



School of Computer Science and Mathematics

6100COMP Project

Final Report Submitted by

Keith Taylor

884378

Computer Science

Title

Solving Procedurally Generated Mazes

Supervised by

Dr Reino Niskanen

Submitted on

21 April 2023

Abstract

Pathfinding algorithms are present in everyday technology such as video games or the route planning in google maps. Despite their ubiquity, they commonly use a mathematical language that requires familiarity and experience to fully understand. Reading algorithms in their most abstract form requires experience and familiarity, even more so if your goal is to extrapolate from the abstract and begin to understand the behaviour of an algorithm.

The goal of this project is to develop software using the python language that acts as a steppingstone to a deeper understanding of pathfinding algorithms through visualisation techniques. By presenting algorithms not in a mathematical language but a visual one, the goal is to give students a new experience to associate with algorithm behaviours and offer new avenues of learning. Further, the software allows users to customise certain settings, giving a level of interactive experience with algorithms they may not have had access to which can allow them to answer questions regarding algorithmic behaviours.

The program implements the binary tree and Aldous-Broder methods of maze generation as well as an algorithm developed alongside the software to create randomised mazes. The program can then implement a range of 3 different pathfinders with 7 total variations, including Breadth-First Search, Depth-First Search and A*.

The program functions as intended, accurately simulating all maze generation and pathfinding algorithms. Through the use of colour and animation it attempts to communicate algorithm behaviour to the user.

This report also suggests several avenues for further work, such as the implementation of new algorithms, improvements upon the already modified binary tree algorithm and offering a greater range of options to the user.

Acknowledgement

I would like to express my deepest appreciation for my supervisor Dr Reino Niskanen, who provided the original project idea and offered assistance with patience and no judgement throughout the year-long project.

This project also would not have been possible without the teaching of Dr Somasundaram Ravindran which was instrumental in beginning this project.

I cannot begin to express my thanks to both Ellen and Emma for their endless emotional support during my final year.

I would also like to thank Saúl and Alex F, who offered their academic wisdom and their patience to answer my many inane questions.

A special thanks to Kate, who was instrumental in keeping me on task and accountable for hitting my deadlines.

I would also like to acknowledge the efforts of Alex B, Jad and Luka who were prepared to offer support whenever they could.

I also want to recognise John and Mandi, who provided me with a place to live when I really needed it.

Declaration

I confirm that the work contained in this BSc project report has been composed solely by myself and has not been accepted in any previous application for a degree. All sources of information have been specifically acknowledged and all verbatim extracts are distinguished by quotation marks.

Signed: Keith Taylor

Date: 21/04/2023

List of Figures

Figure 1 - Depth-first search pseudocode from Brilliant.org [3]	10
Figure 2 - UML Class Diagram	17
Figure 3 - Pseudocode detailing maze data structure	18
Figure 4 - Design sketch for GUI	19
Figure 5 - Design of maze visualisation with key	20
Figure 6 - Design of colour accurate pathfinding visualisation with key	21
Figure 7 - Pseudocode for mazeDrawWhole()	22
Figure 8 - Pseudocode for mazeDrawCell()	22
Figure 9 - Pseudocode for neighbourCount()	23
Figure 10 - Pseudocode for modified binary tree generation	25
Figure 11 - Partial pseudocode for depth-first search	26
Figure 12 - Pseudocode for A* heuristic calculation pfHeuristic()	27
Figure 13 - Implemented code for mazeDrawWhole()	32
Figure 14 - Implemented code for mazeDrawCell()	32
Figure 15 - Implemented code for neighbourCount()	34
Figure 16 - Code snippet of first while loop in pfBF() as implemented	36
Figure 17 - Modified Binary Tree maze with Opposed Depth-First Search Solution	38
Figure 18 - Modified Binary Tree Maze with favourably biased Depth-First Search Solution	38
Figure 19 - Implemented code of pfHeuristic()	39
Figure 20 - Code snippet from implemented pfAStar()	40
Figure 21 - Instance of a maze with two simultaneously visualised solutions from both Breadth-First Search and A* Euclidean	42
Figure 22 – Snippet of code from mazeGen() function, demonstrating code that should be replaced with MATCH/CASE	45
Figure 23 – Modified binary tree maze solved with no yellow cells, i.e. wasted searched, by Depth-First Search (same bias), A* Euclidean and A* Manhattan respectively	46
Figure 24 - Binary Tree Maze used for testing in Table 10	55
Figure 25 - Modified Binary Tree Maze used for testing in Table 10	55
Figure 26 - Aldous-Broder maze used for testing in Table 10	56
Figure 27 - Modified Binary Tree Maze used for test 1 in Table 11	56

Figure 28 - Modified Binary Tree Maze used for test 2 in Table 11	57
Figure 29 - Modified Binary Tree Maze used for test 3 in Table 11	57
Figure 30 - Initial Project Timeline	58
Figure 31 - Initial Project Timeline represented as a Gantt chart	58
Figure 32 – Entire code of 'main.py'	69
Figure 33 – Entire code of 'mazeGen.py'	76
Figure 34 – Entire code of 'pathFind.py'	95
Figure 35 - November 2022 Monthly Meeting Report	96
Figure 36 - December 2022 Monthly Meeting Report	97
Figure 37 - January 2023 Monthly Meeting Report.....	98
Figure 38 - March 2023 Monthly Meeting Report.....	99

List of Tables

Table 1 - Main class and functions.....	31
Table 2 - BinaryTree class and function	33
Table 3 - BinaryTreeModified class and function	33
Table 4 - AldousBroder class and function	33
Table 5 - Tools class and functions	33
Table 6 - BreadthFirst class and function.....	35
Table 7 - DepthFirst class and functions	37
Table 8 - AStarAlgorithm class and functions	39
Table 9 - Explanation of visited dictionary values in pfAStar()	40
Table 10 - Results of testing for solution time in Depth-First Search against Breadth-First Search.....	47
Table 11 - Results of testing for solution lengths in Depth-First Search against Breadth-First Search, in imperfect mazes.....	48
Table 12 - Demonstration of code lines per algorithm variation	48

List of Contents

Abstract.....	2
Acknowledgement	3
Declaration.....	4
List of Figures	5
List of Tables	7
List of Contents	8
1 Introduction	10
1.1 Background Information	10
1.2 Problem Statement	11
1.3 Project Scope.....	11
1.4 Project Aims and Objectives	11
2 Research.....	12
2.1 Topic Research	12
2.2 Literature Review	14
3 Design.....	15
3.1 Overview of the design	15
3.2 Requirement of the Analysis	18
3.3 GUI Design.....	19
3.4 Tool Functionality.....	21
3.5 Algorithm Design.....	24
4 Implementation	28
4.1 Development Platform.....	28
4.2 Python Packages.....	29
4.3 Data Structures.....	30
4.4 Classes and Functions.....	31

5 Testing and Evaluation	41
5.1 Functional Testing	41
5.2 Non-Functional Testing	42
5.3 Evaluation of Artefact	42
6 Summary	47
6.1 Main Findings	47
6.2 Project Evaluation	49
6.3 Conclusion	51
References	52
Appendices.....	55

1 Introduction

This project details the research and development of an application to visualise maze generation and pathfinding solutions. The goal is developing an application that can be used to supplement education around algorithms, as it will take abstract mathematical ideas and turn them into something interactive.

1.1 Background Information

Pathfinding algorithms are invisible tools present in everyday life without people realising. For example, they are necessary for most modern video games [1] and for google maps to plan routes for people work commutes or holidays [2]. Behind the scenes, they are programming routes to move along edges to match nodes, which are weighted according to values programmed into them. A traffic jam may be represented in these graphs by adding heavier weight to that edge than another edge which is longer but has less traffic.

The algorithms themselves however are very abstract, often being presented as lines of mathematical equations of pseudocode designed to help implementation and reduce the algorithm to its most basic elements. An example of this can be seen in Figure 1. This abstraction proves a difficult hurdle to developing knowledge and understanding of these algorithms and how they are implemented.

```
Initialize an empty stack for storage of nodes, S.  
For each vertex u, define u.visited to be false.  
Push the root (first node to be visited) onto S.  
While S is not empty:  
    Pop the first element in S, u.  
    If u.visited = false, then:  
        U.visited = true  
        for each unvisited neighbor w of u:  
            Push w into S.  
End process when all nodes have been visited.
```

Figure 1 - Depth-first search pseudocode from Brilliant.org [3]

1.2 Problem Statement

Pathfinding algorithms are abstract and difficult to perceive in their basic form.

Understanding the decisions an algorithm makes is arduous on its own, but to understand their behaviour requires experimentation and data. This in turn warrants visualisation techniques to internalise the information.

Developing an application that eases a user into developing knowledge and understanding of pathfinding will require taking these degrees of abstraction and developing tangible information for a user to interact with.

1.3 Project Scope

The scope of this artefact is to assist in educating people with understanding pathfinding algorithms, by demonstrating the behaviour and offering data relevant to the performance. Specifically, it would likely be useful for a computer science student in their second year, though it could realistically help anyone interested in learning provided they have a foundational understanding of what an algorithm is and what qualities are preferred. In practical terms, it is a direct comparison tool for pathfinding algorithms.

1.4 Project Aims and Objectives

The goal of this project is to develop an application that can accurately execute different algorithms, both to generate random mazes and to solve them with pathfinding algorithms. It must deliver the resulting information to a user through a visual display, along with statistics for the performance of the given instance such as the time taken, or the number of cells searched.

The aim of this is that users will be able to develop a deeper understanding of different algorithms by associating their practical experience with the visualisation and act of interacting with them directly. This would include the pros and cons of different algorithms, as well as the general behaviour.

I also hope that in implementing these algorithms to an application, I will also develop a deeper understanding of the algorithms.

2 Research

Understanding this topic requires research into the fields of both maze generation algorithms and pathfinding algorithms. The value of different algorithms and their effectiveness at different tasks must be understood before the design, as demonstrating these qualities is the primary goal of the artefact.

2.1 Topic Research

Mazes are important to current technologies as they are used within computer gaming, robotics, and architecture [4]. Mazes are a form of finite connected graphs [5], where the goal is typically to get from one node to another. The challenge of developing effective solutions comes from the finite number of edges present in the graph. Mazes are a common representation of an unweighted graph, which could also be represented as a weighted graph with nodes connected by edges that have a constant weight.

Creating procedurally generated mazes requires an algorithm to fit within certain boundaries. For an accurate maze algorithm, the result must include at least one pathway from any given cell to any other cell [6]. Mazes possess several objective qualities, such as if they are a perfect maze or have a bias. A perfect maze is one that resembles a spanning tree, otherwise known as having no loops [7]. There is precisely one path from any cell to any other cell and no more. An imperfect maze doesn't follow this rule – there are loops present in the maze that result in one or more paths from any cell to another [6].

Bias is another important distinction between maze generation algorithms that must be understood. When a maze is generated, they often come with a bias which represents a tendency to generate pathways in one direction. The binary tree method of maze generation is a clear example of this, where all pathways have a strong bias towards the corner opposite the origin [6]. Creating unbiased mazes, otherwise known as a uniform random spanning tree [8], is difficult and often more expensive, as the algorithm must contain some amount of randomisation and wasted cycles.

Pathfinding algorithms for static graphs compute an entire path from the beginning to the end, given that they know which edges are traversable. They are seen commonly in video games for non-player character movement [9], robotics for navigating real world spaces [10], and GPS machines for route planning atop a weighted graph representation of a real world space [11]. Pathfinding algorithms can be judged on the execution time and the consumption of memory [12].

The two most ubiquitous pathfinding algorithms are breadth-first search and depth-first search. Breadth-first search maintains a growing queue (FIFO) of sources to explore and can be used for web crawling to collect data [13] and will guarantee finding the shortest path between any two nodes [14]. Depth-first search recursively searches each possible path until it reaches the solution or a previously encountered node [15], which could find a path quicker than breadth-first but does not guarantee it, nor does it guarantee the found path to be the shortest.

The A* algorithm is well known for its use in video games, as it is a best-first shortest path finding algorithm, which means that it prioritises finding a path in the shortest amount of time, rather than finding the shortest path overall. For video games, this means using the fewest compute cycles and allowing games to operate smoothly in real-time [16]. The heuristic function of an A* search is what keeps the algorithm running optimally. Heuristic searching decides on which node of a graph is to be explored next by calculating the heuristic cost. This calculation differs by method, and these methods can have different uses. One such heuristic calculation is the Manhattan which is the sum of the horizontal and vertical coordinates. The benefits of this are a quicker calculation resulting in less search time and a faster runtime. It however tends towards less optimal paths than heuristics such as the Euclidean heuristic, which uses Pythagoras' theorem to estimate a diagonal path at the cost of more compute [17].

Generating and solving mazes as unweighted graphs is a low-level form of higher form mathematical problems such as solving Hamiltonian problems [18] or finding minimum spanning tree solutions to graphs for brain network analysis [19].

2.2 Literature Review

In *Mazes for Programmers*, Buck describes several maze generation algorithms including both the binary tree algorithm and Aldous-Broder. A binary tree maze is defined as having a strong diagonal texture, which would likely be useful in showing the inefficiency of algorithms that do not respond well to a strong texture. Buck also found that Aldous-Broder mazes tend towards dead ends, estimating that 30% of the cells will be surrounded by walls and used heatmaps to judge that they had no bias. This type of maze could be a useful judge of how pathfinders deal with dead ends and mistakes, as well as judging if the pathfinder has their own bias towards certain maze textures. Importantly, both algorithms are described as being easy to implement in software, which makes them obvious choices to include in this project [6].

Depth-First Search is a common pathfinding tool renowned for the wide applications and relies on a stack-based data structure [20]. Breadth-First Search exists atop a queue-based data structure and is relevant to big graph applications such as web traversal and requires persistent memory accesses [21]. Both of these algorithms operate with a time complexity of $O(|N| + |E|)$ time, where $|N|$ represents the nodes and $|E|$ represents the edges within a graph G . Both pathfinders will be effective demonstrations of different data structures operating in the same time complexity.

A* excels in video games because it is effective in mutating graphs, when it must adjust to new obstacles. It is considered computationally demanding for the purposes of solving a static graph, which is the kind seen in this artefact [20] [22]. A* does not have a static complexity as the heuristic is subject to change, however with no heuristic is functionally the same as Dijkstra's Algorithm which operates with a complexity of $O(|E| + |N| * \log(|N|))$. The ability to modify a heuristic to achieve different goals with pathfinding, as well as the overlap with Dijkstra's Algorithm, makes A* a strong choice for implementation.

3 Design

The artefact is an app designed to demonstrate the effectiveness of pathfinding algorithms through visualisation techniques. The user can create randomised mazes using different maze generation techniques with different qualities – whether a maze is perfect or imperfect, and whether or not it has a bias. The user can then select a pathfinding algorithm and see it complete in real time, with data outputs showing the length of the solved path, the number of cells that have been searched and the time elapsed.

3.1 Overview of the design

The app will rely on the correct implementation of several algorithms, both to generate the mazes and to solve them. There are many algorithms that exist to fit this criterion and understanding the value of different algorithms before implementation is important. The bulk of the workload in the artefact creation will be implementing the algorithms from mathematical language or pseudocode into python code that correctly interacts with other elements. If the wrong algorithms are chosen, this could severely impact the ability to complete the project on the suggested timeline.

There are two features of maze generation that were considered when choosing algorithms. Whether a maze is perfect, and whether it has a bias. Choosing algorithms with a good spread of features is necessary for the app to be successful in demonstrating the value of different pathfinders.

The maze generation algorithms chosen for implementation are Binary Tree and Aldous-Broder. Binary tree mazes are perfect with a bias, and Aldous-Broder mazes are perfect with no bias. To cover 3/4s of the potential features, a third algorithm will be created that modifies binary tree to create an imperfect maze. It will maintain the same bias but allow for different pathing options to the end which is integral to demonstrate how pathfinders may find different solutions to the problems.

For pathfinding, there will be seven options. However, there are in effect only three actual algorithms implemented, but with modifications made. The first two are breadth-first

search (BFS) and depth-first search (DFS). BFS will only consist of one algorithm, while DFS will be split into two. As DFS also has a bias, like mazes, there will be a version with the same bias as the maze (south-east), and a version with the opposed bias (north-west). This will demonstrate how DFS can be effective in the right scenario, but completely inefficient in the wrong situation, as it will take a long time to solve the maze and likely find a much longer solved path.

The third algorithm implemented will be the A* algorithm. A* has two components – the cost of moving to the current cell, and a calculation known as the heuristic. By modifying this heuristic, A* can be very versatile, and in the case of this artefact split into 4 different algorithms. The first will have a constant heuristic, which is also known as Dijkstra's algorithm. This will function similarly to breadth-first, though with a different code base.

The second and third versions of A* will use the Euclidean and Manhattan heuristics, which are estimates of the remaining distance between the current cell and the goal. Euclidean calculates this using Pythagoras theorem, while the Manhattan heuristic is simply the addition of moves in the x axis and moves in the y axis.

The final A* algorithm will be dysfunctional, using a heuristic that motivates the algorithm to stay away from the goal. The algorithm naturally seeks out cells with the lowest value, and by using an inverted heuristic calculation the value of cells closer to the goal will increase, and the algorithm will avoid getting close to the goal.

Important across these algorithms is also the ease of implementing them into python code. They are largely uncomplicated algorithms relative to the complexity of algorithms within the same domain, and there is plenty of documentation online including pseudocode and mathematical explanations to support in the creation of the artefact.

The artefact structure will consist of 3 components – the main portion of code, the maze generation code, and the pathfinding code. This separation allows for the main to handle the large number of variables required to understand the needs of the user and pass them through to the 6 child classes. The class diagram in Figure 2 shows the virtual topology of the classes, how the algorithm child classes are dependent on the Main class, as well as some of them being dependent on the Tools class.

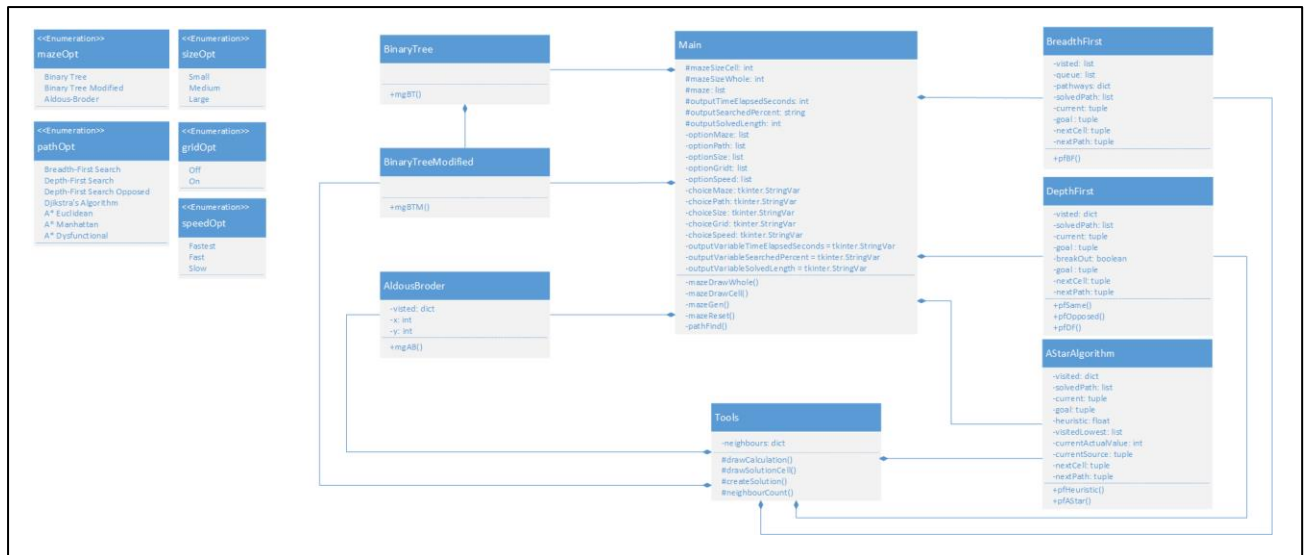


Figure 2 - UML Class Diagram

Due to the structure of the tkinter package within python, some variables appear as if duplicated – dropdown menus and radio buttons are used to present the options to the user, and the current choice is stored in the StringVar equivalent. For example, *gridOpt* will store the options 'Off' and 'On', and the current selection store in *gridChoice*. This works so that when a function is called that requires the grid such as *create()* or *drawSolution()*, the program will call *gridChoice.get()* to find the current selection [23]. All these user choices are shown in the enumeration of the class diagram.

