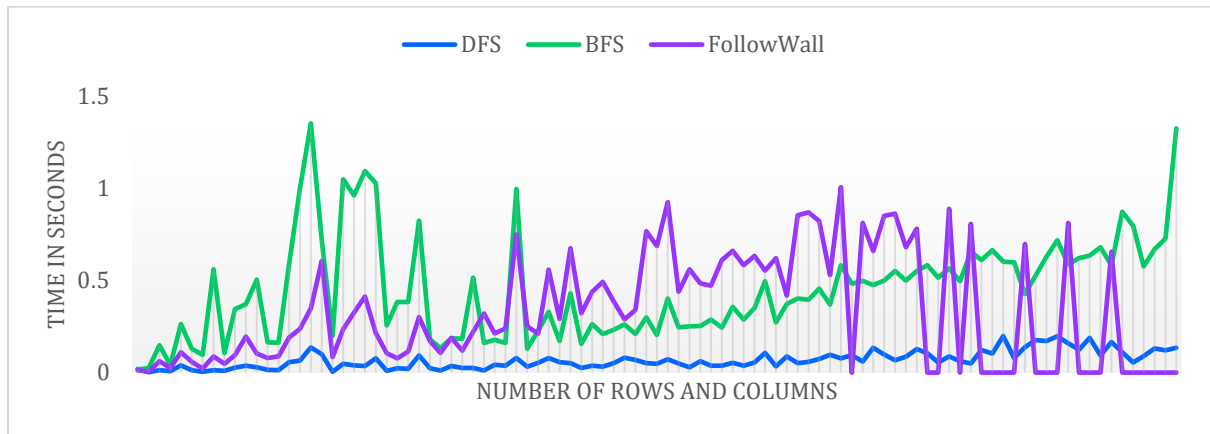


Figure 6.3 - Solving algorithms comparison 1

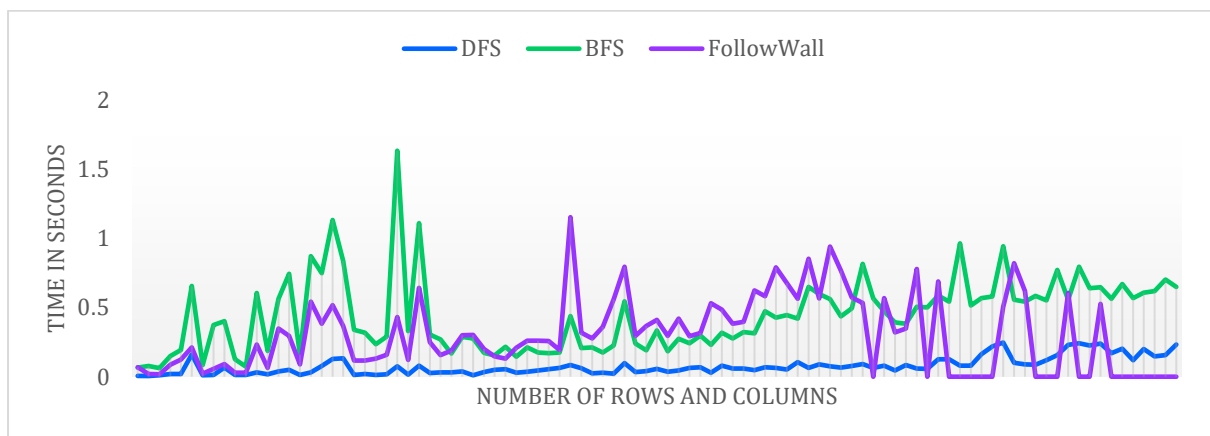


Figure 6.4 - Solving algorithms comparison 2

The Figure 6.5 shows all the algorithms tested to 1000 by 1000, the breadth first search took up to 100 seconds to complete, while the depth firsts search took a maximum of 19 seconds.

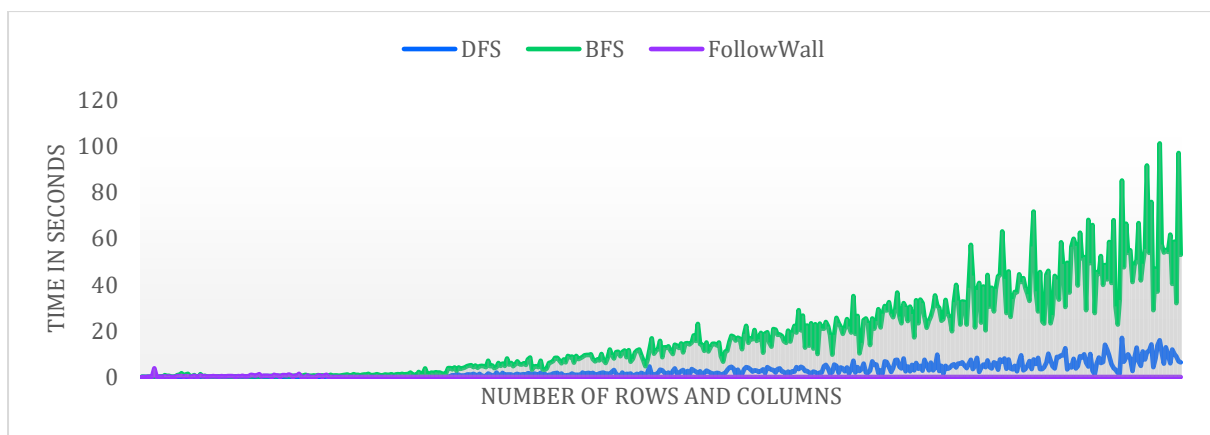


Figure 6.5 - Solving algorithms tested to 1000

6.3.2 Testing rectangular maze solving algorithms

The next tests were made on mazes that weren't square, as this should affect the algorithms. The tests go from 7 to 100 rows, by 7 to 100 columns, the size is measured by multiplying the rows and columns together. Looking at the first chart, DFS performed very well results shown in Figure 6.6. All results achieved below 0.4 seconds, this is expected as the size of the maze is small. Interestingly, the 4th longest time recorded had 7 columns by 57 rows this shouldn't be a difficult maze to solve, due to only time being recorded there isn't a way to tell why this maze took so long. However, for this maze specifically the other algorithms didn't show an increase in time taken.

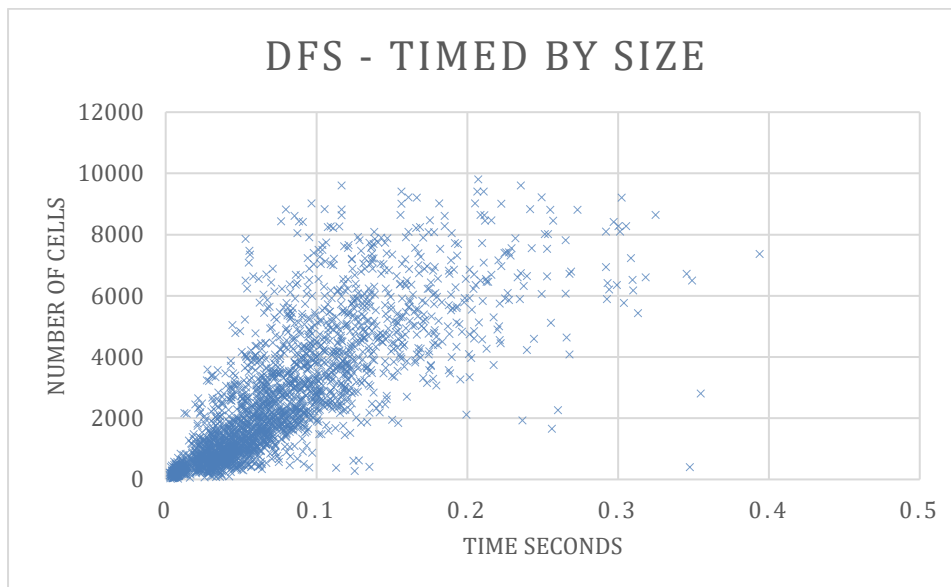


Figure 6.6 - DFS timed by size

The BFS search was also tested on the same mazes, this algorithm had major outliers that were taken out. The largest outlier took eight times longer to solve the maze than that of the largest maze, to see the unedited graph, see Appendix (B.7). The outliers were removed to make the graph easier to read as shown in Figure 6.7. This scatter graph shows little deviation from the large cluster. There is a noticeable line towards bottom of the graph, inspecting the points shows this is the smaller sized mazes, the algorithm solves smaller mazes quicker, for the larger mazes it is slower and follows a predictable path.

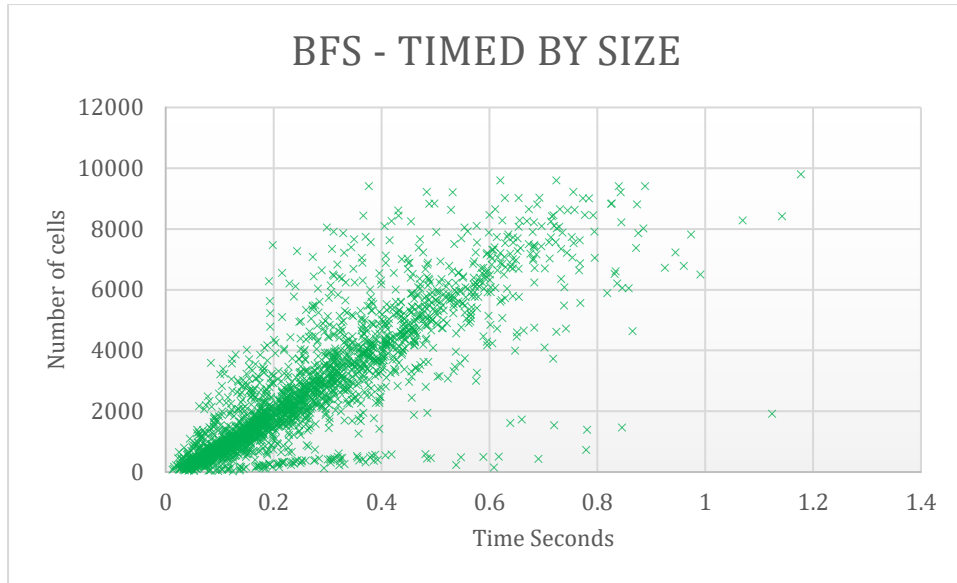


Figure 6.7 - BFS timed by size

The follow the wall algorithm shown in Figure 6.8, had the highest positive correlation with the most deviation coming from larger mazes. While earlier mazes being solved in similar times. This is to be expected as the larger the mazes goes the more chance it must be right turn dominate, which a a left wall follower will have difficulty solving quickly.