The GUI will be drawn in the Main class using visualisation functions from the tkinter package, such as OptionMenu, Radiobutton, Button, and Frame. On top of that, Main will also handle the visualisation of the maze. The *maze* attribute of the Main class will be a list of lists, otherwise referred to as a 2D array. Each secondary list index will refer to a different row number or x coordinate, while the element index of the elements within a secondary list refer to the y coordinate. The value of this data structure lies in the similarity to standard Euclidean geometry notation, allowing for much of the code to be written in similar logical terms as seen in Figure 3.

```

SET maze =[ [ (1, 1), (1, 2), (1, 3) ],
             [ (2, 1), (2, 2), (2, 3) ],
             [ (3, 1), (3, 2), (3, 3) ] ]

SET x = 2
SET y = 3

print ( maze[x][y] )
#console will print (2, 3)

```

Figure 3 - Pseudocode detailing maze data structure

Each algorithm will be separated into its own class. While some algorithms may bear similarities in their requirements to function, no two algorithms require the same attributes from both Main and Tools. This makes it difficult to create a true superclass or package to represent any given set of algorithms.

It is noteworthy that all pathfinding algorithms use a variable called *visited*, and yet the data type changes. The purpose of recording visited cells differs between algorithms. Breadth first uses it to avoid revisiting already visited cells in imperfect mazes, as this would cause it to exponentially create new instances of cells to explore. Depth first however, tracks visited because it also needs to return to a previous cell if the current exploration fails to find a solution. It is important that each given cell has associated values to describe which adjacent cells have already been explored. While it would be possible to use a list for this, it would require additional code to ensure data validation, so it is more practical to use the data validation present with the dictionary data type.

Another area of interest is the relationship between the BinaryTree class and the BinaryTreeModified class. The modified binary tree algorithm will exist on top of the original algorithm, modifying the result to create an imperfect maze. As such, it cannot exist without the BinaryTree class.

3.2 Requirement of the Analysis

The only software requirement for this program to run will be the installation of python version 3.11.3 This version of python requires an AMD or Intel CPU, 4GB of RAM and 5GB of free disk space, as well as running on at least Windows 7, macOS X , or Ubuntu 16.04 [24].

3.3 GUI Design

As this app's primary focus is to visualise algorithms and allow for user interaction, it is important to import the tkinter package available in python. Tkinter is the primary GUI package for python and can draw windows and fill them with shapes, as well as interactive items such as buttons and drop-down menus.

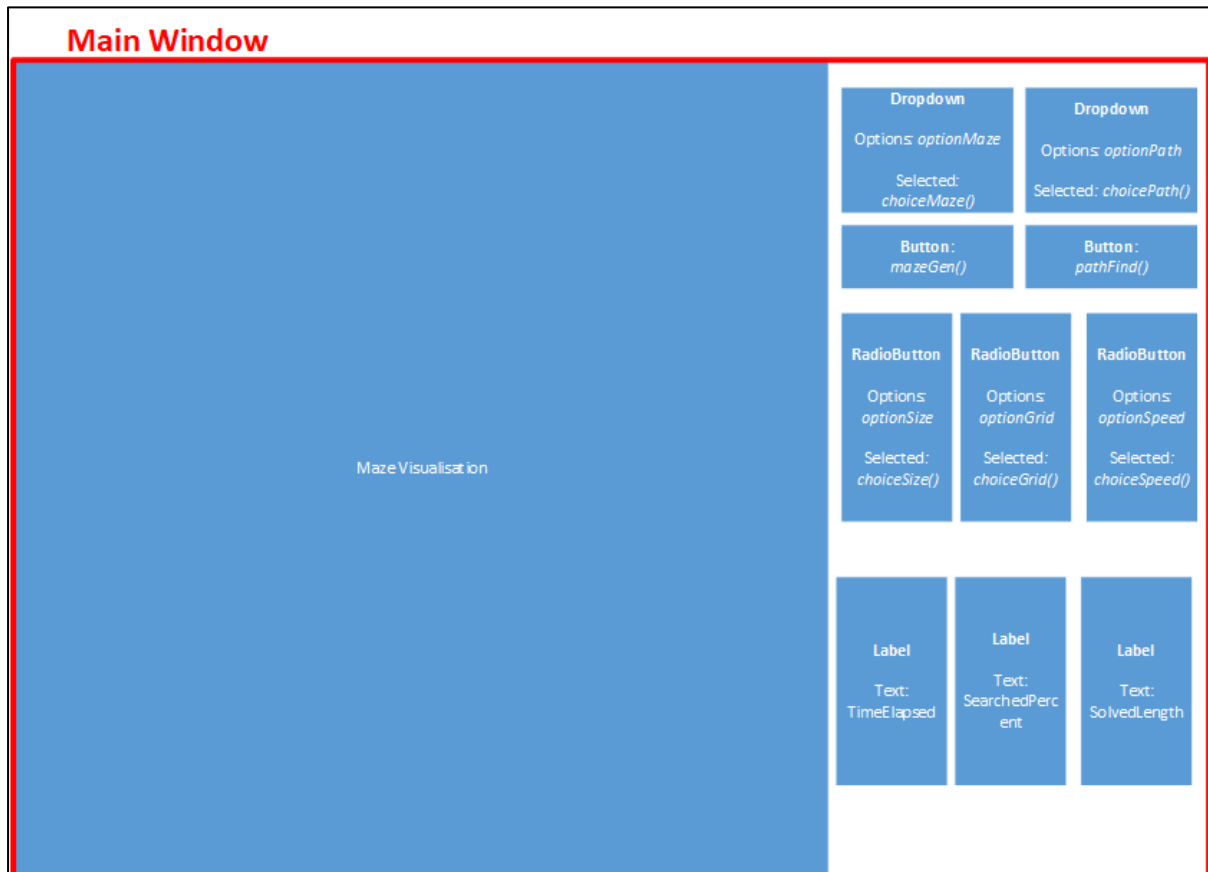


Figure 4 - Design sketch for GUI

The proposed UI can be seen in Figure 4. Using radio buttons and drop-down menus for the user settings will be sufficient for the user interaction, and the two buttons near the top will allow the user to activate the algorithms. The labels at the bottom will update every time a pathfinder is completed with the relevant information. The user interactive experience is not a focus of the artefact, and as such it will be sufficient to use these simple elements for the UI.

The visualisation for the maze will take up two thirds of the window. As seen in Figure 5, the maze will generate walls as blocks rather than single lines between cells. This is an important distinction to make early in the design, as it will affect the implementation of all

the algorithms on the back end as well as affecting the user experience. It is important if the algorithms are to visualise the information that a user can easily interpret what they are seeing, and so the artefact will be using blocks so that at larger scale mazes a user can easily see where the walls are placed.

The maze will initialise with only the default cells, which are outer walls, walls, and empty rooms. Then the maze generation algorithm will operate by deleting walls between empty rooms according to the settings, which is represented by the yellow cells. These generated empty rooms in yellow will be the passageways between the pre-existing empty rooms.

Certain patterns can be found looking at these graphics. For example, if the coordinates of a cell are both even, it is guaranteed to be an empty room on initialisation. If a cell has both odd coordinates, then it will be an irreplaceable wall. These simple logical statements will be important during implementation of the algorithms.

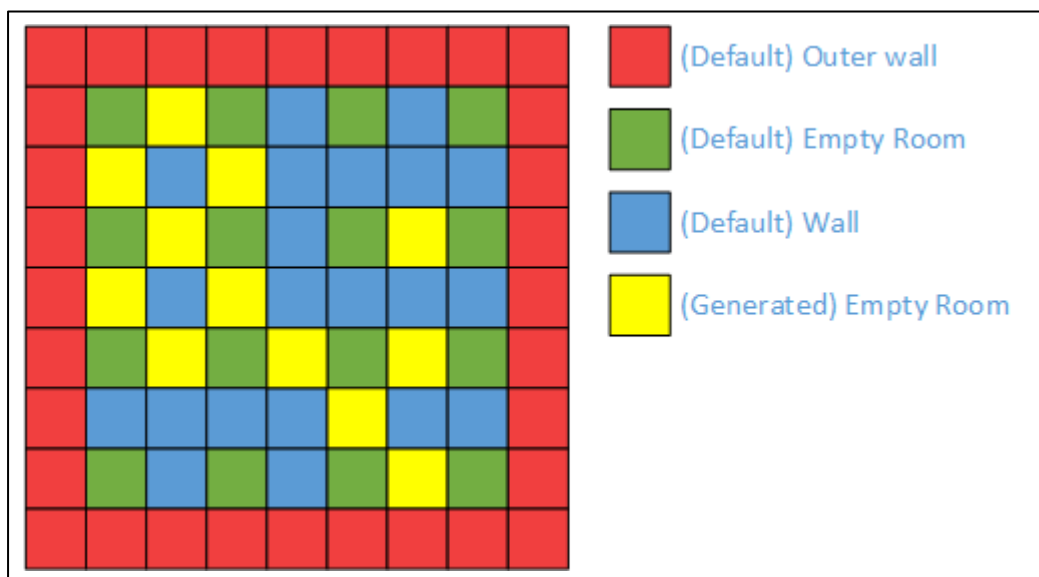


Figure 5 - Design of maze visualisation with key

Figure 6 shows a colour-accurate representation of a maze that has been both generated and solved with a breadth-first search. Outer walls will be indistinguishable from regular walls as that would create unnecessary visual complexity and should be invisible to the user. The user only needs to be able to distinguish between the four cells as seen depicted in the key. Using green to indicate a solved path and yellow path is informed largely by the ubiquity of the traffic light colouring system and should be easily understood by any user especially when given the context of animation.

While the pathfinder algorithm is running, the visualisation will be updating on timed intervals using the Time package within python and decided by the variable *choiceSpeed()*. That way each step of the animation is visible to the user. The other important distinction is that pathfinders will only explore actual rooms with even cell coordinates. The passageways are 1-way and therefore will be skipped and painted as an explored room for the sake of visibility to the user. This way they can still understand the direction the pathfinder is exploring from.

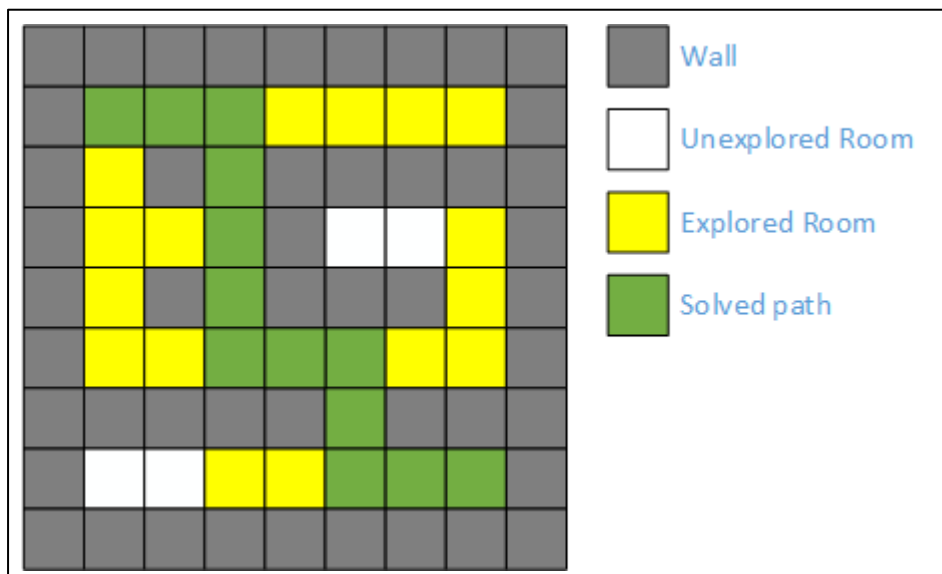


Figure 6 - Design of colour accurate pathfinding visualisation with key

3.4 Tool Functionality

There are some custom functions in the program that are required for functionality which are shared between all algorithms, or some algorithms that share a similar set of requirements. An example of these is the pipeline for drawing cells. Firstly, a function such as the maze generation or pathfinder will pass coordinates to a function that handles the entire drawing output. This is either *mazeDrawWhole()* or *drawSolutionWhole()*. These both use a for loop to extract information from the arrays relevant to the individual cells and convert those into physical coordinates on the screen to draw a cell. Once that is done, they can call the function to draw a cell. An example of this pipeline can be seen in Figure 7 and Figure 8, with *mazeDrawWhole()* and *mazeDrawCell()* used when generating a maze initially.

```

FOR every row in the maze
  FOR every column in the row
    IF i == 2 AND j == 2 THEN                                # if starting cell
      colour = blue
    ELSE IF i == mazeSizeWhole AND j == mazeSizeWhole THEN    # if final
      colour = red
    ELSE IF column == room THEN
      colour = white
    ELSE THEN                                                  # if wall
      colour = black
    END IF

    CALL mazeDrawCell( row, column, colour)
  END FOR
END FOR

```

Figure 7 - Pseudocode for mazeDrawWhole()

```

IF row == 1 AND column == 1 THEN                             # if first cell, use predefined
  coordinates
  CREATE RECTANGLE (startingX, startingY, mazeSizeCell, mazeSizeCell, colour)
ELSE THEN
  SET x1 = row * mazeSizeCell + startingX                    # generate coordinates for rectangle
  SET y1 = column * mazeSizeCell + startingY
  SET x2 = x1 + mazeSizeCell
  SET y2 = y1 + mazeSizeCell

  CREATE RECTANGLE (x1, y1, x2, y2, colour)
END IF

```

Figure 8 - Pseudocode for mazeDrawCell()

The entire premise of the application rests on this unique function, as it creates the visualisations. If it doesn't work correctly, then even if the algorithms work the user would have no way of knowing.

Another key function used is *neighbourCount()* from the *Tools* class. This is important to implementing correct functionality in all but one of the algorithms, as seen by all the dependencies in Figure 2. This function will analyse the current maze array and generate a dictionary that lists every adjacent passageway to that given cell. It will use a for loop similar to *mazeDrawWhole()*, looping through every set of coordinates for a complete set of information. Proposed pseudocode for this function can be seen in Figure 9. As seen, for each cell it runs four if statements, each checking for a passageway in one of the four cardinal directions. When a statement proves true, it appends the dictionary entry for that coordinate with the direction available to it.

```

FOR every row in the maze (i)
    IF the row is odd numbered THEN                # both coordinates must be odd

        FOR every column in the maze (j)
            IF the column is odd numbered THEN

                SET neighbours ( i , j ) to []      # create empty dictionary entry
                IF passageway to east is clear THEN
                    neighbours[ ( i , j ) ] APPEND "East"
                END IF

                IF passageway to south is clear THEN
                    neighbours[ ( i , j ) ] APPEND "South"
                END IF

                IF passageway to west is clear THEN
                    neighbours[ ( i , j ) ] APPEND "West"
                END IF

                IF passageway to north is clear THEN
                    neighbours[ ( i , j ) ] APPEND "North"
                END IF

            END IF
        END FOR
    END IF
END FOR

```

Figure 9 - Pseudocode for *neighbourCount()*

3.5 Algorithm Design

Some of the algorithms used by this artefact do not need to be expanded upon from the explanations found in Chapter 2. Some however warrant further exploration of the design – how the algorithm will be interpreted into code may not be a simple set of one-to-one decisions. This includes the modified binary tree, the depth-first search and the A* algorithm.

A standard binary tree maze creates a perfect maze with a strong bias. The goal of modifying the algorithm is to create a maze with no dead ends, and therefore creating a solvable imperfect maze with many potential solutions. To this end, the algorithm will first create a standard binary tree maze by calling the appropriate function. It will then use *neighbourCount()* to generate the neighbour dictionary. After that, it will run a for loop through the entire dictionary, searching for any cell that has only one neighbour which indicates that it is a dead end. When a dead end is found, it creates an adjacent passageway to any unconnected adjacent room. Figure 10 shows partial pseudocode for this function, showing how the function would handle any given cell if it's current only neighbour is to the north. This can be extrapolated for further directions and including simple logical conditions to check for cells that might be on the boundary of the maze. For example, if a cell uses an x coordinate of 1, then it must be along the first column of the maze and cannot create a western passageway.


```

FOR every key in neighbour dictionary           # for every entry in the dictionary
  IF there is only one neighbour in key THEN   # if only one neighbour ( dead end)
    IF neighbour is north THEN
      IF random integer is 0 THEN
        neighbours[ ( i , j ) ] APPEND "East"
        SET maze[ i + 1, j ] = passageway
      END IF

      ELSE IF random integer is 1 THEN
        neighbours[ ( i , j ) ] APPEND "South"
        SET maze[ i , j + 1 ] = passageway
      END IF

      ELSE IF random integer is 2 THEN
        neighbours[ ( i , j ) ] APPEND "South"
        SET maze[ i - 1, j ] = passageway
      END IF
    END IF
  END IF

  .... # repeat for every potential direction, including checks for
  boundaries

```

Figure 10 - Pseudocode for modified binary tree generation

Depth-First Search consists of three different functions. The primary function called will initialise the variables, call one of two algorithm functions, and then draw the maze to screen and return output variables. The distinction of the algorithm comes from the bias – whether the pathfinder is going to prioritise moving in a southeast direction or a northwest direction. These biases will be handled by two separate functions which use the same algorithm and the same actual code, but the order of the lines is altered. Where the function *pfSame()* will check in the order of east, south, west, north, *pfOpposed()* will check west, north, south, east. Partial pseudocode for *pfSame()* can be found in Figure 11, detailing how the function calls the same search until it finds a solution, and then as the function recurs through itself, it adds the cells to the solution path. When it is finished, it will return the *visited* list to the primary function, which can then pass that to the *drawSolutionWhole()* function and complete the search.

```

SET breakout to false
IF ( breakout is false ) AND ( current cell has a neighbour to the east ) THEN

    SET nextCell to empty room to the east
    SET nextPath to passageway between current cell and nextCell
    REMOVE east from current cell entry in neighbours dictionary

    # Calls next level of recursion to continue exploring
    IF ( nextCell is not the goal ) AND ( nextCell has not been visited ) THEN

        # remove origin from next cell in neighbours to avoid loop
        REMOVE west from nextCell entry in neighbours dictionary
        APPEND nextCell to visited array
        SET goal, breakOut, visited = CALL pfDFSsame()
    END IF

    IF nextCell is the goal THEN

        APPEND goal to solvedPath
        SET goal to currentCell
        RETURN goal, true, visited
    END IF
END IF

... # repeat for all directions

IF ( breakout is false ) AND ( current cell has a neighbour to the south ) THEN
    ...
END IF

IF ( breakout is false ) AND ( current cell has a neighbour to the west ) THEN
    ...
END IF

IF ( breakout is false ) AND ( current cell has a neighbour to the north ) THEN
    ...
END IF

# return to previous level of recursion and continue from there
RETURN goal, False, visited

```

Figure 11 - Partial pseudocode for depth-first search

The A* algorithm uses a combination of the actual cost of traveling the current cell and a heuristic to generate an estimated cost of completing the maze from the current cell. The algorithm itself will use nested for and while loops to continually check if the maze has been solved, and if it hasn't then find the cell with the best total cost and search the neighbours of that cell.