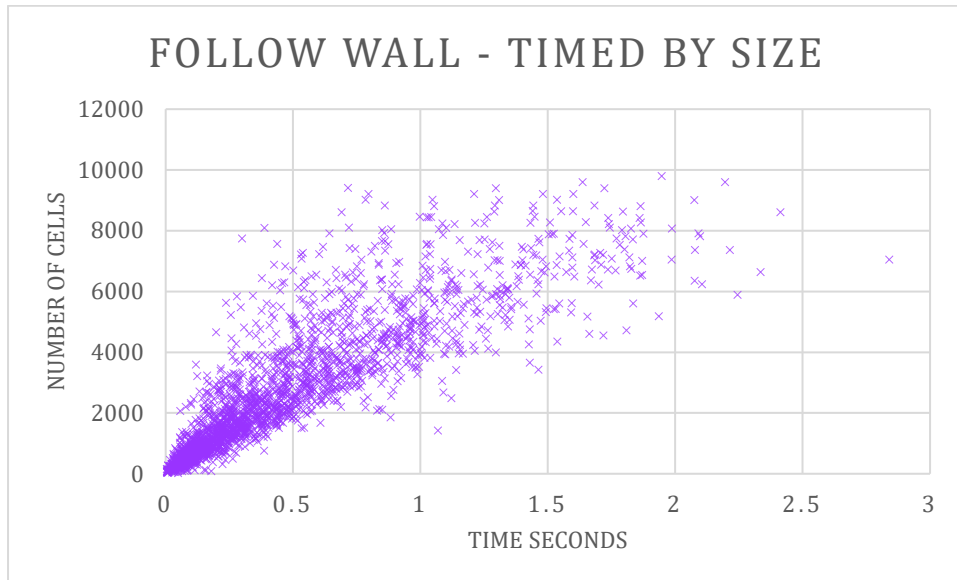


Figure 6.8 - Wall follower timed by size

6.4 Evaluating the size of the project

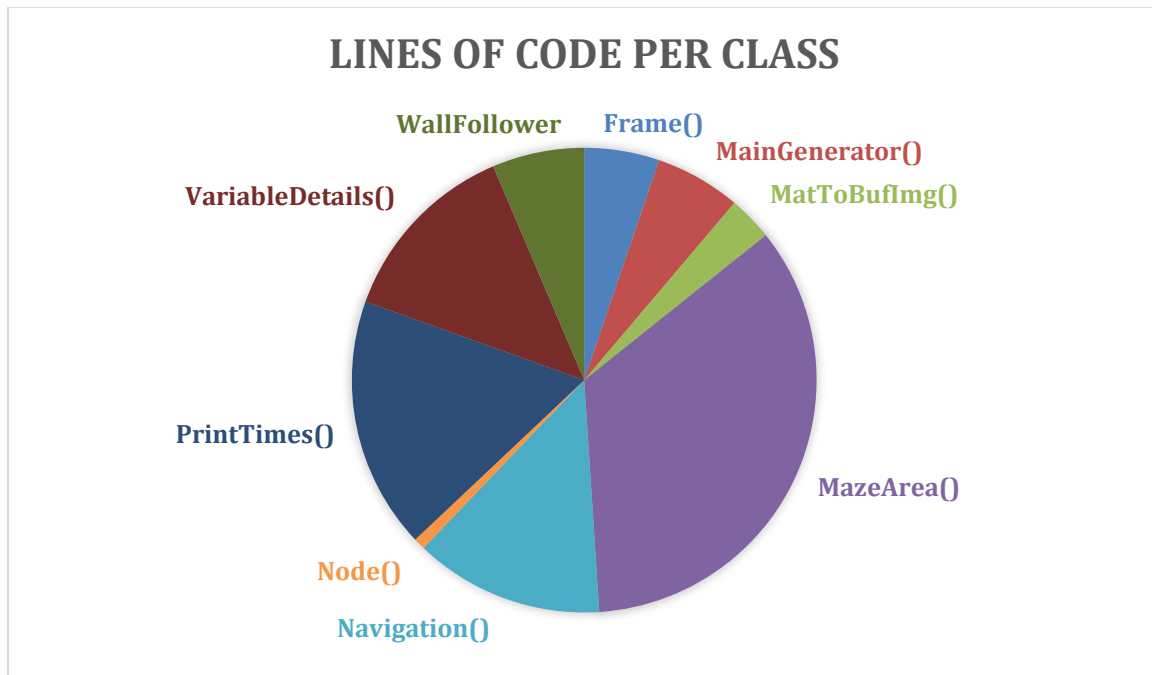


Figure 6.9 - Lines of code per class

In Figure 6.9 the lines of code per class is represented in a pie chart. As expected, the method that is used to paint the maze is the largest, this class can be split up into separate classes for each algorithm reducing the dependency on one class. However, due to time constraints this was not able to be completed. The PrintTimes() class was large due to having duplicate code of the algorithms from the MazeArea() class. VariableDetails() was a large class that held the layout for all the GUI elements,

Chapter 7: Conclusion

7.1 Introduction

This chapter will conclude the project by reflecting upon whether the project has met the objectives that were set out in (Section 1.2 - Aims and Objectives). This will also include a section going over the limitations and future work of the project.

7.2 Objectives of the project

Objective set in Section 1.2	Met?
I. To conduct a thorough literature review on maze generation and solving algorithms identifying which would be best for this project.	Yes, this was completed in Chapter 2.
II. Design an interface, which allows the user to generate their own sized maze.	Yes, this was developed in Chapters 4 and 5.
III. Randomly generate mazes, with the purpose of testing the speed of algorithms. The mazes will need several difficulties to thoroughly test the different algorithms and find out which is best. There should also be an easier difficulty to test the robot.	Yes, this was implemented in Section 5.4.
IV. Implement different algorithms that can solve the generated mazes. The metric for efficiency will be the route in and out of the maze with the least blocks used.	Yes, this was implemented in Section 5.5.
V. Image detection will need to be capable of tracking the robot as it progresses through the maze, and able to detect whether the robot has hit a wall or not.	No, only image capture was implemented in Section 5.7.
VI. Write up a detailed report on the project explaining the choices made throughout and any difficulties encountered.	Yes.
VII. To evaluate the project against the initial objectives to find out whether the overall aim is met.	Yes.

The overall aim had a solution that could work if implemented, however was not due to time constraints.

7.3 Limitations and reflection

The main limitation of this project was the time constraints. Visual tracking could have been implemented with a short extension in time. The robot wasn't implemented in time yet had the path it needed to follow. The DFS and Wall Following algorithms could have been implemented

to a better degree, which would have shortened the time taken to solve the mazes. As the error found with the Wall Following algorithm was discovered late, there wasn't enough time to fix it and complete the project.

7.4 Future Work

This project can continually be worked on with the implementation of more solving and generating algorithms. Moreover, in (Section 3.2- Research analysis and) all the algorithms that were meant to be implemented in this software were detailed, these algorithms would be great to be implemented in the future as they add extra complexity to the generation and solving. Similarly, the visual tracking that was meant to be implemented, would greatly benefit the project, as this allows the robot to find the beginning of the maze itself rather than the user having to position the robot, where there is room for human error. The robot implementation would have been good to compare different robots for their speed of solving the maze.

A feature outside of what was meant to be implemented would be using a projector. Mounting an overhead projector in the same position as the mounted camera, would greatly improve the visibility for the user, this would allow the user to see the maze projected onto the floor and then robot following the correct path, rather than looking at the computer screen. Unfortunately, this couldn't be implemented in this project as a separate image of the maze would have to be created and then presented to the project, this would be hard to implement as it might need to have separate projects running simultaneously.