This algorithm is designed to represent 4 different algorithms within the artefact, and they all use the same algorithm. The only difference between them is the calculation used to find a heuristic. Each of the heuristic calculations can be seen in Figure 12, which represents the function *pfHeuristic()*.

```
IF heuristicChoice is Dijkstra's algorithm THEN
    SET heuristic = 0

ELSE IF heuristicChoice is Euclidean THEN
    SET heuristic = hypotenuse inputting remaining X distance and Y distance to
    pythagoras'

ELSE IF heuristicChoice is manhattan THEN
    SET heuristic = remaining X and Y distance added together

ELSE IF heuristicChoice is dysfunctional THEN
    SET heuristic = actual cost of the current cell

END IF
```

Figure 12 - Pseudocode for A* heuristic calculation *pfHeuristic()*

4 Implementation

The artefact was programmed using VS Code as an IDE and using Python as the programming language. All proposed design functionality has been implemented.

4.1 Development Platform

VS Code is an Integrated Development Environment or IDE, used in programming to provide more tools to the developer. Visual Studio Code was used as it is fit for purpose. It is quick, responsive and there is a plethora of online resources to troubleshoot problems. VS Code is intuitive to use, as changing settings is as easy as searching for them in the bar at the top of the screen. However, it is likely that many alternative IDEs such as Eclipse or Visual Studio would have been just as useful, as there are no specific requirements for this project. Familiarity was the deciding factor in this choice.

The artefact was programmed using the Python language, which was developed in the 1980s with the intention of making a simple and beautiful language. A common phrase in python documentation is developing “Pythonic” code, which essentially means the code is reduced to the simplest and most digestible form. Python also has a large ecosystem of libraries built to add modularity, and within data science python is a common language due to the data-oriented packages like NumPy and SciPy.

The decision to use Python was primarily based on its presence in the professional programming world and the wide range of applicability. Throughout most of the 3-year course, Java was the language of choice for programming in coursework. Diversifying the languages that I am familiar with will increase my employability and improve my understanding of programming as an overall skill. It is easy to compare the strengths and weaknesses of both Java and Python now that I have used them both for extensive programming solutions.

4.2 Python Packages

This program relies on several officially supported python packages to be imported. The most integral is the *tkinter* package which includes all of the functions required for GUI functionality. Without this, the program would be forced to resolve in the console and the aim of using visualisation techniques would be unsuccessful.

The primary uses of this package are to render the window that the program takes place in, as well as various objects such as the maze and the components of the user interface. This package is imported to *main.py* as this handles the majority of the visualisation code, as well as being used by the pathfinding algorithms in *pathFind.py* to render the animation.

Time is another important package imported, which achieves two objectives. The first is to create the data output of time elapsed for the pathfinding algorithms, which can be done by calling *time.time()* to record the current atomic time according to the operating system. It can then be called again when the algorithm is resolved, allowing the program to calculate time elapsed by subtracting the original time from the current time.

It is also used to enforce the user setting regarding animation speed, by calling *time.sleep()* to pause the program for a set amount of time between every sequential step of the algorithm. Without this intentional slow down, some algorithms may solve in under a second and any animation would be imperceptible to the user.

The remaining imports use only one imported function and are summarised below:

Package	Function	Use case
<i>random</i>	<code>randint()</code>	Allows for maze generation algorithms to choose random outcomes during sequential steps to achieve the creation of procedurally generated mazes.
<i>sys</i>	<code>setrecursionlimit()</code>	Increases recursion limit to allow for highest potential depth-first search solutions
<i>math</i>	<code>sqrt()</code>	Finds the square root of the argument, allowing for the function to calculate the hypotenuse in Pythagoras theorem to find the Euclidean heuristic of an A* algorithm.

4.3 Data Structures

The most common data in the project is the storage of the maze information in a list of lists, which is a type of array found in Python. Lists are ordered arrays, where elements are indexed and changeable, and allow for duplicate values. This 2D list stores the individual elements as either 1 or 0, representing a wall or a passageway. Each list within represents a column, and the elements within that list represent a row. By this nature, x and y coordinates can be referred to as *maze[x][y]* to replicate the cartesian coordinate system.

This data structure is initialised with a for loop that determines every element is a wall. After that, the *mazeGen* function will determine every non-edge cell as either a wall or passageway. The maze is structured such that every cell with both x and y elements being odd will be a passageway, and the algorithms will determine passageways between them. This creates a more visually appealing and cohesive image, as the boundaries between passageways are clear. The *maze* array is passed from *main.py* to every other file and class as they perform functions to it, whether that be generating a new maze and injecting the new cell data or running the pathfinding algorithm on top of it.

Lists are also used to store the choices made by the user in the drop down menus, including the choice of algorithms for maze generation, path finding, the size of the maze, the visibility of the grid and the speed at which the pathfinding animation plays. The final use of lists is to maintain the queue and record the correct path in algorithms such as breadth first.

Dictionaries are another type of array in Python, which contain a list of keys. These keys do not allow for duplicates and are ordered, and each key is paired with a value. One dictionary is the *neighbours* dictionary, created by the *neighbourCount()* function in *pathFind.py* – it stores a list of every cell coordinate as the key. The value paired with each key is whether they have an open pathway in any of the four cardinal directions. For example if cell (5,5) connects to the north, south and west then the dictionary element would look like '(5,5) : N, S, W'. This is important for several of the pathfinding and maze generation algorithms to run, as both counting the number of neighbours and knowing which neighbours are available is important. For example, the modified binary tree algorithm specifically searches for cells which have only one neighbour, allowing for it to find and modify dead ends to ensure an imperfect maze is created.

A dictionary is also used in the Aldous-Broder implementation, as it uses a dictionary to track which cells have been visited and from which cell. This is important for the correct implementation of the algorithm, as it continually refers to visited cells to move around the cells that it has previously visited to open new passageways.

When stored, cell coordinates are often stored as a tuple. This is an ordered, unchangeable list which allows duplicates, and is useful when storing data that doesn't need to be modified. These are used as keys in the *neighbours* dictionary, as well as being stored this way in the pathways and correct path arrays.

It is noteworthy that there is a default recursion limit enforced by python, which had to be overwritten for the depth-first search to complete on large mazes. If a perfect maze were to have the longest possible path, it would require 79^2 or 6241 recursions to successfully resolve.

4.4 Classes and Functions

Classes are used in both secondary files that are imported into *main.py*. The classes have been implemented as designed in Chapter 3.1, collecting the maze generation classes into *mazeGen.py* and the pathfinding classes into *pathFind.py*, along with the Tools class.

Below is a summary of the functions found within *main.py*.

Main	Purpose	Arguments	Parent Function
<i>mazeDrawWhole()</i>	Manages the visualisation of the maze	maze(list)	
<i>mazeDrawCell()</i>	Draws individual cells of the maze	row(int), col(int), colour(string)	<i>mazeDrawWhole()</i>
<i>mazeGen()</i>	Assembles user settings and calls maze generation functions		
<i>mazeReset()</i>	Resets pathfinding visualisation		
<i>pathFind()</i>	Assembles user settings and calls pathfinding functions		

Table 1 - Main class and functions

These functions are all uncomplicated, with minimal nesting or variation. It is worthwhile however to explore the implementation of the functions to render visualisation in comparison to the designs seen in Figure 7 and Figure 8.

```

def mazeDrawWhole(maze):
    canvas.delete("all") # avoid memory leak
    for i, row in enumerate(maze):
        for j, col in enumerate(row):
            if i == 1 and j == 1: # print starting cell
                colour = "#7a68d4"
            elif i == mazeSizeWhole - 2 and j == mazeSizeWhole - 2: # print final cell as red
                colour = "#a31a00"
            elif str(col) == "c": # print clearings as white
                colour = "white"
            elif str(col) == "w": # print walls as black
                colour = "#22323d"
            mazeDrawCell(i, j, colour)

```

Figure 13 - Implemented code for `mazeDrawWhole()`

```

def mazeDrawCell(row, col, colour):
    # cells are drawn 2px larger on both length and width if there is no grid
    if choiceGrid.get() == "Off":
        if row == 0 and col == 0:
            canvas.create_rectangle(11, 11, 30, 30, fill="#22323d", outline="#22323d")
        else:
            x1 = (row * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
            x2 = x1 + mazeSizeCell
            y1 = (col * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
            y2 = y1 + mazeSizeCell
            canvas.create_rectangle(x1, y1, x2, y2, fill=colour, width=0)
    else:
        if row == 0 and col == 0:
            canvas.create_rectangle(11, 11, 30, 30, fill="#22323d", outline="#22323d")
        else:
            x1 = 1 + (row * mazeSizeCell) + (0.5 * mazeSizeCell)
            x2 = x1 + mazeSizeCell
            y1 = 1 + (col * mazeSizeCell) + (0.5 * mazeSizeCell)
            y2 = y1 + mazeSizeCell
            canvas.create_rectangle(x1, y1, x2, y2, fill=colour, outline="#22323d")

```

Figure 14 - Implemented code for `mazeDrawCell()`

As seen in Figure 13 and Figure 14, both functions have been programmed into python like-for-like with the pseudocode initially proposed. There are two primary differences between the design and implementation here however. The first is that the coordinate system within the software is offset by -1 in both axis, as the first element in any list has an ID of 0 rather than 1. This is counteracted by the data structure, as the maze is surrounded by an unbroken ring of walls. This enforces the coherence with the cartesian coordinate system as demonstrated in Figure 3.

The other difference is that `mazeDrawCell()` now checks for the user setting `choiceGrid` before creating the visualisation. This user setting slightly affects the positioning and settings of each individual cell, as excluding the grid requires the cell coordinates to be offset by one pixel, to change the border width and have no declaration of border colour.

BinaryTree	Purpose	Arguments	Parent Function
<i>mgBT()</i>	Creates a maze using the binary tree method	maze(list)	<i>Main.mazeGen()</i>

Table 2 - BinaryTree class and function

BinaryTreeModified	Purpose	Arguments	Parent Function
<i>mgBTM()</i>	Creates a maze using the modified binary tree method	maze(list)	<i>Main.mazeGen()</i>

Table 3 - BinaryTreeModified class and function

AldousBroder	Purpose	Arguments	Parent Function
<i>mgAB()</i>	Creates a maze using the Aldous-Broder method	maze(list)	<i>Main.mazeGen()</i>

Table 4 - AldousBroder class and function

The tables above show the classes and respective functions within the maze generation class group (*mazeGen.py*). These each implement the functions

Tools	Purpose	Arguments	Parent Function
<i>drawCalculation()</i>	Draws individual cells of the pathfinder as it finds the solution	canvas(tkinter) mazeSizeCell(int), row(int) col(int) grid(string)	
<i>drawSolutionCell()</i>	Draws individual cells of the solution	canvas(tkinter) mazeSizeCell(int) row(int) col(int) grid(string)	drawSolutionWhole ()
<i>drawSolutionWhole()</i>	Manages the visualisation of the Solution	canvas(tkinter) mazeSizeCell(int) solvedPath(list) grid(string) speed(float)	
<i>neighbourCount()</i>	Populates a dictionary array with a key for every cell of the maze. The value of each key is a list storing the directions of neighbours.	maze(list)	

Table 5 - Tools class and functions

The tools class contains four functions, three of which are used to visualise the pathfinding animation. They are functionally similar to the main class functions regarding visualisation, but for the purposes of simplicity this program uses different functions for different coloured cells, rather than calling one function with several nested statements for different colours.

Shown in Figure 15 is the implemented function for *neighbourCount()*, in contrast to the pseudocode shown in Figure 9. This function when implemented looks strikingly similar to the pseudocode design. The if statements condition of 'if row/column is odd numbered' is replaced with a simple calculation to compare the modulus with 1; as the modulus of all odd numbers is equal to 1 this is a practical solution to the pseudocode condition.

The condition of 'If passageway to [direction] is clear' is replaced with the calculation of the actual coordinates in the maze list. By adding 1 to i, which represents the x coordinate, then the code can compare against a coordinate immediately to the east of the current cell.

```
def neighbourCount(maze):
    neighbours = {} # dictionary storing cells as key and neighbour directions as associated values
    for i in range(len(maze) - 1):
        if i % 2 == 1:
            for j in range(len(maze) - 1):
                if j % 2 == 1:
                    if maze[i][j] == "c":
                        neighbours[(i, j)] = []
                        if maze[i + 1][j] == "c":
                            neighbours[(i, j)].append("E")
                        if maze[i][j + 1] == "c":
                            neighbours[(i, j)].append("S")
                        if maze[i - 1][j] == "c":
                            neighbours[(i, j)].append("W")
                        if maze[i][j - 1] == "c":
                            neighbours[(i, j)].append("N")
    return neighbours
```

Figure 15 - Implemented code for *neighbourCount()*

<i>BreadthFirst</i>	Purpose	Arguments	Parent Function
<i>pfBF()</i>	Solve the current maze using Breadth-First Search	canvas(tkinter) mazeSizeCell(int) maze(list) grid(string) speed(float)	<i>Main.pathFind()</i>

Table 6 - *BreadthFirst* class and function

The *BreadthFirst* class contains only one function, for solving a maze using breadth-first search. As this algorithm uses a queue-based structure, the implementation is required to interact with a list containing a queue several times which can be seen in Figure 16.

The queue is initialised with the starting cell when the program first runs. Then a while loop begins and continues to run as long as the queue is not empty. When the loop begins, it pops the coordinate out of the bottom which was the first coordinate in and therefore the first out.

It then compares this coordinate against the goal to check if the maze has been solved. If it has, then it breaks out of the entire while loop and moves onto the next stage of the algorithm which involves animating the solution.

The next stage of the algorithm is a for loop which runs once for each direction a neighbour could exist in so that it can be recorded in the queue array. Each neighbour also calls the `drawCalculation()` function to generate an animation and communicate to the user what cells have just been explored.

Finally, each cell is recorded in the pathways dictionary. This dictionary records each coordinate as a key with the corresponding value equal to the original cell it was found from. This is used when the solution is found to work backwards, as each cell can recursively find their way back to the origin on the shortest path.

This is a while loop, meaning this code will continue to run until the goal has been found and it can move onto the second portion of the code which can be found in the appendix.

```

queue = [(1, 1)] # list to store cells yet to run through the algorithm
while len(queue) > 0:
    time.sleep(speed) # time.sleep used to slow down visualisation
    current = queue.pop(0)
    if current == ((len(maze) - 2), (len(maze) - 2)):
        break # used to break out of the while loop if the goal has been found
    for d in "ESWN":
        if d in neighbours[current]:
            if d == "E":
                nextCell = (current[0] + 2, current[1])
                nextPath = (current[0] + 1, current[1])
            elif d == "S":
                nextCell = (current[0], current[1] + 2)
                nextPath = (current[0], current[1] + 1)
            elif d == "W":
                nextCell = (current[0] - 2, current[1])
                nextPath = (current[0] - 1, current[1])
            elif d == "N":
                nextCell = (current[0], current[1] - 2)
                nextPath = (current[0], current[1] - 1)

        # draws explored cells for each given loop
        Tools.drawCalculation(canvas, mazeSizeCell, nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell, nextCell[0], nextCell[1], grid)

        if nextCell not in visited:
            visited.append(nextCell)
            queue.append(nextCell)
            pathways[nextCell] = current

```

Figure 16 - Code snippet of first while loop in *pfBF()* as implemented.

DepthFirst	Purpose	Arguments	Parent Function
<i>pfDF()</i>	Manage the two variations of the depth-first search algorithm as well as handle the solution visualisation	canvas(tkinter) mazeSizeCell(int) maze(list) grid(string) speed(float) bias(boolean)	<i>Main.pathFind()</i>
<i>pfSame()</i>	Solve the maze using the same bias as the mazes will have (southeastern)	goal(tuple) current(tuple) neighbours(dictionary) canvas(tkinter) mazeSizeCell(int) maze(list) grid(string) speed(float) solvedPath(list) visited(dictionary)	<i>pfDF()</i>
<i>pfOpposed()</i>	Solve the maze using a bias opposed to the maze bias (northwestern)	goal(tuple) current(tuple) neighbours(dictionary) canvas(tkinter) mazeSizeCell(int) maze(list) grid(string) speed(float) solvedPath(list) visited(dictionary)	<i>pfDF()</i>

Table 7 - *DepthFirst* class and functions

Depth-First Search is performed across three separate functions within the class, and consists of two variations of the same algorithm. As this search is performed with a bias, it is possible to run the algorithm with a bias similar to that of the maze, and one that is in opposition to the maze. Through this it is possible to generate less efficient solutions with imperfect mazes, and to demonstrate the unbiased nature of the Aldous-Broder maze generation. It is also effective at generating incredibly long and inefficient paths when paired with the modified binary tree maze as seen in Figure 17 in comparison to Figure 18.

The decision of which function to run is decided by a boolean passed from the *Main* class based on which algorithm is selected by the user. The code of both functions is identical, except that the order of if conditions is rotated by 2. Where the favourably biased depth-first search explores with the priority of east, south, west, north; the opposed bias explores north, west, south, east.

This algorithm uses a stack-based data structure and was implemented as planned and seen in Figure 11.