References

- AlgoList.net. (n.d.). Depth-first search (DFS) for undirected graphs :: Graph theory | Algorithms and Data Structures. [online] Available at: http://www.algoList.net/Algorithms/Graph/Undirected/Depth-first_search [Accessed 1 Apr. 2019].
- Boofcv.org. (2019). BoofCV. [online] Available at: <https://boofcv.org/> [Accessed 1 Apr. 2019].
- Buck, J. (2010). Buckblog: Maze Generation: Recursive Backtracking. [online] Weblog.jamisbuck.org. Available at: <http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking> [Accessed 1 Apr. 2019].
- Buck, J. (2011). Maze Algorithms. [online] Jamisbuck.org. Available at: <https://www.jamisbuck.org/mazes/> [Accessed 1 Apr. 2019].
- Buck, J. and Carter, J. (2015). Mazes for programmers.
- Cylog.org. (2004). CyLog Software - Maze. [online] Available at: <https://www.cylog.org/sourcecode/maze.jsp> [Accessed 1 Apr. 2019].
- Dracopoulos, D. (1998). Robot Path Planning for Maze Navigation.
- Dreibholz, M. (n.d.). Maze generation algorithms (Prim, Kruskal, flood fill) - algostructure.com. [online] Algostructure.com. Available at: <http://www.algostructure.com/specials/maze.php> [Accessed 1 Apr. 2019].
- Feuerborn, T. (2017). Maze Generator. [online] Tenurian.com. Available at: <http://tenurian.com/maze.html> [Accessed 1 Apr. 2019].
- Garcia, B. (2013). Minotaur. [online] Ancient History Encyclopedia. Available at: <https://www.ancient.eu/Minotaur/> [Accessed 1 Apr. 2019].
- Ics.uci.edu. (2012). ICS 23 / CSE 23 Summer 2012, Project #1: Dark at the End of the Tunnel. [online] Available at: <https://www.ics.uci.edu/~thornton/ics23/LabManual/Dark/> [Accessed 1 Apr. 2019].
- Ioannidis, P. (2016). Procedural Maze Generation. Bachelor Thesis. National and Kapodistrian University of Athens.
- Jeulin-Lagarrigue, M. (n.d.). Kruskal's Maze Generator. [online] Hurna Wiki. Available at: https://wiki.hurna.io/algorithms/maze_generator/kruskal_s.html [Accessed 1 Apr. 2019].
- Kirupa (2006). kirupa.com - Depth First and Breadth First Search: Page 1. [online] Kirupa.com. Available at: https://www.kirupa.com/developer/actionsript/depth_breadth_search.htm [Accessed 1 Apr. 2019].
- Mazegenerator.net. (2019). Maze Generator. [online] Available at: <http://www.mazegenerator.net/> [Accessed 1 Apr. 2019].
- Micromouse Online. (2019). History - Micromouse Online. [online] Available at: <http://www.micromouseonline.com/micromouse-book/history/> [Accessed 1 Apr. 2019].

- Mishra, S. and Pande, P. (2008). Maze Solving Algorithms for Micro Mouse.
- National Building Museum. (2014). A Brief History of Mazes - National Building Museum. [online] Available at: <https://www.nbm.org/brief-history-mazes/> [Accessed 1 Apr. 2019].
- Opencv.org. (2019). OpenCV library. [online] Available at: <https://opencv.org/> [Accessed 1 Apr. 2019].
- Opendatastructures.org. (n.d.). 12.3 Graph Traversal. [online] Available at: http://opendatastructures.org/versions/edition-0.1e/ods-java/12_3_Graph_Traversal.html#SECTION00153200000000000000 [Accessed 1 Apr. 2019].
- Powell-Morse, A. (2016). V-Model: What Is It And How Do You Use It?. [online] Airbrake Blog. Available at: <https://airbrake.io/blog/sdlc/v-model> [Accessed 1 Apr. 2019].
- Pullen, W. (2015). Think Labyrinth: Maze Algorithms. [online] Astrolog.org. Available at: <http://www.astrolog.org/labyrnth/algrithm.htm#solve> [Accessed 1 Apr. 2019].
- Razimantv (2017). razimantv/mazegenerator. [online] GitHub. Available at: <https://github.com/razimantv/mazegenerator> [Accessed 1 Apr. 2019].
- Refactoring.guru. (2019). Duplicate Code. [online] Available at: <https://refactoring.guru/smells/duplicate-code> [Accessed 1 Apr. 2019].
- Reichert, A. (2019). armin-reichert/mazes. [online] GitHub. Available at: <https://github.com/armin-reichert/mazes/wiki> [Accessed 1 Apr. 2019].
- Schrijver, A. (2012). On the history of the Shortest Path Problem.
- Simplecv.org. (2019). SimpleCV. [online] Available at: <http://simplecv.org/> [Accessed 1 Apr. 2019].
- Software Integrity Blog. (2017). Top 4 Software Development Methodologies | Synopsys. [online] Available at: <https://www.synopsys.com/blogs/software-security/top-4-software-development-methodologies/> [Accessed 1 Apr. 2019].
- theJollySin (2018). theJollySin/mazelib. [online] GitHub. Available at: https://github.com/theJollySin/mazelib/blob/master/docs/MAZE_GEN_ALGOS.md [Accessed 1 Apr. 2019].
- Sebmins. (2019). Sebmins/MazeGenerator [online] GitHub. Available at: <https://github.com/sebmins/MazeGenerator> [Accessed 1 Apr. 2019]
- Williams, S. (2017). What the Waterfall Project Management Methodology Can (and Can't) Do for You. [online] Lucidchart.com. Available at: <https://www.lucidchart.com/blog/waterfall-project-management-methodology> [Accessed 1 Apr. 2019].
- 101 Computing. (2017). Backtracking Maze – Path Finder | 101 Computing. [online] Available at: <https://www.101computing.net/backtracking-maze-path-finder/> [Accessed 1 Apr. 2019].

Appendix A Personal Reflection

A.1 Reflection on Project

The project went well, I think it was a too ambitious project from the beginning and should have been scaled back, if there is time at the end extra could have been implemented. Implementing the visual tracking at an earlier stage rather than focusing on many different algorithms could have allowed the project to be fully completed.