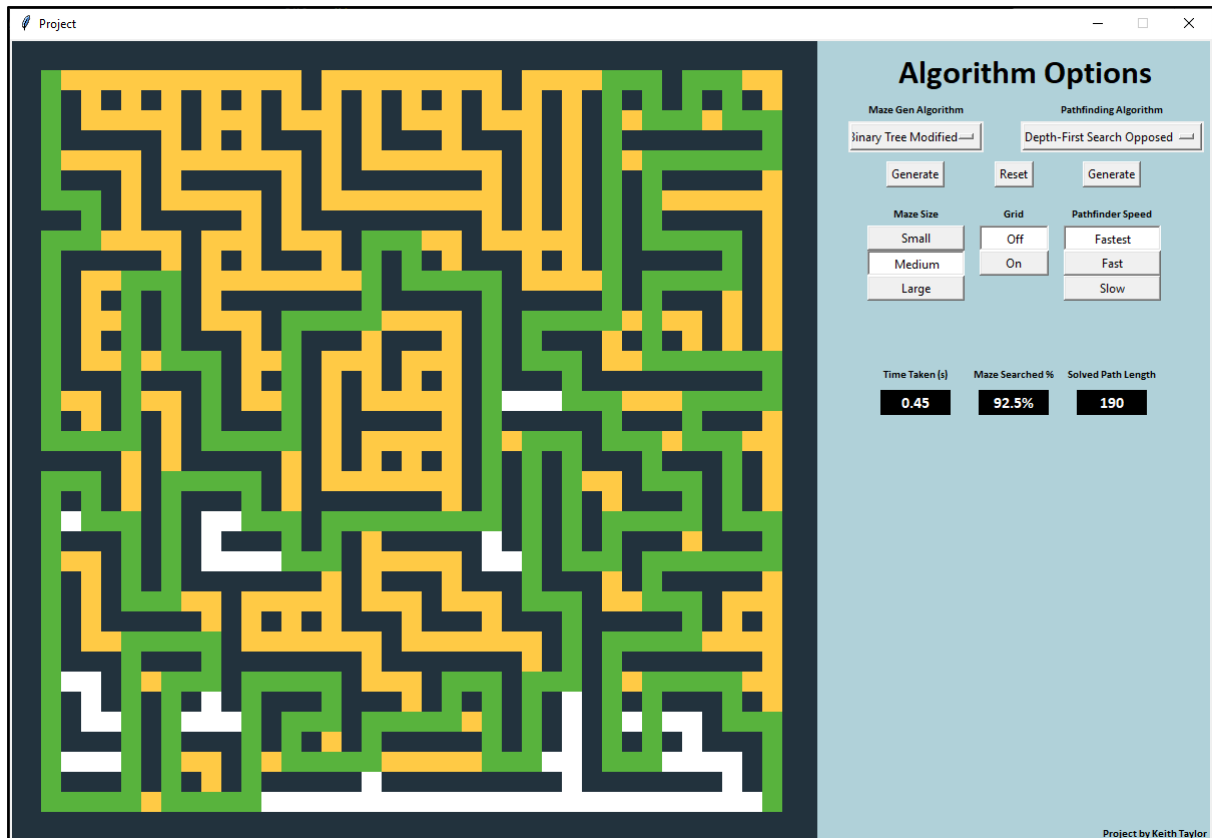


Figure 17 - Modified Binary Tree maze with Opposed Depth-First Search Solution

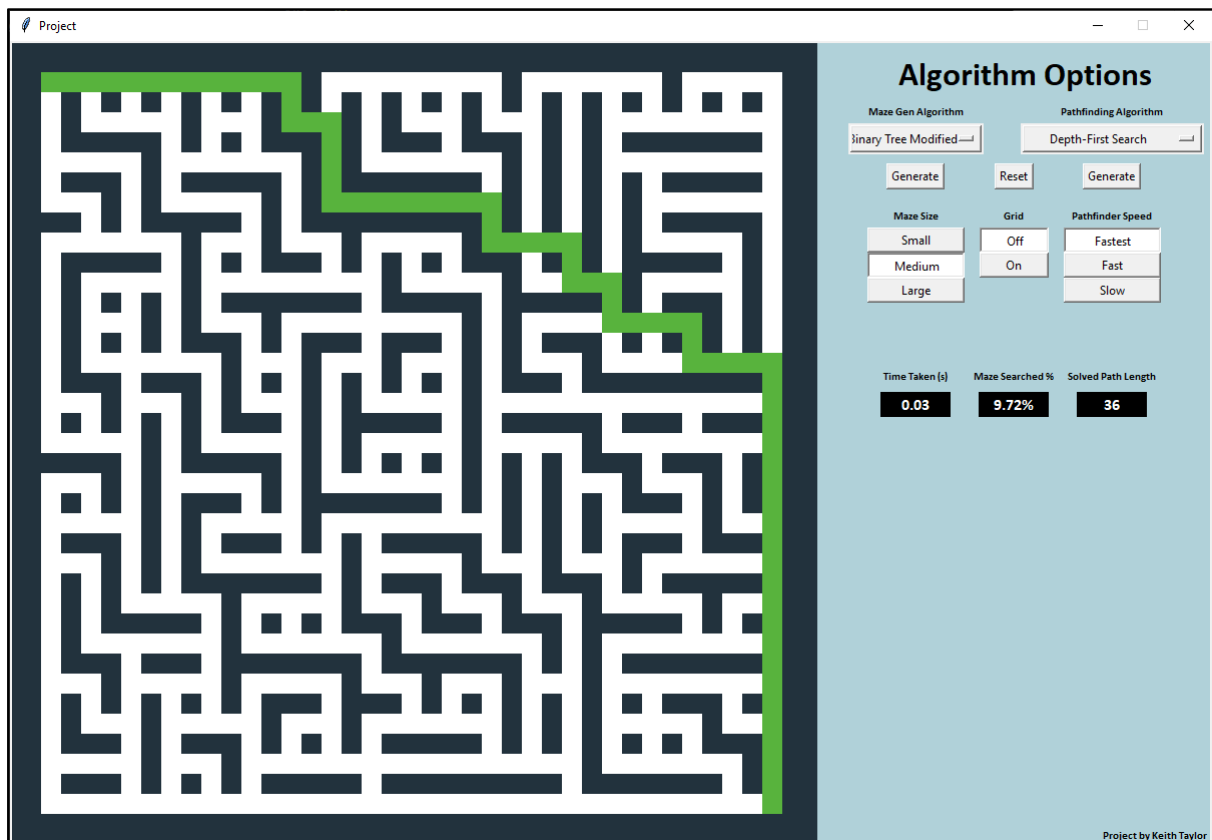


Figure 18 - Modified Binary Tree Maze with favourably biased Depth-First Search Solution

AStarAlgorithm	Purpose	Arguments	Parent Function
<i>pfAStar()</i>	Solve the maze using the A* Algorithm. Will provide different solutions depending on the heuristic choice of the user	canvas(tkinter) mazeSizeCell(int) maze(list) grid(string) speed(float) heuristicChoice(int)	<i>Main.pathFind()</i>
<i>pfHeuristic()</i>	Calculate the heuristic cost of the cell passed via arguments in accordance with the heuristic choice	current(tuple) goal(tuple) heuristicChoice(int)	<i>pfAStar()</i>

Table 8 - AStarAlgorithm class and functions

The A* Algorithm is implemented using only two functions and representing four different algorithms as planned in chapter 3.5. In – the implemented code for *pfHeuristic()* is shown, which can be compared against the pseudocode of Figure 12.

```
def pfHeuristic(current, goal, heuristicChoice):
    if heuristicChoice == 0:    # Dijkstra's
        heuristic = 0
    elif heuristicChoice == 1:  # Euclidean
        heuristic = math.sqrt( ((abs(goal[0] - current[0]) / 2 ) ** 2) + ((abs(goal[1] - current[1]) / 2 )
        ** 2) )
    elif heuristicChoice == 2:  # Manhattan
        heuristic = (abs(goal[0] - current[0]) / 2 + abs(goal[1] - current[1]) / 2 )
    else:                       # Dysfunction
        heuristic = (abs(goal[0] + current[0]) / 2 + abs(goal[1] + current[1]) / 2 )
    return heuristic
```

Figure 19 - Implemented code of *pfHeuristic()*

Structurally the function is the same as planned, with the notable difference being the implementation of the mathematical equations required to calculate the heuristic cost of a given cell. For example, the horizontal distance between the current cell and the goal can be calculated by subtracting the current x coordinate from the x coordinate of the goal, and finding the absolute value using *abs()*. While finding the absolute value isn't necessary, it provides scalability if in future the parameters of the mazes are altered. In this case, the goal may not be at the maximum x coordinate. This is also true for the y coordinate.

Using that distance calculation, it is easy to implement Pythagoras' theorem with *math.sqrt()* for the Euclidean distance, or to calculate the simpler Manhattan distance.

```

# calculates the lowest value cell which hasn't been explored
for coordinate, value in visited.items():
    coordinateTotalValue = value[0] + value[1]
    coordinateHeuristicValue = AStarAlgorithm.pfHeuristic(coordinate, goal, heuristicChoice)
    if visited[coordinate][3] == 0:
        if visitedLowest[0] > coordinateTotalValue:
            visitedLowest = [coordinateTotalValue, coordinateHeuristicValue, coordinate]
        elif visitedLowest[1] > coordinateHeuristicValue:
            visitedLowest = [coordinateTotalValue, coordinateHeuristicValue, coordinate]

current = visitedLowest[2]
visited[current][3] = 1

```

Figure 20 - Code snippet from implemented *pfAStar()*

Figure 20 shows the implementation of the heuristic cost within the algorithmic function. Before a cell's neighbours can be explored and evaluated, the algorithm must first decide which cell needs to be chosen. It does this by calculating the total value, which is stored within the *visited* dictionary. Each coordinate is a key with a corresponding value being equal to a list that stores information relevant to that coordinate, shown in Table 9.

By using a for loop to calculate the heuristic cost of every cell that has been found but not visited the algorithm can decide which cell has the best chance of achieving the heuristic goal; continuing the next sequence of the loop from that cell. This goal is usually to solve the maze in the shortest time, with the only exception being the dysfunctional heuristic designed to be as inefficient as possible.

Element ID	Purpose
0	Actual cost of reaching the current cell (stored as key) from the starting cell.
1	Heuristic calculation of the current cell, found by <i>pfHeuristic()</i> .
2	Origin of the current cell i.e., which cell would be next on the shortest path to the starting cell
3	Boolean data where false means this cell has not been visited before, and true means that it has

Table 9 - Explanation of *visited* dictionary values in *pfAStar()*

5 Testing and Evaluation

5.1 Functional Testing

The program when launched correctly loads the GUI. The various drop-down menus and radio buttons correctly change settings, which then affect how the algorithms run and the visualisations that accompany. The visualisations are accurate to the data on the backend, and when pathfinding algorithms are running they are accurate replications to what can be reproduced by hand. Each algorithm will always find a solution and demonstrate this to the user.

There are however notable bugs and flaws in the functionality which are described below:

- If a pathfinder algorithm is already running and not yet solved, pressing the reset button will cause the current visualisation to disappear. The algorithm however will not stop, and carry on from the instance the button was pressed. This does not affect the data outputs or the solution, however it will hamper a user's ability to perceive visually the efficiency of an algorithm.
- If the user presses the maze generation button without first selecting an algorithm, it will default to Aldous-Broder without communicating this to the user.
- If the user presses the pathfind button without first selecting an algorithm, nothing will happen. While this is intended, it should communicate to the user that an error has occurred and how to fix it.
- If a maze is generated with one grid setting, and then the pathfinder is activated with a different grid setting, the visualisation will be incorrectly aligned. It can still be read correctly; however it does not visualise as intended.
- Breadth-first search and Dijkstra's algorithm should have a similar time complexity in theory, however in practice breadth-first runs much slower than Dijkstra's algorithm.
- If a pathfinding algorithm has run to completion, visualising a solution, and then another pathfinding algorithm is run, it will simply visualise over the pre-existing visualisation. This results in an inaccurate visualisation and would also constitute a memory leak if the user does not reset.

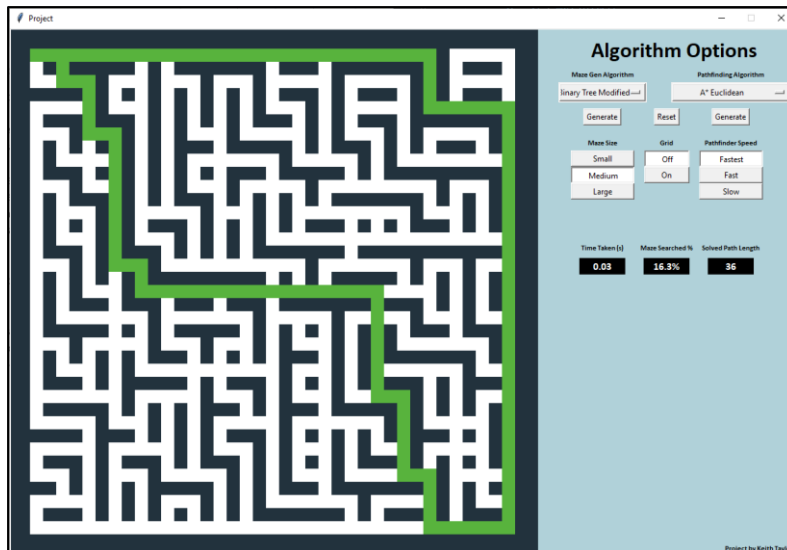


Figure 21 - Instance of a maze with two simultaneously visualised solutions from both Breadth-First Search and A* Euclidean

5.2 Non-Functional Testing

The application uses a simple colouring system as designed in chapter 3.3, with empty cells being white, explored cells being yellow and cells along the solved path being green. Walls are indicated by being a dark blue. The contrast between these colours is enough so that each type of cell is distinct, while they are not distracting.

There are no instructions within the app, nor any information on what any of the algorithms are intended to output. The data outputs are clearly labelled, but the context that gives them importance could be lacking for some users.

The visualisation of the maze is not correctly centred within the visualisation window, which is especially noticeable on large scale mazes.

5.3 Evaluation of Artefact

The artefact achieves the goal set out by the project aims and objectives. It is a tool that presents algorithms with a visualisation leading to a different perspective or improved understanding of the strengths, weaknesses, and general traits of the algorithm. I feel through development of the project, I have developed a deeper understanding of how algorithms are implemented practically and interact with other elements of software.

I think the visualisation is very clear, and the animation is very effective at demonstrating the behaviour of pathfinding algorithms. Every algorithm that has been implemented works as intended, though they do not necessarily have the most optimised time complexity.

My attempt to create my own imperfect maze generating algorithm was a success, as the mazes are reliably solvable and imperfect. They are effective at demonstrating some of the different qualities found in pathfinding algorithms. However, due to my inexperience when designing the algorithm, I underestimated the complexity of an imperfect maze. The modified binary tree algorithm regularly generates mazes which are so imperfect, it is incredibly difficult not to solve them quickly. As seen in Figure 23, the modified algorithm allows pathfinders to find different solutions to demonstrate their qualities, though the mazes are so easy to solve that they make no mistakes. This is an imperfect solution and is something that I would resolve if this project continued. Further research into maze generation could solve this by using pre-existing algorithms that are already fit for purpose. Alternatively, adding further of randomisation in the modified binary tree could allow for mazes that are still imperfect, but not so easy to resolve. If I were to work on this now, I would modify the algorithm to aim to expand around 40% of the dead ends present in the original maze, while leaving 60% of them. This would create smaller pockets of adjacent rooms for mazes to make decisions, while still having a bias to the maze that allows for algorithms to make mistakes and search rooms unnecessarily.

Depth-first search was implemented in an inefficient manner, requiring many more lines of code than necessary. If I were to repeat the task, I would break it down into a function that takes a directional argument, allowing both the functions *pfSame()* and *pfOpposed()* to call this directional argument function 4 times. I estimate this would reduce the footprint of the code by around 60 lines of code. A similar optimisation could be made of the modified binary tree algorithm, avoiding the many repeated calls to append arrays with the same information.

The *neighbourCount()* function in the Tools class would likely be better served recording actual coordinates rather than cardinal directions. This would make for a more complex data structure, as it would store tuples rather than single characters. However, in the current implementation coordinates are converted into directions, and then must be converted back into coordinates for operations.

Replacing the current code with code optimised from the *numPy* package within python. This package includes mathematical formulae implemented as functions, as well as more options to alter arrays with code. I was unaware of this package when implementation began, but found that many programmers online recommend using *numPy* when working with mazes.

Changing the start and end locations of a maze. This could disrupt biases, and would be most interesting on a binary tree maze. They have very clear biases such as an unbroken corridor to the southern and eastern boundaries which funnel any pathfinder into the correct path with few opportunities for errors. However, by reversing this and moving the goal to the north-western portion of the maze with the start to the south-eastern portion, this could include more complexity. Including this would likely not be too difficult, as *Tkinter* can return cell coordinates to the program when objects are clicked. By reverse engineering these coordinates, a mouse click could be converted into the element IDs within an array, and therefore moving the start or goal location.

Some IF/ELSE statements should have been written as MATCH/CASE statements instead. An example of this is in the *mazeGen()* function, as seen in Figure 22. Both IF/ELSE chains within this function would be easier to read if they were written as MATCH/CASE, as they both search for specific user input and reply. This is also the source of the bug causing an unselected maze generation prompt to create an Aldous-Broder maze.

```

# establishes size before calling maze generation algorithm
if choiceSize.get() == "Small":
    mazeSizeWhole = 9
    mazeSizeCell = 80
elif choiceSize.get() == "Medium":
    mazeSizeWhole = 39
    mazeSizeCell = 20
else:
    mazeSizeWhole = 159
    mazeSizeCell = 5
maze = [["w" for i in range(mazeSizeWhole)] for i in range(mazeSizeWhole)]

# calls corresponding maze generation algorithm, followed by visualisation
if choiceMaze.get() == "Binary Tree":
    maze = mg.BinaryTree.mgBT(maze)
elif choiceMaze.get() == "Binary Tree Modified":
    maze = mg.BinaryTreeModified.mgBTM(maze)
else:
    maze = mg.AldousBroder.mgAB(maze)
mazeDrawWhole(maze)

```

Figure 22 – Snippet of code from `mazeGen()` function, demonstrating code that should be replaced with MATCH/CASE

Solving Procedurally Generated Mazes

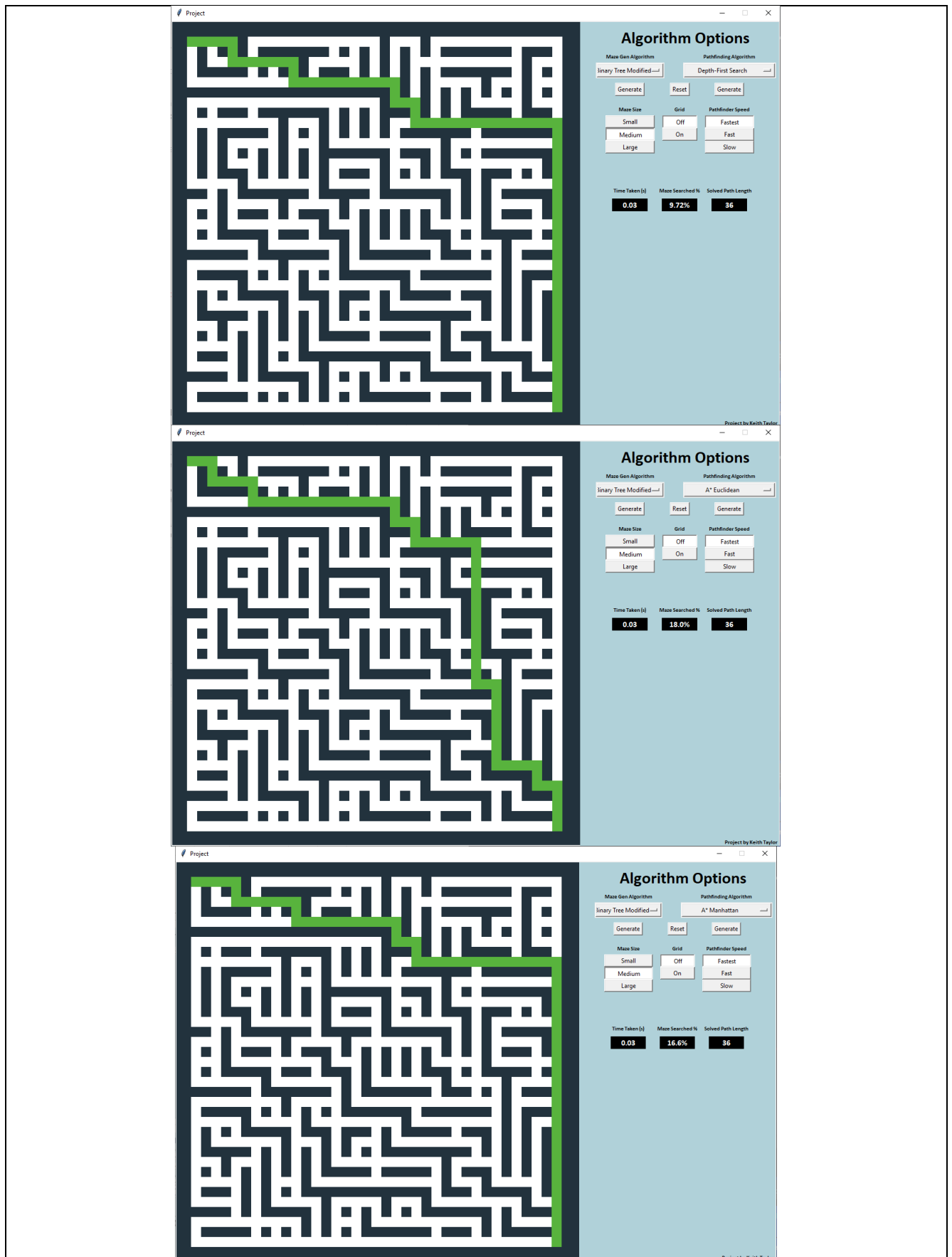


Figure 23 – Modified binary tree maze solved with no yellow cells, i.e. wasted searched, by Depth-First Search (same bias), A* Euclidean and A* Manhattan respectively

6 Summary

6.1 Main Findings

The research presented several claims that can be compared against the objective functionality of this application. This is also true for some of the conclusions I drew from the research.

I claimed that Depth-First Search will often complete faster than Breadth-First Search, though these solutions are not guaranteed to be the shortest pathway. The results shown in Table 10 support this hypothesis, as in each example the depth-first search completed in faster time. In each example, breadth-first search is forced to explore every dead end between the start and end cells, and as such is guaranteed to be slower. Each of the mazes used for this test can be found in Figure 24, Figure 25, and Figure 26 within the Appendices.

	<i>Binary Tree</i>	<i>Modified Binary Tree</i>	<i>Aldous-Broder</i>
<i>Breadth- First Search</i>			
<i>Time Taken(s)</i>	3.59	51.3	21.3
<i>Solved Path Length</i>	156	156	392
<i>Depth- First Search</i>			
<i>Time Taken(s)</i>	0.24	0.24	2.96
<i>Solved Path Length</i>	156	156	392

Table 10 - Results of testing for solution time in Depth-First Search against Breadth-First Search

To demonstrate the potential for longer solutions, Table 11 tested the Depth-First Search with an opposed bias in imperfect mazes generated by the modified binary tree algorithm. These results support the hypothesis that breadth-first search will consistently find a shorter route, but take longer to complete. This is true even when the depth-first search algorithm has an opposed bias to overcome in solving the maze. The mazes used for this test can be found in Figure 27, Figure 28, and Figure 29 within the appendices.

	<i>Test 1</i>	<i>Test 2</i>	<i>Test 3</i>
<i>Breadth- First Search</i>			
<i>Time Taken(s)</i>	49.5	52.2	56.7

<i>Solved Path Length</i>	156	156	156
Depth- First Search			
<i>Time Taken(s)</i>	14.5	15.8	19.0
<i>Solved Path Length</i>	3024	3086	2420

Table 11 - Results of testing for solution lengths in Depth-First Search against Breadth-First Search, in imperfect mazes

I suggested in chapter 2.2 that one of the benefits of implementing A* algorithm is it's code efficiency – the algorithm can perform differently based on a heuristic calculation and would be easy to implement. I believe this is true as it has the shortest amount of lines per-variation as compared to the breadth-first search class and depth-first search class. This is shown in Table 12.

This however should be offset by my own criticism of the implementation of Depth-First Search, which could be programmed more efficiently as described in chapter 5.3.

Pathfinding Class	Total Lines of Code	Variations within class	Lines per Variation
<i>BreadthFirst</i>	53	1	53
<i>DepthFirst</i>	290	2	145
<i>AStarAlgorithm</i>	91	4	23

Table 12 - Demonstration of code lines per algorithm variation

Throughout the development of this program, I discovered that imperfect mazes contain a lot of complexity. The modified binary tree algorithm creates a very imperfect maze that is too easily solvable. To properly demonstrate the behaviour of different pathfinding algorithms, it would be most effective to have a range of imperfect mazes with different traits that can be combined for a more universal perspective. This is described further in chapter 5.3.

6.2 Project Evaluation

I believe the goals of this project were reasonable. Implementation of the various algorithms is a difficult task, but well within my area of expertise and it was a very practical learning experience that I will be able to take forward. However, some of these aims – such as developing a program that assists users in deepening their understanding of algorithms – are intangible and difficult to assess.

Originally I planned to use a survey amongst classmates to assess whether they found it helpful, but I did not account for the time taken to achieve ethical approval for this and so I decided against it.

Time management was the greatest challenge throughout this project for me. I consistently prioritised my module deadlines over working on my project. While this has helped me to attain high grades in those modules, it did cause a lot of stress and resulted in burnout. I believe the main cause of this is my inexperience with managing my recently diagnosed ADHD – module deadlines are much more immediate and novel to work on, and as such it's easier to maintain focus on those. In future to counteract this, I aim set myself tangible and regular deadlines for projects that I am participating in so that I can remain focused.

The main method of overcoming this poor time management was through working longer hours than I should have needed to. My artefact was completed over the course of about 10 days in which I spent 10-12 hours a day working on it. This consistent work meant I completed an artefact that I am proud of, but left me exhausted and unmotivated which resulted in the report being started at the end of march. This is opposed to the original report start time of February, as shown in my project plan (Figure 30).

If I were to continue working on this project, my first goal would be to expand the repertoire of maze generation algorithms. I prioritised the implementation of different pathfinding algorithms because I believed that more algorithms would result in more behaviours to demonstrate. However, it is difficult to demonstrate those behaviours when they are not given a diversity of challenges to complete.

Spending more time researching and implementing existing maze generation algorithms would provide the most immediate benefit to this artefact.

Allowing the user to change the start and end cells to explore mazes with a different directional bias would be another huge benefit to exploring behaviours. As every solution can be found in a south-eastern direction, pathfinders benefit from a bias. By changing the direction of the solution, it would offer a greater opportunity for investigating pathfinding algorithm behaviours.

A further project in its own right would be to redevelop this artefact with the idea of solving mazes with weighted edges. To maintain visual clarity, cells could be coloured according to their weight similar to a heatmap, and the pathfinding algorithms could be represented by a different shape rather than simply recolouring the cell to maintain clear information to the user. To me, this is the next logical step in developing an understanding of pathfinding.

A smaller goal I wished to achieve was learning how to program using Python for the purpose of expanding my range of skills and appearing more employable. I believe this project has given me invaluable experience that I will absolute carry forward in my career, and being comfortable with Python is a huge component of that.

6.3 Conclusion

Over the course of this project, I developed an application that had some bugs but I believe achieved the fundamental objectives set in the original project specification. It visualises pathfinding algorithms effectively atop procedurally generated mazes and I think it is clear and communicative of what is happening on the back end. It also provided plenty of experience and understanding for myself to further develop a program like this, especially as this is the largest software project on which I have worked.

While I did not survey others to estimate the experience that users would have with my software, I can speak to my own experience with the development of the program which I believe is a similar experience to the one I aimed to achieve. By creating this program, I've interacted with algorithms in a way I hadn't previously. I've taken abstract mathematical concepts and given myself a tangible experience to associate with them that has deepened my own understanding. It is my hope that this experience will be replicated by users interacting with the program.

References

- [1] K. Jeong and J. Kim, “Event-Centered Maze Generation Method for Mobile Virtual Reality Applications,” *Symmetry*, vol. 8, no. 11, 2016.
- [2] S. Zlatanova and S. Nedkov, “Enabling obstacle avoidance for Google maps' navigation service,” OTB Research Institute for the Built Environment, 2011.
- [3] K. Moore, K. Jennison and J. Khim, “Brilliant.org,” [Online]. Available: <https://brilliant.org/wiki/depth-first-search-dfs/>. [Accessed 03 04 2023].
- [4] P. H. Kim, *Intelligent Maze Generation*, ProQuest Dissertations Publishing, 2019.
- [5] L. Fredes and J.-F. Marckert, “A combinatorial proof of Aldous–Broder theorem for general Markov chains,” *Random Structures & Algorithms*, vol. 62, no. 2, p. 19, 2022.
- [6] J. Buck, *Mazes for programmers : code your own twisty little passages*, Dallas, Texas: The Pragmatic Bookshelf, 2015.
- [7] V. Bellot, M. Cautrès, J.-M. Favreau, M. Gonzalez-Thauvin, P. Lafourcade, K. Le Cornec, B. Mosnier and S. Rivière-Wekstein, “How to generate perfect mazes?,” *Information Sciences*, vol. 572, no. 444-459, 2021.
- [8] A. Schild, “An almost-linear time algorithm for uniform random spanning tree generation,” *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 214-227, 2018.
- [9] P. Yap, “Grid-Based Path-Finding,” *Lecture Notes in Computer Science*, vol. 2338, pp. 44-55, 2002.
- [10] Y. Björnsson, M. Enzenberger, R. Holte, J. Schaeffer and P. Yap, “Comparison of Different Grid Abstractions for Pathfinding on Maps,” in *IJCAI International Joint Conference on Artificial Intelligence*, 2003.

- [11] S. Edelkamp and S. Schrödl, "Route Planning and Map Inference with Global," *Lecture Notes in Computer Science*, vol. 2598, pp. 128-151, 2003.
- [12] A. Noori and F. Moradi, "Simulation and Comparison of Efficiency in Pathfinding algorithms in Games," *Ciência e Natura*, vol. 37, 2015.
- [13] J. L. Wiener and M. Najork, "Breadth-first crawling yields high-quality pages," in *WWW '01: Proceedings of the 10th international conference on World Wide Web*, 2001.
- [14] A. Bundy and L. Wallen, "Breadth-First Search," in *Catalogue of Artificial Intelligence Tools*, 1986.
- [15] A. Bundy and L. Wallen, "Depth-First Search," in *Catalogue of Artificial Intelligence Tools*, 1986.
- [16] R. Inam, "A* Algorithm for Multicore Graphics Processors," Chalmers University of Technology, 2009.
- [17] J. Xurui and Y. Xiaojun, "Application and Improvement of Heuristic Function in A * Algorithm," in *37th Chinese Control Conference*, 2018.
- [18] R. J. Wilson, "A Brief History of Hamiltonian Graphs," *Annals of Discrete Mathematics*, vol. 41, pp. 487-496, 1988.
- [19] P. Twarie, E. v. Dellen, A. Hillebrand and C. J. Stam, "The minimum spanning tree: An unbiased method for brain network analysis," *NeuroImage*, vol. 104, pp. 177-188, 2015.
- [20] A. A. Z, S. MS and K. H, "A comprehensive study on pathfinding for robotics and video games," Hindawi Publishing Corporation, 2015.
- [21] G. Zhang, S. Cheng, J. Shu, Q. Hu and W. Zheng, "Accelerating breadth-first graph search on a single server by dynamic edge trimming," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 383-394, 2018.

- [22] C. Ntakolia, "A swarm intelligence graph-based pathfinding algorithm (SIGPA) for multi-objective route planning," *Computers & Operations Research*, vol. 133, 2021.
- [23] Python, "tkinter — Python interface to Tcl/Tk," [Online]. Available: <https://docs.python.org/3/library/tkinter.html>. [Accessed 22 January 2023].
- [24] Python, "3.11.3 documentation," Python Software Foundation, 20 4 2023. [Online]. Available: <https://docs.python.org/3.11/>. [Accessed 24 3 2023].
- [25] R. C. Brewster and D. Funk, "On the hamiltonicity of line graphs of locally finite, 6-edge-connected graphs," *Journal of Graph Theory*, vol. 71, no. 2, pp. 182-191, 2011.
- [26] I. Nunes, G. Iacobelli and D. R. Figueredo, "A TRANSIENT EQUIVALENCE BETWEEN ALDOUS-BRODER AND WILSON'S ALGORITHMS AND A TWO-STAGE FRAMEWORK FOR GENERAING UNIFORM SPANNING TREES," Cornell University Library, 2022.
- [27] R. Tarjan, "DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS," *SIAM Journal on Computing*, vol. 1, no. 2, 1972.
- [28] S. Baswana, S. Gupta and A. Tulsyan, "Fault Tolerant Depth First Search in Undirected Graphs: Simple Yet Efficient," *Algorithmica*, vol. 84, 2022.