If the time had been greater, I would have liked to experiment with different types of mazes in the prototyping phase. Such as generating the maze like a graph with nodes, this would have made it easier to implement certain algorithms greatly, however, even now I do not know if this would be a better choice over a matrix maze, as it would be more difficult to visualise and paint onto the Java Panel.

A.2 Personal Reflection

I started this project early in October and continued a steady pace of work, taking a break in December to work on other modules coursework. On the other hand, I started the write up of the dissertation very late, 3 weeks from the deadline. This was the biggest downfall as all the effort that went into implementation wasn't covered as well as hoped. As the project wasn't finished, I had to write while continually developing right up until the final week. Another area I could have improved on, was spending a lot of time implementing features that improved the users experience such as, if they inputted a number of rows and columns it saves and it displays it in the next box, this is to stop the user from having to retype the same number again and again. Another example would be the feature that didn't allow the user to close the program while the camera feed was being accessed, these features and many others were implemented to make the software more user friendly, however it took up a lot of development time that could have been used to fully develop the project, then go back and add the finishing touches.

Appendix B Appendices

B.1 Ethics Approval Confirmation

Message ×

Project Short Title	Robot navigating maze given predetermined route
Date	12/11/2018 09:12

Thank you for your application. On the basis of the information you have provided on the application form, your project does not require research ethics approval.

[View Form](#)[Close](#)

Figure 0.1 - Ethics approval

B.2 Literary review pseudocode and tables

The Algorithm:

1. Pick a random direction and follow it
2. Backtrack if and only if you hit a dead end.

Figure 0.2 - Recursive Backtracker pseudocode (theJollySin, 2018)

Click to return to (0 - There are many different solving algorithms which solve many different types of mazes. This section will only cover the algorithms that are most likely to be implemented. The literacy for this project will focus on the most popular, the fastest or whether it finds the shortest path.

Recursive Backtracker (DFS))

The Algorithm:

1. create a possible solution for each neighbor of the starting position
2. find the neighbors of the last element in each solution, branches create new solutions
3. repeat step 2 until you reach the end
4. The first solution to reach the end wins.

Results

- finds all solutions
- finds shortest solution(s)
- works against imperfect mazes

Figure 0.3 - Breadth First pseudocode (theJollySin, 2018)

[Click to return to \(2.4.2 - Breadth First Search\)](#)

The Algorithm

Follow the right wall and you will eventually end up at the end.

The details:

1. Choose a random starting direction.
2. At each intersection, take the rightmost turn. At dead-ends, turn around.
3. If you have gone more than $(H * W) + 2$ cells, stop; the maze will not be solved.
4. Prune the extraneous branches from the solution before returning it.

Optional Parameters

- *Turn*: String ['left', 'right']
- Do you want to follow the right wall or the left wall? (default 'right')

Figure 0.4 - Wall Following pseudocode (theJollySin, 2018)

[Click to return to \(2.4.3 - Wall Following\)](#)

Algorithm	Solution	Works 100%	Human Doable?	Passage Free?	Memory Free?	Fast?
<i>Random Mouse</i>	1	no	Inside / Above	no	Yes	no
<i>Wall Follower</i>	1	no	Inside / Above	Yes	Yes	Yes
<i>Pledge Algorithm</i>	1	no	Inside / Above	Yes	Yes	Yes

<i>Chain Algorithm</i>	1	Yes	no	Yes	no	Yes
<i>Recursive Backtracker</i>	1	Yes	no	Yes	no	Yes
<i>Trémaux's Algorithm</i>	1	Yes	Inside / Above	no	no	Yes
<i>Dead End Filler</i>	All +	no	Above	no	Yes	Yes
<i>Cul-de-sac Filler</i>	All +	no	Above	no	Yes	Yes
<i>Blind Alley Sealer</i>	All +	Yes	no	no	no	Yes
<i>Blind Alley Filler</i>	All	Yes	Above	no	Yes	no
<i>Collision Solver</i>	All Shortest	Yes	no	no	no	Yes
<i>Shortest Paths Finder</i>	All Shortest	Yes	no	Yes	no	Yes
<i>Shortest Path Finder</i>	1 Shortest	Yes	no	Yes	no	Yes

Figure 0.5 - Recursive Backtracker pseudocode (theJollySin, 2018)

Click to return to 2.4.2(2.4 - Maze Solving Algorithms)

The Algorithm

1. Randomly choose a starting cell.
2. Randomly choose a wall at the current cell and open a passage through it to any random, unvisited, adjacent cell. This is now the current cell.
3. If all adjacent cells have been visited, back up to the previous and repeat step 2.
4. Stop when the algorithm has backed all the way up to the starting cell.

Results

perfect

Figure 0.6 - Recursive Backtracker pseudocode (theJollySin, 2018)

Click to return to (2.4.22.5.1 - Recursive Backtracker (DFS)2.4)

Kruskal's Original Version	Maze Generator Version
<ul style="list-style-type: none"> • Give a unique subset id for each existing cell. • Throw all of the edges into a big set. • While there is edge to be handled in the set: • <u>1. Pull out the edge with the lowest weight.</u> • 2. If the edge connects two disjoint subsets: <ol style="list-style-type: none"> a. Connect cells. b. Join the subsets. 	<ul style="list-style-type: none"> • Give a unique subset id for each existing cell • Throw all of the edges into a big set. • While there is edge to be handled in the set: • <u>1. Pull out one edge at random.</u> • 2. If the edge connects two disjoint subsets: <ol style="list-style-type: none"> a. Connect cells. b. Join the subsets.

Figure 0.7 - Kruskal's Algorithm pseudocode (Jeulin Lagarrigue, n.d.)

[Click to return to \(2.5.2 - Kruskal's Algorithm2.4.2\)](#)

The Algorithm

1. Choose a random cell.
2. Choose a random neighbor of the current cell and visit it. If the neighbor has not yet been visited, add the traveled edge to the spanning tree.
3. Repeat step 2 until all cells have been visited.

Results

perfect, unbiased

Figure 0.8 - Aldous Broder pseudocode (theJollySing, 2018)

[Click to return to \(2.5.3 - Aldous-Broder Algorithm2.5.22.4.2\)](#)

The Algorithm

1. Put the each cell of the first row in its own set.
2. Join adjacent cells. But not if they are already in the same set. Merge the sets of these cells.
3. For each set in the row, create at least one vertical connection down to the next row.
4. Put any unconnected cells in the next row into their own set.
5. Repeat steps 2 through 4 until the last row.
6. In the last row, join all adjacent cells that do not share a set.