Appendices

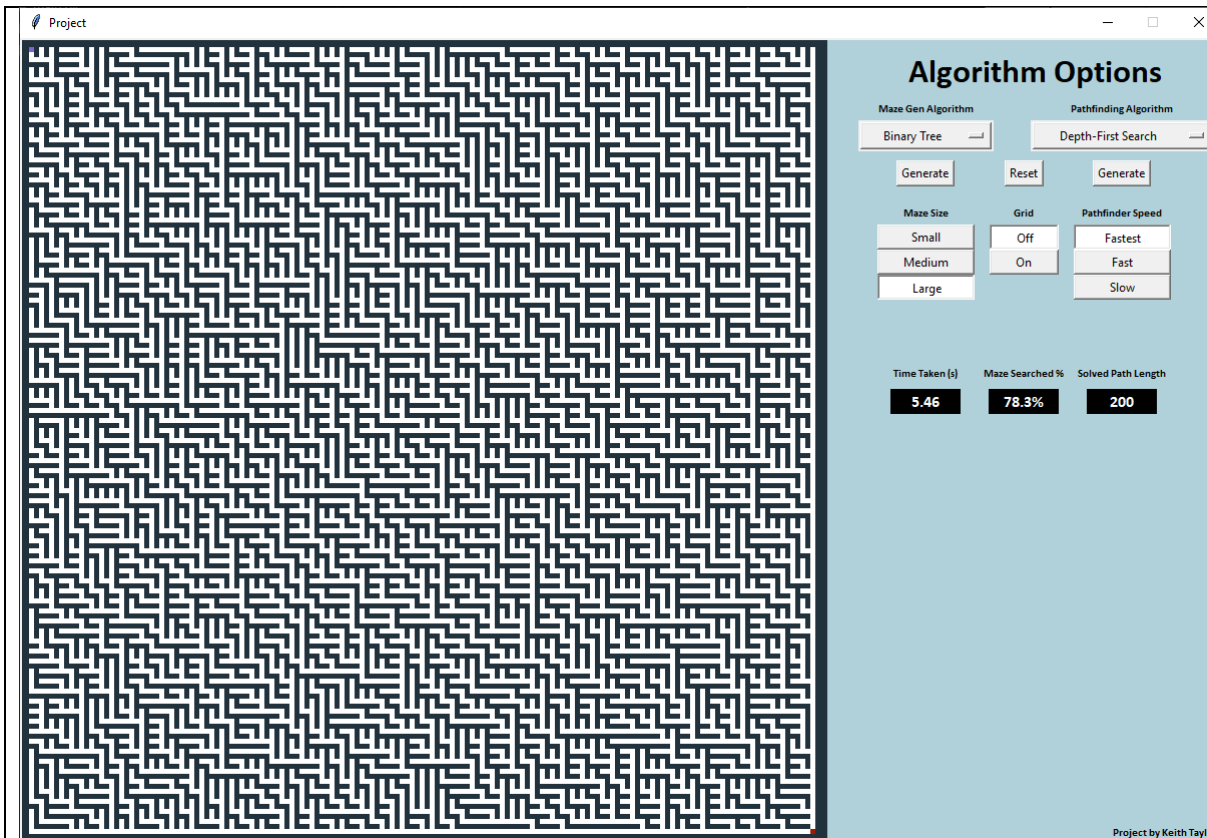


Figure 24 - Binary Tree Maze used for testing in Table 10

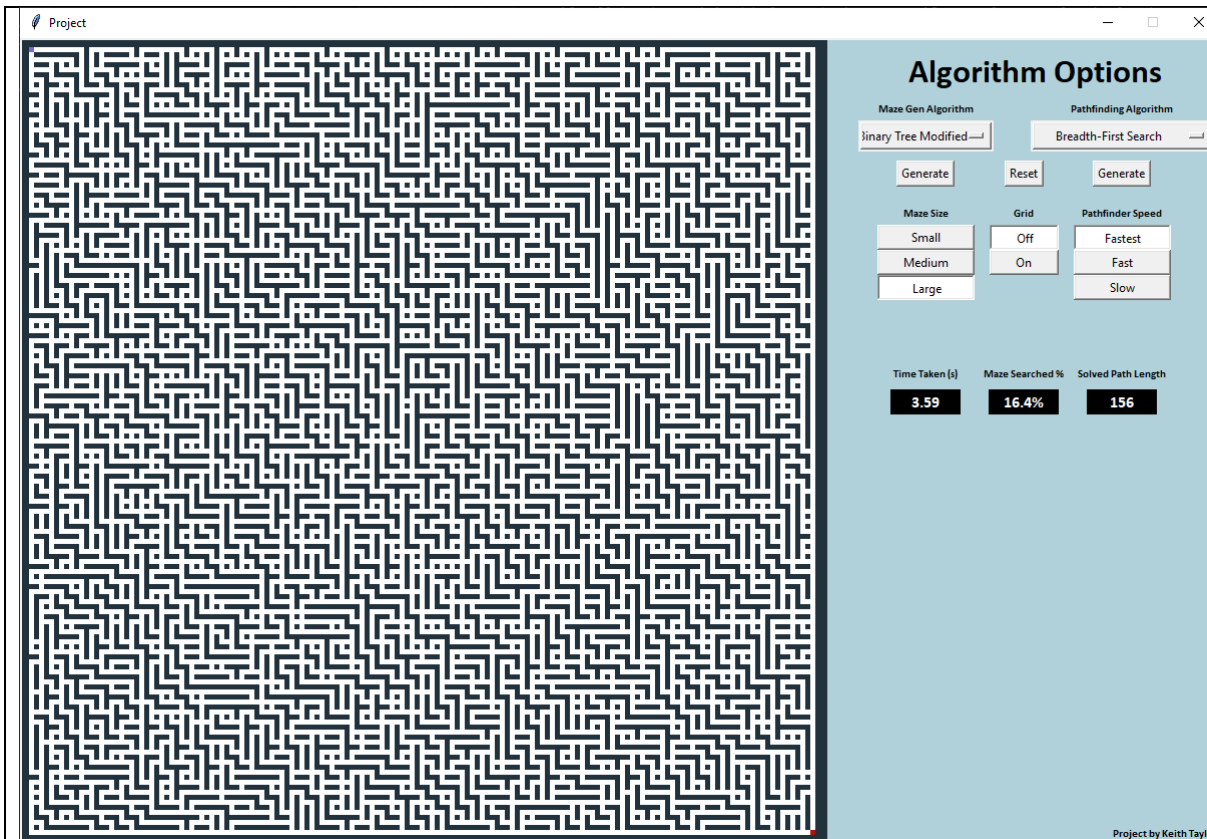


Figure 25 - Modified Binary Tree Maze used for testing in Table 10

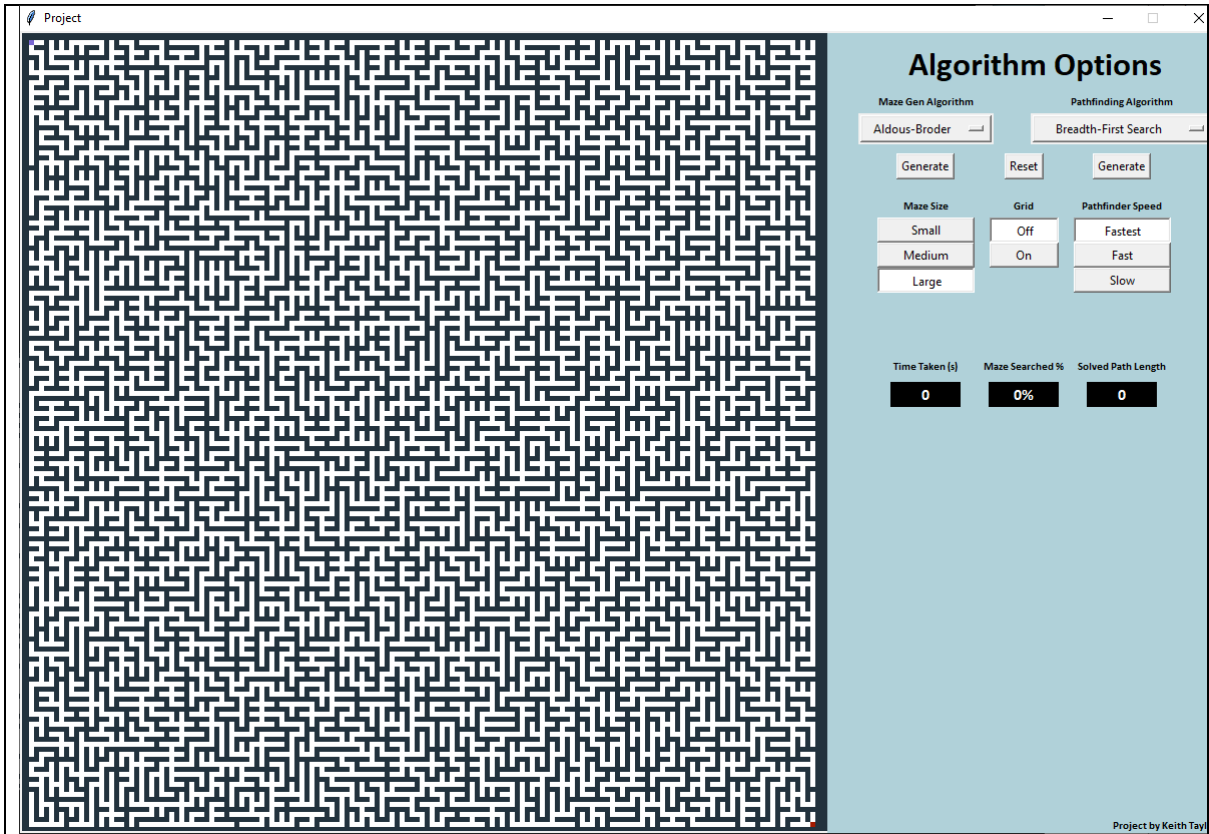


Figure 26 - Aldous-Broder maze used for testing in Table 10

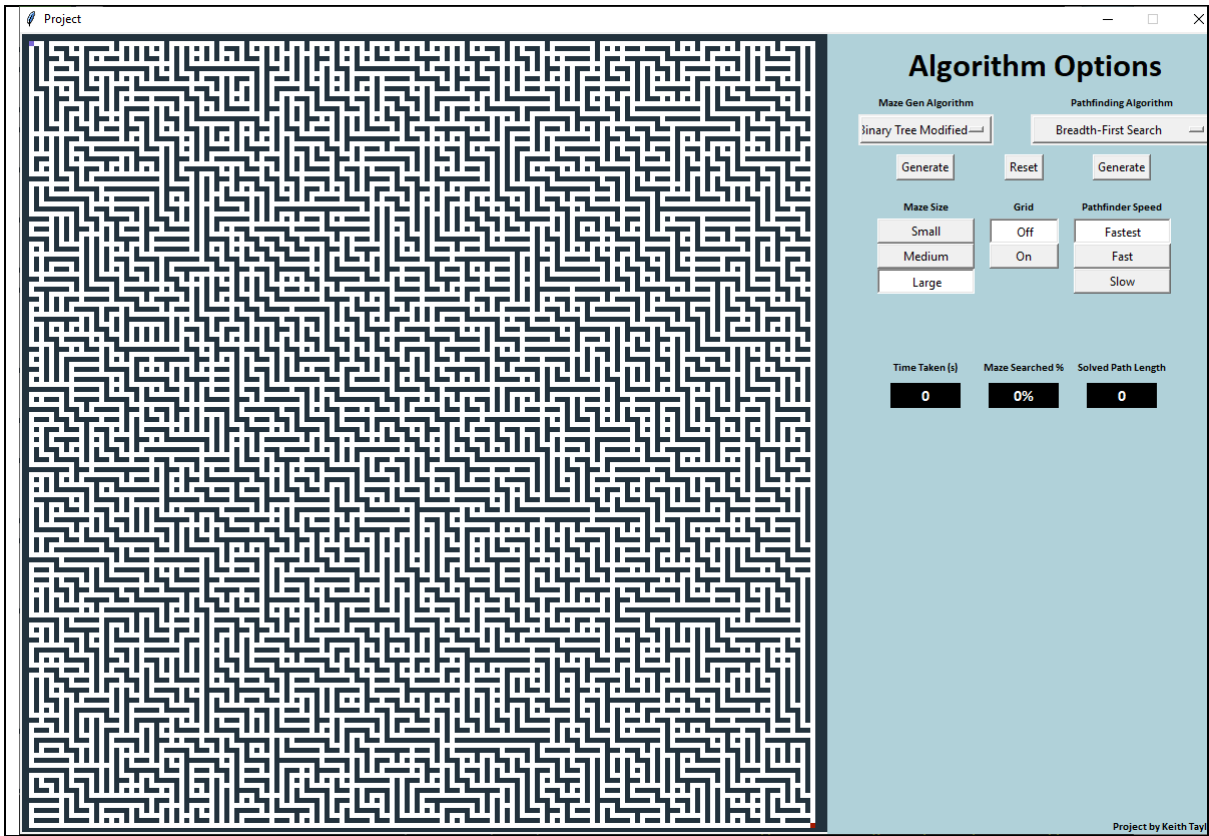


Figure 27 - Modified Binary Tree Maze used for test 1 in Table 11

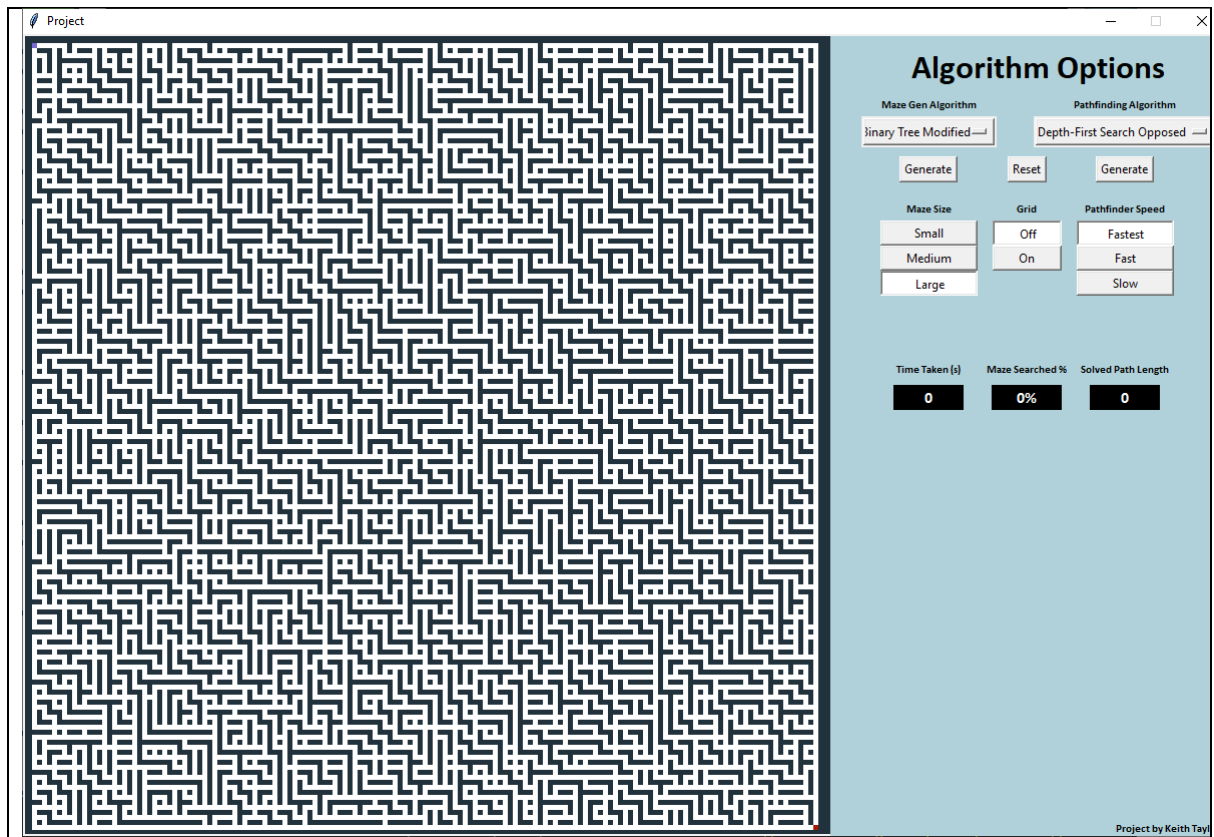


Figure 28 - Modified Binary Tree Maze used for test 2 in Table 11

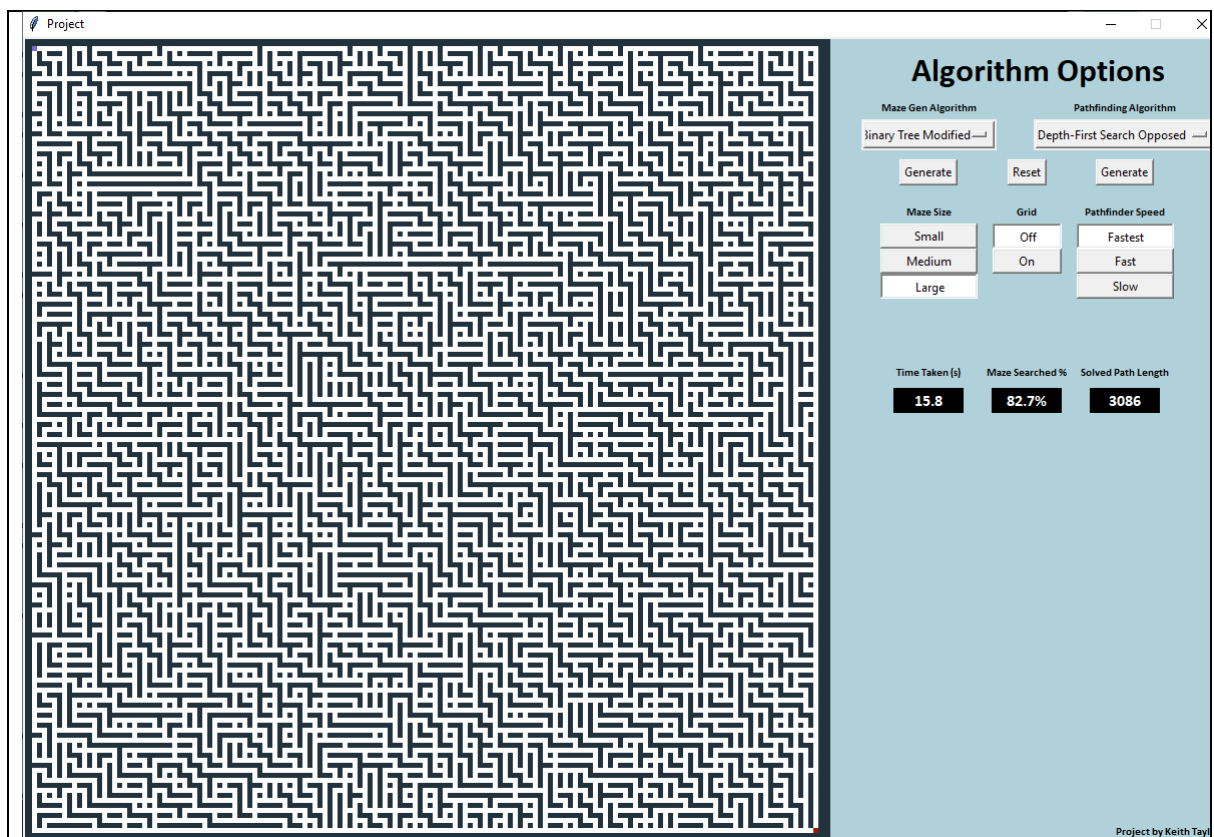


Figure 29 - Modified Binary Tree Maze used for test 3 in Table 11

Task Name	Start	Finish	Duration
Final Year Project	07 October 2022	21 April 2023	
Phase 1 : Research	07 October 2022	28 October 2022	
Research Programming Languages	07 October 2022	14 October 2022	
Research Maze Generation	14 October 2022	21 October 2022	
Research Pathfinding Algorithms	21 October 2022	28 October 2022	
Phase 2: Programming	28 October 2022	27 February 2023	
Complete GUI integration of app	28 October 2022	25 November 2022	
Complete Maze Generation integration of app	25 November 2022	04 January 2023	
Complete Pathfinding Algorithm Integration	04 January 2023	13 February 2023	
Personal Testing	13 February 2023	20 February 2023	
Testing with sample users	20 February 2023	27 February 2023	
Phase 3: Writing	27 February 2023	06 April 2023	
Abstract	27 February 2023	02 March 2023	
Background	02 March 2023	09 March 2023	
Conclusion	09 March 2023	16 March 2023	
Evaluation of Artefact	16 March 2023	23 March 2023	
Evaluation of Project	23 March 2023	30 March 2023	
Preparation for Presentation	30 March 2023	06 April 2023	
Phase 4: Finalise Report	06 April 2023	21 April 2023	

Figure 30 - Initial Project Timeline

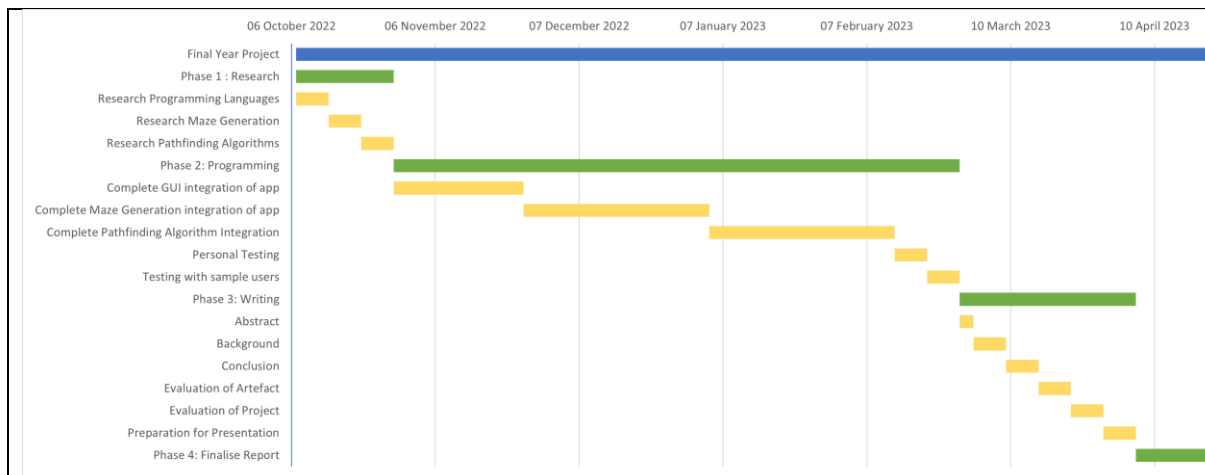


Figure 31 - Initial Project Timeline represented as a Gantt chart

```

import tkinter as tk
import mazeGen as mg
import pathFind as pf
import time

# variables used for the dimensions in the maze
mazeSizeCell = 80 # length of cells in pixels
mazeSizeWhole = 9 # number of rows/columns
maze = [["w" for i in range(mazeSizeWhole)] for i in
range(mazeSizeWhole)] # list of lists to store maze cell
information

# variables used in data outputs for pathfinding evaluation
outputTimeElapsedSeconds = 0 # tracks the time elapsed since
pathfinding began
outputSearchedPercent = "0%" # tracks the percent of maze cells
searched
outputSolvedLength = 0 # counts the number of cells used in the
solved path

# variables used in option menus and radio buttons
optionMaze = ["Binary Tree", "Binary Tree Modified", "Aldous-
Broder"]
optionPath = ["Breadth-First Search", "Depth-First Search",
"Depth-First Search Opposed",
"Dijkstra's Algorithm", "A* Euclidean", "A*
Manhattan", "A* Dysfunctional"]
optionSize = ["Small", "Medium", "Large"]
optionGrid = ["Off", "On"]
optionSpeed = ["Fastest", "Fast", "Slow"]

#####

```

```
#####
###      Functions      ###
#####
#####

# function for creating the visualisation of the entire maze
def mazeDrawWhole(maze):
    canvas.delete("all") # avoid memory leak
    for i, row in enumerate(maze):
        for j, col in enumerate(row):
            if i == 1 and j == 1: # print starting cell
                colour = "#7a68d4"
            elif i == mazeSizeWhole - 2 and j == mazeSizeWhole -
2: # print final cell as red
                colour = "#a31a00"
            elif str(col) == "c": # print clearings as white
                colour = "white"
            elif str(col) == "w": # print walls as black
                colour = "#22323d"
            mazeDrawCell(i, j, colour)

# function for drawing individual cells within the maze
def mazeDrawCell(row, col, colour):
    # cells are drawn 2px larger on both length and width if there
    is no grid
    if choiceGrid.get() == "Off":
        if row == 0 and col == 0:
            canvas.create_rectangle(11, 11, 30, 30,
fill="#22323d", outline="#22323d")
        else:
            x1 = (row * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
            x2 = x1 + mazeSizeCell
```

```

        y1 = (col * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
        y2 = y1 + mazeSizeCell
        canvas.create_rectangle(x1, y1, x2, y2, fill=colour,
width=0)
    else:
        if row == 0 and col == 0:
            canvas.create_rectangle(11, 11, 30, 30,
fill="#22323d", outline="#22323d")
        else:
            x1 = 1 + (row * mazeSizeCell) + (0.5 * mazeSizeCell)
            x2 = x1 + mazeSizeCell
            y1 = 1 + (col * mazeSizeCell) + (0.5 * mazeSizeCell)
            y2 = y1 + mazeSizeCell
            canvas.create_rectangle(x1, y1, x2, y2, fill=colour,
outline="#22323d")

# function that calls maze generation algorithms, called by
mazeGenButton
def mazeGen():
    global maze
    global mazeSizeCell
    global mazeSizeWhole

    # establishes size before calling maze generation algorithm
    match choiceSize.get():
        case "Small":
            mazeSizeWhole = 9
            mazeSizeCell = 80
        case "Medium":
            mazeSizeWhole = 39
            mazeSizeCell = 20
        case "Large":