Optional Parameters

- *xskew*: Float [0.0, 1.0]
- Probability of joining cells in the same row. (default 0.5)
- *yskew*: Float [0.0, 1.0]
- Probability of joining cells in the same column. (default 0.5)

Results

perfect

Figure 0.9 - Eller's Algorithm pseudocode (theJollySin, 2018)

[Click to return to \(2.5.4 - Eller's \)](#)

Algorithm	Dead End %	Type	Bias Free?	Memory	Time	Solution %
<i>Uncursal</i>	0	Tree	Yes	N^2	379	100.0
<i>Recursive</i>	10	Tree	Yes	N^2	27	19.0
<i>Backtracker</i>						
<i>Hunt and Kill</i>	11 (21)	Tree	Yes	0	100 (143)	9.5 (3.9)

<i>Recursive Division</i>	23	Tree	Yes	N^*	10	7.2
Binary Tree	25	Set	no	\emptyset^*	10	2.0
<i>Sidewinder</i>	27	Set	no	\emptyset^*	12	2.6
Eller's Algorithm	28	Set	no	N^*	20	4.2 (3.2)
<i>Wilson's Algorithm</i>	29	Tree	Yes	N^2	48 (25)	4.5
Aldous-Broder Algorithm	29	Tree	Yes	\emptyset	279 (208)	4.5
<i>Kruskal's Algorithm</i>	30	Set	Yes	N^2	33	4.1
Prim's Algorithm (true)	30	Tree	Yes	N^2	160	4.1
<i>Prim's Algorithm (simplified)</i>	32	Tree	Yes	N^2	59	2.3
Prim's Algorithm (modified)	36 (31)	Tree	Yes	N^2	30	2.3
<i>Growing Tree</i>	49 (39)	Tree	Yes	N^2	48	11.0
Growing Forest	49 (39)	Both	Yes	N^2	76	11.0

Figure 0.10 - Table comparing generation algorithms (Pullen, 2015)

[Click to return to \(2.5 - Maze Generation Algorithms2.5.4\)](#)