```

```

        mazeSizeWhole = 159
        mazeSizeCell = 5
        maze = [["w" for i in range(mazeSizeWhole)] for i in
range(mazeSizeWhole)]

        # calls corresponding maze generation algorithm, followed by
visualisation
        if choiceMaze.get() == "Binary Tree":
            maze = mg.BinaryTree.mgBT(maze)
        elif choiceMaze.get() == "Binary Tree Modified":
            maze = mg.BinaryTreeModified.mgBTM(maze)
        else:
            maze = mg.AldousBroder.mgAB(maze)
        mazeDrawWhole(maze)

# function for resetting the pathfinding visualisation while
maintaining the same maze design
def mazeReset():
    canvas.delete("all") # avoid memory leak
    mazeDrawWhole(maze)

    outputVariableSearchedPercent.set("0%")
    outputVariableSolvedLength.set(0)
    outputVariableTimeElapsedSeconds.set(0)

# function for calling the pathfinding functions based on user
choice
def pathFind():
    global outputSearchedPercent
    global outputSolvedLength
    global outputTimeElapsedSeconds

```

```

outputTimeElapsedSeconds = time.time()

# establishes speed before calling pathfinding algorithm
match choiceSpeed.get():
    case "Fast":
        speedPass = 2 / ((mazeSizeWhole + 1) * (mazeSizeWhole
+ 1))
    case "Slow":
        speedPass = 2 / (mazeSizeWhole + 1)
    case "Fastest":
        speedPass = 0

# chained elif statements to call corresponding pathfinding
algorithm

match choicePath.get():
    case "Breadth-First Search":
        outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.BreadthFirst.pfBF(canvas, mazeSizeCell,
maze, choiceGrid.get(), speedPass)
    case "Depth-First Search":
        outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.DepthFirst.pfDF(canvas, mazeSizeCell,
maze, choiceGrid.get(), speedPass, True)
    case "Depth-First Search Opposed":
        outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.DepthFirst.pfDF(canvas, mazeSizeCell,
maze, choiceGrid.get(), speedPass, False)
    case "Dijkstra's Algorithm":
        outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.AStarAlgorithm.pfAStar(canvas,
mazeSizeCell, maze, choiceGrid.get(), speedPass, 0)
    case "A* Euclidean":

```

```

        outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.AStarAlgorithm.pfAStar(canvas,
mazeSizeCell, maze, choiceGrid.get(), speedPass, 1)
        case "A* Manhattan":
            outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.AStarAlgorithm.pfAStar(canvas,
mazeSizeCell, maze, choiceGrid.get(), speedPass, 2)
        case "A* Dysfunctional":
            outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength = pf.AStarAlgorithm.pfAStar(canvas,
mazeSizeCell, maze, choiceGrid.get(), speedPass, 3)

        # modifying string to display correctly in GUI
        outputSearchedPercent = str(outputSearchedPercent * 100 )[0:4]
+ "%"
        outputTimeElapsedSeconds = str(outputTimeElapsedSeconds)[0:4]

        # setting variable strings for GUI display
        outputVariableSearchedPercent.set(outputSearchedPercent)
        outputVariableSolvedLength.set(outputSolvedLength)
        outputVariableTimeElapsedSeconds.set(outputTimeElapsedSeconds)

#####
#####
###      GUI LOOP      ###
#####
#####

#####
## Initialisation and Decoration ##
#####

```



```

# initialise the main window which contains all of the gui/tkinter
elements and begin gui loop
main = tk.Tk()
main.resizable(False, False)
main.title("Project")
main.geometry("1200x800")

# initialise the variable strings required for dropdowns and radio
buttons
choiceMaze = tk.StringVar(main)
choiceMaze.set("Select an Option")
choicePath = tk.StringVar(main)
choicePath.set("Select an Option")
choiceSize = tk.StringVar(main)
choiceSize.set("Small")
choiceGrid = tk.StringVar(main)
choiceGrid.set("Off")
choiceSpeed = tk.StringVar(main)
choiceSpeed.set("Fastest")

#initialise the variable strings required for data outputs
outputVariableTimeElapsedSeconds = tk.StringVar(main)
outputVariableTimeElapsedSeconds.set(outputTimeElapsedSeconds)
outputVariableSearchedPercent = tk.StringVar(main)
outputVariableSearchedPercent.set(outputSearchedPercent)
outputVariableSolvedLength = tk.StringVar(main)
outputVariableSolvedLength.set(outputSolvedLength)

# tkinter objects for window and decoration
canvas = tk.Canvas(main, height=800, width=1200, bg="#22323d")
canvas.pack()
canvasOpt = tk.Frame(main, bg="#b0d1d9")

```

```

canvasOpt.place(x=806, y=2, width=392, height=800)
canvasRadioSize = tk.Frame(canvasOpt)           # Radiobuttons
to change the size
canvasRadioSize.place(relx=0.25, rely=0.23, anchor="n")
canvasRadioGrid = tk.Frame(canvasOpt)           # Radiobuttons
to toggle the grid
canvasRadioGrid.place(relx=0.5, rely=0.23, anchor="n")
canvasRadioSpeed = tk.Frame(canvasOpt)          # Radiobuttons
to change the speed
canvasRadioSpeed.place(relx=0.75, rely=0.23, anchor="n")

# Labels for the right side of the window
mazeTitle = tk.Label(canvasOpt, text="Algorithm Options",
bg="#b0d1d9", font=("Calibri Bold", 25))
mazeTitle.place(relx=0.2, rely=0.01)
credits = tk.Label(canvasOpt, text="Project by Keith Taylor",
bg="#b0d1d9", font=("Calibri Bold", 8))
credits.place(relx=1, rely=1, anchor="se")

#####
## Algorithm Utility ##
#####

# tkinter objects relating to selection and generation of mazes
mazeGenLabel = tk.Label(canvasOpt, text="Maze Gen Algorithm",
bg="#b0d1d9", font=("Calibri Bold", 8))
mazeGenLabel.place(relx=0.25, rely=0.085, anchor="center")
mazeDrop = tk.OptionMenu(canvasOpt, choiceMaze, *optionMaze)
mazeDrop.configure(width=15)
mazeDrop.pack()
mazeDrop.place(relx=0.25, rely=0.1, anchor=tk.N)

```

```

mazeGenButton = tk.Button(canvasOpt, text="Generate",
command=mazeGen)
mazeGenButton.pack()
mazeGenButton.place(relx=0.25, rely=0.15, anchor=tk.N)

# tkinter objects relating to selection and pathfinding of mazes
pathFindLabel = tk.Label(canvasOpt, text="Pathfinding Algorithm",
bg="#b0d1d9", font=("Calibri Bold", 8))
pathFindLabel.place(relx=0.75, rely=0.085, anchor="center")
pathDrop = tk.OptionMenu(canvasOpt, choicePath, *optionPath)
pathDrop.configure(width=23)
pathDrop.pack()
pathDrop.place(relx=0.75, rely=0.1, anchor=tk.N)
pathFindButton = tk.Button(canvasOpt, text="Generate",
command=pathFind)
pathFindButton.pack()
pathFindButton.place(relx=0.75, rely=0.15, anchor=tk.N)

# tkinter object for resetting the current pathfinding
visualisation while maintaining the current maze
mazeResetButton = tk.Button(canvasOpt, text="Reset",
command=mazeReset)
mazeResetButton.pack()
mazeResetButton.place(relx=0.5, rely=0.15, anchor=tk.N)

#####
## Radio Buttons ##
#####

# tkinter objects for changing the size of the maze
mazeGenSizeLabel = tk.Label(canvasOpt, text="Maze Size",
bg="#b0d1d9", font=("Calibri Bold", 8))

```

```

mazeGenSizeLabel.place(relx=0.25, rely=0.215, anchor="center")
for value in optionSize:
    tk.Radiobutton(
        canvasRadioSize, text=value, indicatoron=0, width=7, padx=20, variable=
        choiceSize, value=value).pack(fill="both")

# tkinter objects for toggling the grid of the visualisation
mazeGenGridLabel = tk.Label(canvasOpt, text="Grid", bg="#b0d1d9",
    font=("Calibri Bold", 8))
mazeGenGridLabel.place(relx=0.5, rely=0.215, anchor="center")
for value in optionGrid:
    tk.Radiobutton(canvasRadioGrid, text=value, indicatoron=0, width=
    3, padx=20, variable=choiceGrid, value=value).pack(fill="both")

# tkinter objects for changing the speed of the pathfinding
pathFindSpeedLabel = tk.Label(canvasOpt, text="Pathfinder Speed",
    bg="#b0d1d9", font=("Calibri Bold", 8))
pathFindSpeedLabel.place(relx=0.75, rely=0.215, anchor="center")
for value in optionSpeed:
    tk.Radiobutton(canvasRadioSpeed, text=value, indicatoron=0, width
    =7, padx=20, variable=choiceSpeed, value=value).pack(fill="both")

#####
##  Data Outputs  ##
#####

# tkinter objects for displaying the time taken for the pathfinder
to solver
outputTimeTitle = tk.Label(canvasOpt, text="Time Taken (s)",
    bg="#b0d1d9", font=("Calibri Bold", 8))
outputTimeTitle.place(relx=0.25, rely=0.415, anchor="center")

```

```

outputTimeLabel = tk.Label(canvasOpt,
textvariable=(outputVariableTimeElapsedSeconds), bg="black", fg =
"white", font=("Calibri Bold", 12), height = 1, width =8)
outputTimeLabel.place(relx=0.25, rely=0.45, anchor="center")

# tkinter objects for displaying % of maze searched
outputSearchedTitle = tk.Label(canvasOpt, text="Maze Searched %",
bg="#b0d1d9", font=("Calibri Bold", 8))
outputSearchedTitle.place(relx=0.5, rely=0.415, anchor="center")
outputSearched = tk.Label(canvasOpt,
textvariable=(outputVariableSearchedPercent), bg="black", fg =
"white", font=("Calibri Bold", 12), height = 1, width =8)
outputSearched.place(relx=0.5, rely=0.45, anchor="center")

# tkinter objects for displaying the length of the solved path
outputLengthTitle = tk.Label(canvasOpt, text="Solved Path Length",
bg="#b0d1d9", font=("Calibri Bold", 8))
outputLengthTitle.place(relx=0.75, rely=0.415, anchor="center")
outputLength = tk.Label(canvasOpt,
textvariable=(outputVariableSolvedLength), bg="black", fg =
"white", font=("Calibri Bold", 12, ), height = 1, width =8)
outputLength.place(relx=0.75, rely=0.45, anchor="center")

# close GUI loop
main.mainloop()

```

Figure 32 – Entire code of 'main.py'

```
import random
```

```

from pathFind import Tools

class BinaryTree:
    # function for generating mazes with binary tree algorithm
    def mgBT(maze):
        for i in range(len(maze) - 1):
            if i % 2 == 1:
                for j in range(len(maze) - 1):
                    if j % 2 == 1:
                        # coinflip to decide whether to clear wall
north or east

                        if random.randint(0, 1) == 1:
                            if i == (len(maze) - 2) and j == (
                                len(maze) - 2
                            ): # final square doesn't clear a
wall

                                maze[i][j] = "c"
                            elif 0 < j < (len(maze) - 2):
                                maze[i][j] = "c"
                                maze[i][j + 1] = "c"
                            else: # prevents maze from breaking
the outer wall

                                maze[i][j] = "c"
                                maze[i + 1][j] = "c"

                        else:
                            if i == (len(maze) - 2) and j == (
                                len(maze) - 2
                            ): # final square doesn't clear a
wall

                                maze[i][j] = "c"
                            elif 0 < i < (len(maze) - 2):

```

```

        maze[i][j] = "c"
        maze[i + 1][j] = "c"
    else: # prevents maze from breaking
the outer wall

        maze[i][j] = "c"
        maze[i][j + 1] = "c"

    return maze

class BinaryTreeModified:
    # function for generating modified binary tree mazes
    #    first calls BinaryTree.mgBT to generate a maze, and then
    #    edits the maze
    #    by searching for dead ends and adding new pathways to
    #    unconnected passages
    #    this creates an imperfect maze with many passageways
    def mgBTM(maze):
        maze = BinaryTree.mgBT(maze)
        neighbours = Tools.neighbourCount(maze)

        for i, value in neighbours.items():
            # first searches for cells with only one neighbour
i.e. dead ends
            if len(value) == 1:
                # first if statement used to ensure top left
corner doesn't create a passageway outside the maze
                if i[0] == 1 and i[1] == 1:
                    if value[0] == "E":
                        neighbours[(i)].append("S")
                        maze[i[0]][i[1] + 1] = "c"
                    else:
                        neighbours[(i)].append("E")
                        maze[i[0] + 1][i[1]] = "c"

```

```

        # prevents northern wall from creating northern
passageways

        elif i[0] == 1:
            if value[0] == "E":
                if random.randint(0, 1) == 1 and i[1] !=
len(maze) - 2:
                    neighbours[(i)].append("S")
                    maze[i[0]][i[1] + 1] = "c"
                elif i[1] != 1:
                    neighbours[(i)].append("W")
                    maze[i[0]][i[1] - 1] = "c"
            else:
                if random.randint(0, 1) == 1 and i[1] !=
len(maze) - 2:
                    neighbours[(i)].append("E")
                    maze[i[0] + 1][i[1]] = "c"
                elif i[1] != 1:
                    neighbours[(i)].append("W")
                    maze[i[0]][i[1] - 1] = "c"

        # prevents western wall from creating western
passageways

        elif i[1] == 1:
            if value[0] == "E":
                if random.randint(0, 1) == 1 and i[0] !=
len(maze) - 2:
                    neighbours[(i)].append("S")
                    maze[i[0]][i[1] + 1] = "c"
                elif i[0] != 1:
                    neighbours[(i)].append("N")
                    maze[i[0] - 1][i[1]] = "c"
            else:

```



```

        if random.randint(0, 1) == 1 and i[0] !=
len(maze) - 2:
            neighbours[(i)].append("E")
            maze[i[0] + 1][i[1]] = "c"
        elif i[0] != 1:
            neighbours[(i)].append("N")
            maze[i[0] - 1][i[1]] = "c"

        # any cells that are not along the northern or
western wall
        else:
            if value[0] == "E":
                randomCheck = random.randint(0, 2)
                if randomCheck == 0 and i[1] != len(maze)
- 2:
                    neighbours[(i)].append("S")
                    maze[i[0]][i[1] + 1] = "c"
                elif randomCheck == 1:
                    neighbours[(i)].append("W")
                    maze[i[0]][i[1] - 1] = "c"
                else:
                    neighbours[(i)].append("N")
                    maze[i[0] - 1][i[1]] = "c"
            elif value[0] == "W":
                randomCheck = random.randint(0, 2)
                if randomCheck == 0 and i[1] != len(maze)
- 2:
                    neighbours[(i)].append("S")
                    maze[i[0]][i[1] + 1] = "c"
                elif randomCheck == 1:
                    neighbours[(i)].append("E")
                    maze[i[0] + 1][i[1]] = "c"
                else:

```

```

        neighbours[(i)].append("N")
        maze[i[0] - 1][i[1]] = "c"
    elif value[0] == "N":
        randomCheck = random.randint(0, 2)
        if randomCheck == 0 and i[1] != len(maze)
- 2:
            neighbours[(i)].append("S")
            maze[i[0]][i[1] + 1] = "c"
            elif randomCheck == 1:
                neighbours[(i)].append("E")
                maze[i[0] + 1][i[1]] = "c"
            else:
                neighbours[(i)].append("W")
                maze[i[0]][i[1] - 1] = "c"
        else:
            randomCheck = random.randint(0, 2)
            if randomCheck == 0 and i[1] != len(maze)
- 2:
                neighbours[(i)].append("N")
                maze[i[0] - 1][i[1]] = "c"
            elif randomCheck == 1:
                neighbours[(i)].append("E")
                maze[i[0] + 1][i[1]] = "c"
            else:
                neighbours[(i)].append("W")
                maze[i[0]][i[1] - 1] = "c"

    return maze

class AldousBroder:
    # function for generating mazes with the Aldous-Broder
    algorithm

```

```

def mgAB(maze):
    visited = {(1, 1): ""}
    x = 1
    y = 1
    # while loop runs until every cell has been visited once
    while len(visited) < ((len(maze) - 1) / 2) * ((len(maze) -
1) / 2):
        maze[x][y] = "c"
        randomCheck = random.randint(0, 3)
        # next cell is chosen at random by randomCheck
        # with a second condition to prevent breaking
outside the boundary
        if x != (len(maze) - 2) and randomCheck == 0:
            # nested within each if statement is the
possibility of adding new cells
            # to the visited dictionary, if it has not been
seen
            if ((x + 2, y)) not in visited:
                maze[x + 1][y] = "c"
                maze[x + 2][y] = "c"
                visited[x + 2, y] = ""
            # else the algorithm simply moves to that cell and
repeats the cycle
            x += 2

        elif y != (len(maze) - 2) and randomCheck == 1:
            if ((x, y + 2)) not in visited:
                maze[x][y + 1] = "c"
                maze[x][y + 2] = "c"
                visited[x, y + 2] = ""
            y += 2

        elif x != 1 and randomCheck == 2:

```

```
        if ((x - 2, y)) not in visited:
            maze[x - 1][y] = "c"
            maze[x - 2][y] = "c"
            visited[x - 2, y] = ""
        x += -2

    elif y != 1 and randomCheck == 3:
        if ((x, y - 2)) not in visited:
            maze[x][y - 1] = "c"
            maze[x][y - 2] = "c"
            visited[x, y - 2] = ""
        y += -2

    return maze
```

Figure 33 – Entire code of 'mazeGen.py'

```

import tkinter as tk
import time
import sys
import math

class Tools: # class for tools used by pathfinders and mazeGen
    # function for drawing individual cells of the calculation in
    progress
    def drawCalculation(canvas, mazeSizeCell, row, col, grid):
        if grid == "On":
            x1 = (row * mazeSizeCell) + (0.5 * mazeSizeCell) + 2
            x2 = x1 + mazeSizeCell - 1
            y1 = (col * mazeSizeCell) + (0.5 * mazeSizeCell) + 2
            y2 = y1 + mazeSizeCell - 1
            if row != 1 or col != 1:
                canvas.create_rectangle(x1, y1, x2, y2,
fill="#ffca45", width=0)
        else:
            x1 = (row * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
            x2 = x1 + mazeSizeCell
            y1 = (col * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
            y2 = y1 + mazeSizeCell
            if row != 1 or col != 1:
                canvas.create_rectangle(x1, y1, x2, y2,
fill="#ffca45", width=0)
            canvas.update()

    # function for drawing individual cells of the solution
    # first checks if Grid is an option, as it modifies the
    visualisation
    def drawSolutionCell(canvas, mazeSizeCell, row, col, grid):
        if grid == "On":