B.3 Flow diagram

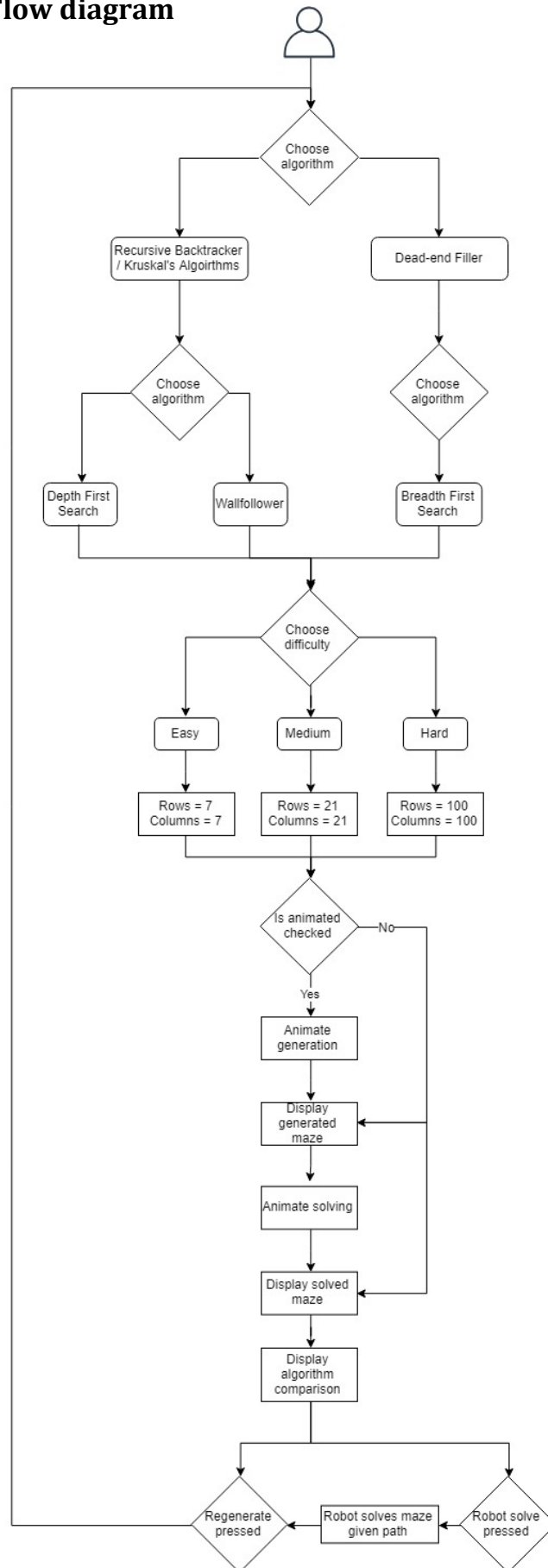


Figure 0.11 - Flow diagram

B.4 Design mock up

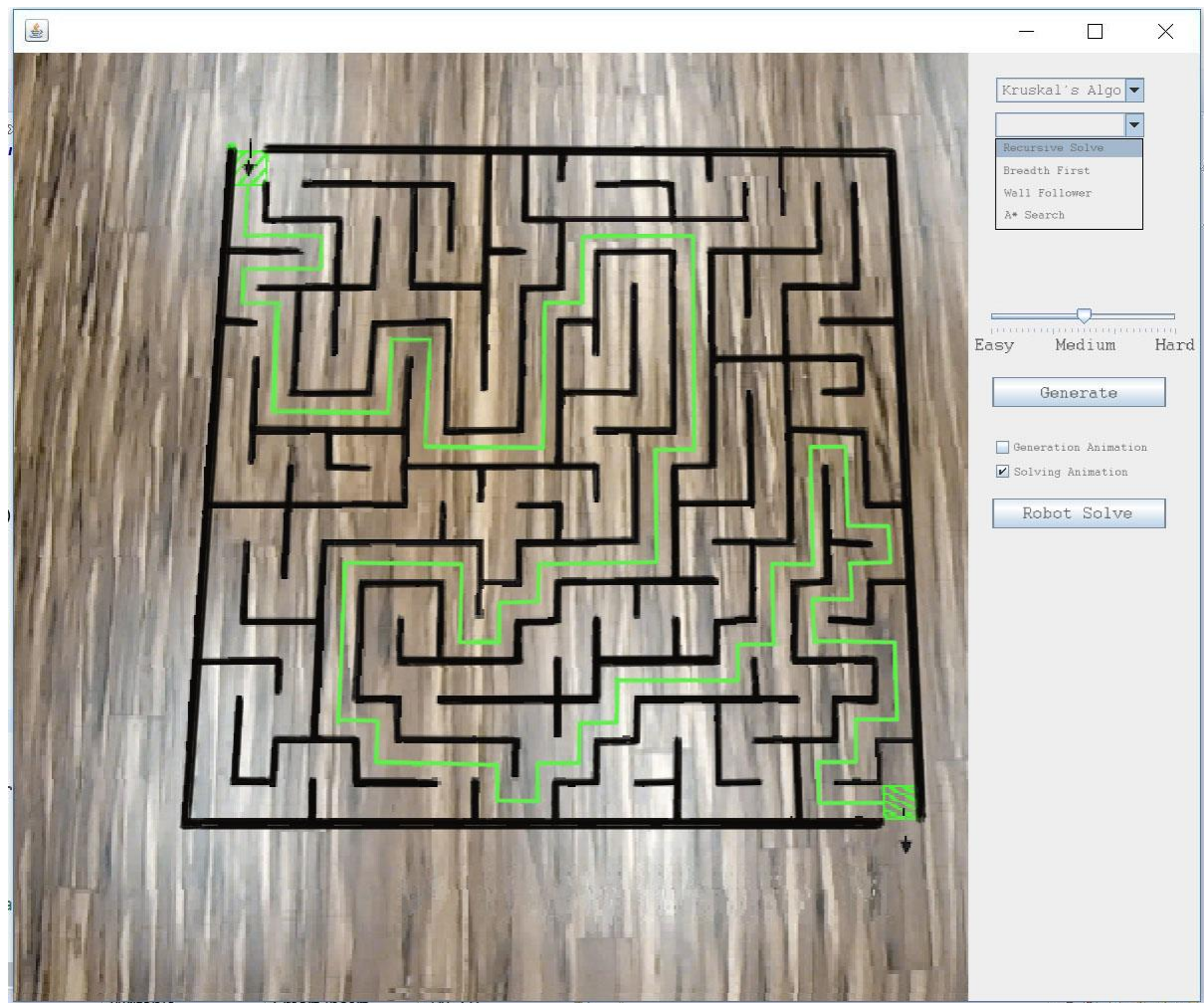


Figure 0.12 - Design mock up

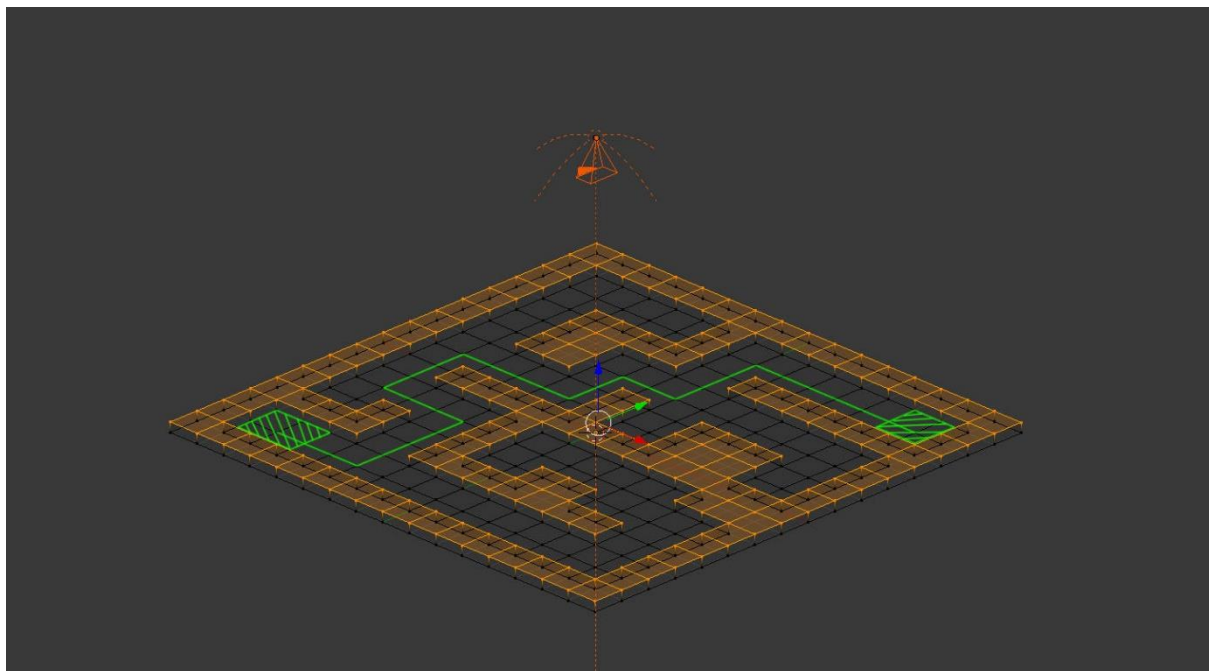


Figure 0.13 - Camera assisting robot

B.5 Depth First Search Algorithm code

```

225=boolean recursiveSolve(int row, int col, int rowEnd, int colEnd) {
226     if (maze[row][col] == emptyCode) {
227         maze[row][col] = pathCode;
228
229         if (row == rowEnd && col == colEnd)
230             return true;
231
232         if (recursiveSolve(row+1,col,rowEnd, colEnd) == true ||
233             recursiveSolve(row,col+1,rowEnd, colEnd) == true ||
234             recursiveSolve(row-1,col,rowEnd, colEnd) == true ||
235             recursiveSolve(row,col-1,rowEnd, colEnd) == true )
236             return true;
237
238         maze[row][col] = visitedCode;
239     }
240     return false;
241 }