```

```

        x1 = (row * mazeSizeCell) + (0.5 * mazeSizeCell) + 2
        x2 = x1 + mazeSizeCell - 1
        y1 = (col * mazeSizeCell) + (0.5 * mazeSizeCell) + 2
        y2 = y1 + mazeSizeCell - 1
        canvas.create_rectangle(x1, y1, x2, y2,
fill="#58b33d", width=0)
    else:
        x1 = (row * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
        x2 = x1 + mazeSizeCell
        y1 = (col * mazeSizeCell) + (0.5 * mazeSizeCell) + 1
        y2 = y1 + mazeSizeCell
        canvas.create_rectangle(x1, y1, x2, y2,
fill="#58b33d", width=0)
    canvas.update()

    # function for creating the solution visualisation by
    recursively calling drawSolutionCell
    # uses time.sleep to slow down the visualisation based on
    the user's chosen speed
    def drawSolutionWhole(canvas, mazeSizeCell, solvedPath, grid,
speed):
        for i in range(len(solvedPath) - 1):
            time.sleep(speed / 4)
            Tools.drawSolutionCell(canvas, mazeSizeCell,
solvedPath[i][0], solvedPath[i][1], grid)
            time.sleep(speed / 4)
            if solvedPath[i][0] == solvedPath[i + 1][0]:
                j = (int(solvedPath[i][1]) + int(solvedPath[i +
1][1])) / 2
                Tools.drawSolutionCell(canvas, mazeSizeCell,
solvedPath[i][0], j, grid)
            elif solvedPath[i][1] == solvedPath[i + 1][1]:

```

```

        j = (int(solvedPath[i][0]) + int(solvedPath[i +
1][0])) / 2

        Tools.drawSolutionCell(canvas, mazeSizeCell, j,
solvedPath[i][1], grid)

        canvas.update()

        Tools.drawSolutionCell(canvas, mazeSizeCell, 1, 1, grid)

    # function for counting up neighbours of every given cell in a
maze to store in a dictionary
    def neighbourCount(maze):
        neighbours = {} # dictionary storing cells as key and
neighbour directions as associated values

        for i in range(len(maze) - 1):
            if i % 2 == 1:
                for j in range(len(maze) - 1):
                    if j % 2 == 1:
                        if maze[i][j] == "c":
                            neighbours[(i, j)] = []
                            if maze[i + 1][j] == "c":
                                neighbours[(i, j)].append("E")
                            if maze[i][j + 1] == "c":
                                neighbours[(i, j)].append("S")
                            if maze[i - 1][j] == "c":
                                neighbours[(i, j)].append("W")
                            if maze[i][j - 1] == "c":
                                neighbours[(i, j)].append("N")

        return neighbours

#class to store the function used by Breadth-First Search
class BreadthFirst:
    # function for solving mazes with breadth first search
    def pfbf(canvas, mazeSizeCell, maze, grid, speed):
        visited = [(1, 1)] # list to store all visted cells

```

```

        queue = [(1, 1)] # list to store cells yet to run through
the algorithm
        pathways = {} # dictionary storing every cell's origin
during exploration
        solvedPath = [] # list storing every cell in the solved
path
        neighbours = Tools.neighbourCount(maze)
        goal = ((len(maze) - 2), (len(maze) - 2)) # sets goal
equal to the last cell of the maze
        outputTimeElapsedSeconds = time.time() # records start
time

        # will run until every explorable cell has been seen
        #     each loop removes one cell from the queue and
explores every neighbour
        while len(queue) > 0:
            time.sleep(speed) # time.sleep used to slow down
visualisation
            current = queue.pop(0)
            if current == ((len(maze) - 2), (len(maze) - 2)):
                break # used to break out of the while loop if the
goal has been found
            for d in "ESWN":
                if d in neighbours[current]:
                    if d == "E":
                        nextCell = (current[0] + 2, current[1])
                        nextPath = (current[0] + 1, current[1])
                    elif d == "S":
                        nextCell = (current[0], current[1] + 2)
                        nextPath = (current[0], current[1] + 1)
                    elif d == "W":
                        nextCell = (current[0] - 2, current[1])
                        nextPath = (current[0] - 1, current[1])

```



```

        elif d == "N":
            nextCell = (current[0], current[1] - 2)
            nextPath = (current[0], current[1] - 1)

            # draws explored cells for each given loop
            Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
            Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)

            if nextCell not in visited:
                visited.append(nextCell)
                queue.append(nextCell)
                pathways[nextCell] = current
while goal != (1, 1):
    solvedPath.append(goal)
    goal = pathways[goal]
solvedPath.append((1, 1))

    outputTimeElapsedSeconds = time.time() -
outputTimeElapsedSeconds
    Tools.drawSolutionWhole(canvas, mazeSizeCell, solvedPath,
grid, speed)

    outputSearchedPercent = (len(visited) - 1) / ( (
((len(maze) - 1) / 2) * ((len(maze) - 1) / 2) ) - 1 )
    outputSolvedLength = len(solvedPath) - 1

    return outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength

#class to store all 3 functions used by depth first search
class DepthFirst:

```

```

# function for completing the majority of the depth-first
search with the same bias as the maze
def pfSame(
    goal,
    current,
    neighbours,
    canvas,
    mazeSizeCell,
    maze,
    grid,
    speed,
    solvedPath,
    visited,
):
    breakOut = False
    if "E" in neighbours[current] and breakOut != True:
        nextCell = (current[0] + 2, current[1])
        nextPath = (current[0] + 1, current[1])
        neighbours[current].remove("E")
        time.sleep(speed)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
        if nextCell != goal and nextCell not in visited:
            neighbours[nextCell].remove("W")
            visited[nextCell] = ""
            goal, breakOut, visited = DepthFirst.pfSame(
                goal,
                nextCell,
                neighbours,
                canvas,
                mazeSizeCell,

```

```

        maze,
        grid,
        speed,
        solvedPath,
        visited,
    )
    if nextCell == goal:
        solvedPath.append(goal)
        goal = current
        return goal, True, visited
    if "S" in neighbours[current] and breakOut != True:
        nextCell = (current[0], current[1] + 2)
        nextPath = (current[0], current[1] + 1)
        neighbours[current].remove("S")
        time.sleep(speed)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
        if nextCell != goal and nextCell not in visited:
            neighbours[nextCell].remove("N")
            visited[nextCell] = ""
            goal, breakOut, visited = DepthFirst.pfSame(
                goal,
                nextCell,
                neighbours,
                canvas,
                mazeSizeCell,
                maze,
                grid,
                speed,
                solvedPath,
                visited,

```

```

    )

    if nextCell == goal:
        solvedPath.append(goal)
        goal = current
        return goal, True, visited

    if "W" in neighbours[current] and breakOut != True:
        nextCell = (current[0] - 2, current[1])
        nextPath = (current[0] - 1, current[1])
        neighbours[current].remove("W")
        time.sleep(speed)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
        if nextCell != goal and nextCell not in visited:
            neighbours[nextCell].remove("E")
            visited[nextCell] = ""
            goal, breakOut, visited = DepthFirst.pfSame(
                goal,
                nextCell,
                neighbours,
                canvas,
                mazeSizeCell,
                maze,
                grid,
                speed,
                solvedPath,
                visited,
            )

    if nextCell == goal:
        solvedPath.append(goal)
        goal = current
        return goal, True, visited

```

```

        if "N" in neighbours[current] and breakOut != True:
            nextCell = (current[0], current[1] - 2)
            nextPath = (current[0], current[1] - 1)
            neighbours[current].remove("N")
            time.sleep(speed)
            Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
            Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
            if nextCell != goal and nextCell not in visited:
                neighbours[nextCell].remove("S")
                visited[nextCell] = ""
                goal, breakOut, visited = DepthFirst.pfSame(
                    goal,
                    nextCell,
                    neighbours,
                    canvas,
                    mazeSizeCell,
                    maze,
                    grid,
                    speed,
                    solvedPath,
                    visited,
                )
            if nextCell == goal:
                solvedPath.append(goal)
                goal = current
                return goal, True, visited
        return goal, False, visited

# function used to perform depth first search with an opposed
bias
def pfOpposed(

```

```

        goal,
        current,
        neighbours,
        canvas,
        mazeSizeCell,
        maze,
        grid,
        speed,
        solvedPath,
        visited,
    ):
        breakOut = False
        if "N" in neighbours[current] and breakOut != True:
            nextCell = (current[0], current[1] - 2)
            nextPath = (current[0], current[1] - 1)
            neighbours[current].remove("N")
            time.sleep(speed)
            Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
            Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
            if nextCell != goal and nextCell not in visited:
                neighbours[nextCell].remove("S")
                visited[nextCell] = ""
                goal, breakOut, visited = DepthFirst.pfOpposed(
                    goal,
                    nextCell,
                    neighbours,
                    canvas,
                    mazeSizeCell,
                    maze,
                    grid,
                    speed,

```

```

        solvedPath,
        visited,
    )
    if nextCell == goal:
        solvedPath.append(goal)
        goal = current
        return goal, True, visited
    if "W" in neighbours[current] and breakOut != True:
        nextCell = (current[0] - 2, current[1])
        nextPath = (current[0] - 1, current[1])
        neighbours[current].remove("W")
        time.sleep(speed)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
        if nextCell != goal and nextCell not in visited:
            neighbours[nextCell].remove("E")
            visited[nextCell] = ""
            goal, breakOut, visited = DepthFirst.pfOpposed(
                goal,
                nextCell,
                neighbours,
                canvas,
                mazeSizeCell,
                maze,
                grid,
                speed,
                solvedPath,
                visited,
            )
    if nextCell == goal:
        solvedPath.append(goal)

```

```

        goal = current
        return goal, True, visited
    if "S" in neighbours[current] and breakOut != True:
        nextCell = (current[0], current[1] + 2)
        nextPath = (current[0], current[1] + 1)
        neighbours[current].remove("S")
        time.sleep(speed)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
        if nextCell != goal and nextCell not in visited:
            neighbours[nextCell].remove("N")
            visited[nextCell] = ""
            goal, breakOut, visited = DepthFirst.pfOpposed(
                goal,
                nextCell,
                neighbours,
                canvas,
                mazeSizeCell,
                maze,
                grid,
                speed,
                solvedPath,
                visited,
            )
        if nextCell == goal:
            solvedPath.append(goal)
            goal = current
            return goal, True, visited
    if "E" in neighbours[current] and breakOut != True:
        nextCell = (current[0] + 2, current[1])
        nextPath = (current[0] + 1, current[1])

```



```

        neighbours[current].remove("E")
        time.sleep(speed)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextPath[0], nextPath[1], grid)
        Tools.drawCalculation(canvas, mazeSizeCell,
nextCell[0], nextCell[1], grid)
        if nextCell != goal and nextCell not in visited:
            neighbours[nextCell].remove("W")
            visited[nextCell] = ""
            goal, breakOut, visited = DepthFirst.pfOpposed(
                goal,
                nextCell,
                neighbours,
                canvas,
                mazeSizeCell,
                maze,
                grid,
                speed,
                solvedPath,
                visited,
            )
        if nextCell == goal:
            solvedPath.append(goal)
            goal = current
            return goal, True, visited
    return goal, False, visited

# function called by main class. Initialises variables used by
both versions of depth first
# and then passes that information to the corresponding
algorithm.
# Is also responsible for visualisation
def pfDF(canvas, mazeSizeCell, maze, grid, speed, bias):

```

```

visited = {(1, 1): ""}
solvedPath = []
neighbours = Tools.neighbourCount(maze)
current = (1, 1)
goal = ((len(maze) - 2), (len(maze) - 2))
outputTimeElapsedSeconds = time.time()

    sys.setrecursionlimit(6500) # required to prevent overflow
of recursion limit

    if bias == True:
        DepthFirst.pfSame(
            goal,
            current,
            neighbours,
            canvas,
            mazeSizeCell,
            maze,
            grid,
            speed,
            solvedPath,
            visited,
        )
    else:
        DepthFirst.pfOpposed(
            goal,
            current,
            neighbours,
            canvas,
            mazeSizeCell,
            maze,
            grid,
            speed,

```

```

        solvedPath,
        visited,
    )

    solvedPath.append((1, 1))

    outputTimeElapsedSeconds = time.time() -
outputTimeElapsedSeconds
    Tools.drawSolutionWhole(canvas, mazeSizeCell, solvedPath,
grid, speed)

    outputSearchedPercent = (len(visited) - 1) / ( (
((len(maze) - 1) / 2) * ((len(maze) - 1) / 2) ) - 1 )
    outputSolvedLength = len(solvedPath) - 1

    return outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength

#class to store the 2 functions used by A* algorithm
class AStarAlgorithm:
    # function to calculate the heuristic for any given cell
    def pfHeuristic(current, goal, heuristicChoice):
        if heuristicChoice == 0:    # Dijkstra's
            heuristic = 0
        elif heuristicChoice == 1:  # Euclidean
            heuristic = math.sqrt( ((abs(goal[0] - current[0]) / 2
) ** 2) + ((abs(goal[1] - current[1]) / 2 ) ** 2) )
        elif heuristicChoice == 2: # Manhattan
            heuristic = (abs(goal[0] - current[0]) / 2 +
abs(goal[1] - current[1]) / 2 )
        else:                       # Dysfunction
            heuristic = (abs(goal[0] + current[0]) / 2 +
abs(goal[1] + current[1]) / 2 )

```

```

    return heuristic

# function for performing the bulk of the a* algorithm
def pfAStar(canvas, mazeSizeCell, maze, grid, speed,
heuristicChoice):

    # initialisation of standard
    solvedPath = []
    neighbours = Tools.neighbourCount(maze)
    current = (1, 1)
    goal = ((len(maze) - 2), (len(maze) - 2))
    heuristic = AStarAlgorithm.pfHeuristic(current, goal,
heuristicChoice)

    # dict to store cells as key, followed by their actual
value, their heuristic value, their origin cell
    # and if they have been visited already
    visited = {(1,1) : [0, heuristic, (0,0), 1]}
    outputTimeElapsedSeconds = time.time()

    while current != goal:

        visitedLowest = [2048]
        time.sleep(speed) # time.sleep used to slow down
visualisation
        if current == goal:
            break # used to break out of the while loop if the
goal has been found
        currentActualValue = 0
        currentSource = current
        while visited[currentSource][2] != (0,0):
            currentActualValue += 1

```

```

        currentSource = visited[currentSource][2]
        for d in "ESWN": #for loop that scouts values for
neighbouring cells
            if d in neighbours[current]:
                if d == "E":
                    nextCell = (current[0] + 2, current[1])
                elif d == "S":
                    nextCell = (current[0], current[1] + 2)
                elif d == "W":
                    nextCell = (current[0] - 2, current[1])
                elif d == "N":
                    nextCell = (current[0], current[1] - 2)

                if nextCell not in visited:
                    # if cell has not been visited before,
adds information to the dict
                    nextCellHeuristic =
AStarAlgorithm.pfHeuristic(nextCell, goal, heuristicChoice)
                    visited[nextCell] = [currentActualValue +
1, nextCellHeuristic, current, 0]
                elif nextCell in visited and
visited[nextCell][0] > currentActualValue:
                    # if cell has been visited before, checks
to see if a new lowest value is found
                    visited[nextCell][0] = currentActualValue
+ 1

                # calculates the lowest value cell which hasn't been
explored
                for coordinate, value in visited.items():
                    coordinateTotalValue = value[0] + value[1]
                    coordinateHeuristicValue =
AStarAlgorithm.pfHeuristic(coordinate, goal, heuristicChoice)

```

```

        if visited[coordinate][3] == 0:
            if visitedLowest[0] > coordinateTotalValue:
                visitedLowest = [coordinateTotalValue,
coordinateHeuristicValue, coordinate]
            elif visitedLowest[1] >
coordinateHeuristicValue:
                visitedLowest = [coordinateTotalValue,
coordinateHeuristicValue, coordinate]

        current = visitedLowest[2]
        visited[current][3] = 1

        if current[0] + 2 == visited[current][2][0]:
            nextCell = (current[0] + 2, current[1])
            nextPath = (current[0] + 1, current[1])
        elif current[1] + 2 == visited[current][2][1]:
            nextCell = (current[0], current[1] + 2)
            nextPath = (current[0], current[1] + 1)
        elif current[0] - 2 == visited[current][2][0]:
            nextCell = (current[0] - 2, current[1])
            nextPath = (current[0] - 1, current[1])
        elif current[1] - 2 == visited[current][2][1]:
            nextCell = (current[0], current[1] - 2)
            nextPath = (current[0], current[1] - 1)
        Tools.drawCalculation(
            canvas, mazeSizeCell, nextPath[0], nextPath[1],
grid
        )
        Tools.drawCalculation(
            canvas, mazeSizeCell, current[0], current[1], grid
        )

        # while loop for calculating the solution

```

```

while goal != (1, 1):
    solvedPath.append(goal)
    goal = visited[goal][2]
solvedPath.append((1, 1))

# records final time when solution is found before drawing
outputTimeElapsedSeconds = time.time() -
outputTimeElapsedSeconds

Tools.drawSolutionWhole(canvas, mazeSizeCell, solvedPath,
grid, speed)

outputSearchedPercent = (len(visited) - 1) / ( (
((len(maze) - 1) / 2) * ((len(maze) - 1) / 2) ) - 1 )
outputSolvedLength = len(solvedPath) - 1

return outputTimeElapsedSeconds, outputSearchedPercent,
outputSolvedLength

```

Figure 34 – Entire code of 'pathFind.py'




6100COMP Project

Monthly Supervision Meeting Record

Month Meeting: November 2022

Name: Keith Taylor.....

Main issues / Points of discussion / Progress made
<p>Not been documenting as I program – may make dissertation writing more difficult</p> <p>Spent much of the last couple weeks familiarising myself with Python and how to create a GUI with a little research into algorithm implementation</p> <p>Basic GUI nearly complete</p>
List of actions for the next month
<p>Complete GUI (as much as can be done without maze/algo implementation)</p> <p>Begin maze generation implementation</p>
List of deliverables for next time
<p>Application that has some amount of maze generation function</p>
Other comments
<p>Meeting deliverable late as I was ill last week, which resulted in our meeting being cancelled.</p>

Supervisor Signature 


Student signature: 

Figure 35 - November 2022 Monthly Meeting Report



6100COMP Project Monthly Supervision Meeting Record

Month Meeting: December 2022

Name: Keith Taylor

Main issues / Points of discussion / Progress made
Not much progress made as focus has been on other modules with impending deadlines
List of actions for the next month
Continue Maze implementation and begin Algo implementation Begin documentation
List of deliverables for next time
Application that has some amount of maze generation function
Other comments

Supervisor Signature *Reino Nish*

Student signature: *K. Taylor*

Figure 36 - December 2022 Monthly Meeting Report




6100COMP Project Monthly Supervision Meeting Record

Month Meeting: January 2023

Name: Keith Taylor

Main issues / Points of discussion / Progress made
Read book <i>Mazes for programmers</i> .
List of actions for the next month
Continue implementation Continue drafting dissertation Complete ethical training
List of deliverables for next time
Basic questionnaire to serve with implementation to colleagues
Other comments
Personal issues made progress slow over the break.

Supervisor Signature 


Student signature: 

Figure 37 - January 2023 Monthly Meeting Report




6100COMP Project Monthly Supervision Meeting Record

Month Meeting: March 2023

Name: Keith Taylor

Main issues / Points of discussion / Progress made
<p>Behind on the work overall as I've not begun writing the report</p> <p>Artefact is 99% complete</p>
List of actions for the next month
<p>Complete report</p>
List of deliverables for next time
<p>Complete project</p>
Other comments

Supervisor Signature 

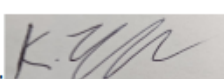
Student signature: 

Figure 38 - March 2023 Monthly Meeting Report