```

Figure 0.14 - Recursive solve code

B.6 Wall Following Algorithm code

```

357=boolean move(int row, int col, int[][] maze) {
358     maze[row][col]=3;
359     if(row==rows-2 && col==columns-1)
360     {
361         maze[row][col]=2;
362         repaint();
363         return true;
364     }else {
365         if(wf.getCurrentDirection() == Direction.Right){
366             if(maze[row-1][col]==1 || maze[row-1][col]==2) {
367                 wf.setCurrentDirection(Direction.Up);
368                 maze[row][col]=2;
369                 move(row-1,col,maze);
370             }else if (maze[row][col+1]==1 || maze[row][col+1]==2){
371                 maze[row][col]=2;
372                 move(row,col+1,maze);
373             }else {
374                 wf.setCurrentDirection(Direction.Left);
375                 move(row,col,maze);
376             }
377         }

```

0.1 - Wall Follower method

B.7 BFS chart with outliers

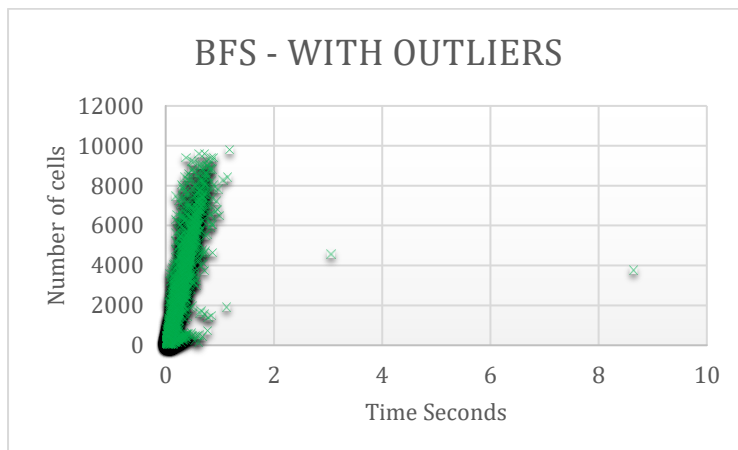


Figure 0.15 - BFS with outliers

B.8 Kruskal's Algorithm - Non square mazes

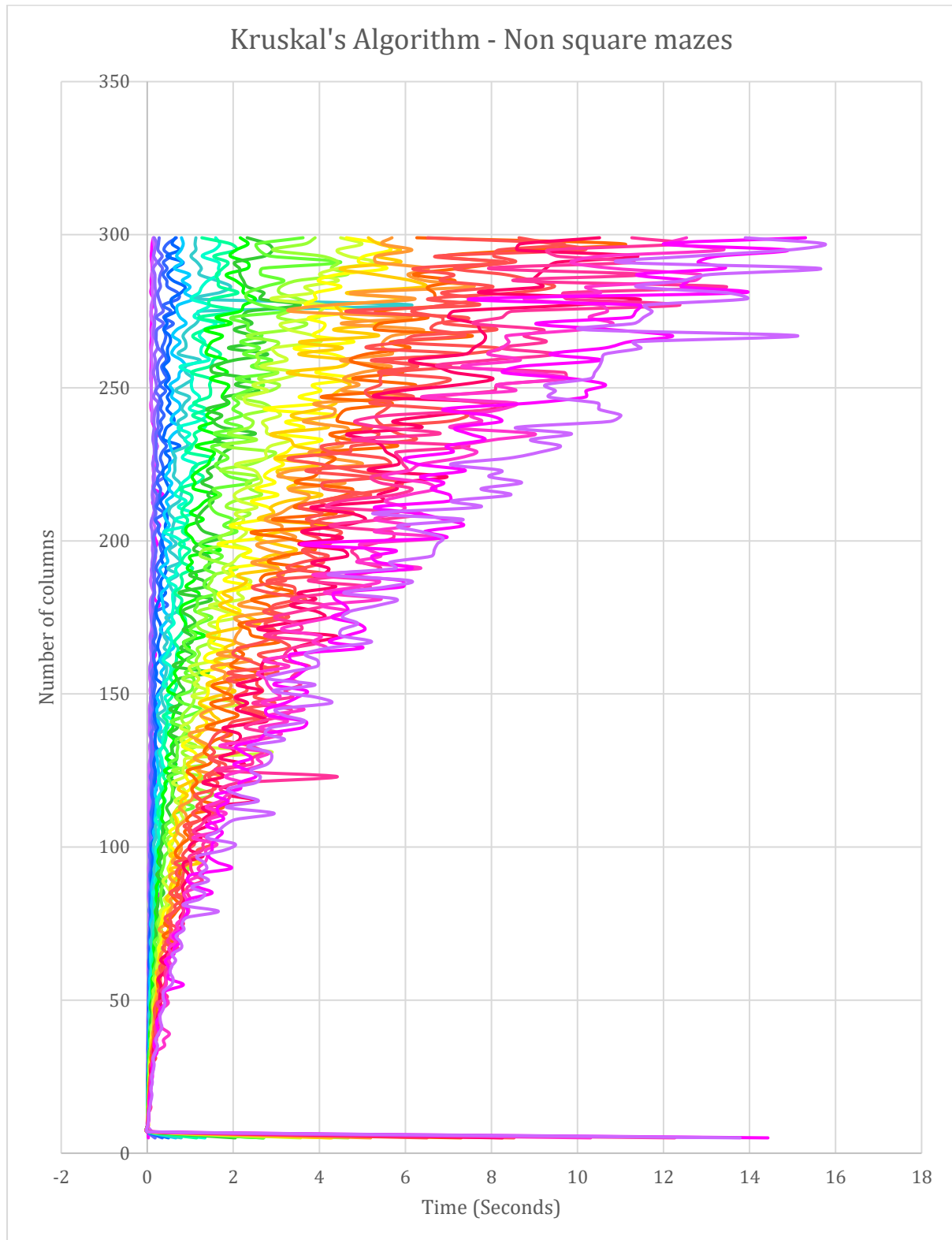


Figure 0.16 - Kruskal's algorithm plotted