**Design Overview**

In this section of the documentation, I plan to describe and design the main functionalities of the technical solution. Throughout, I aim to thoroughly investigate all the key algorithms in the program, while representing this through flowcharts, class diagrams and python code. Envisioning this code, I will create designs for the structure of the code and how the code will flow in the project.

**Using Python and Object-Oriented Programming**

Python is a multi-purpose, high-level languages that is used in a variety of different circumstances such as machine learning or even creating websites (both front-end and back-end). Furthermore, python can be used to make games through one of the popular game libraries, pygame. Pygame is a 2D python library largely used to make games or 2D graphics in python. Due to my prior experience in pygame and my intermediate programming skills in python, I know this is a strong fit as I am already familiar with the pygame syntax and have already used it to make small games and applications.

There are more python game libraries I can use, including Arcade, a python arcade game development library, however, pygame has more extensive use in the community and with the integration of better graphical user interfaces, this separates using pygame from arcade.

Additionally, there are more optimised languages over python to use for making games like C# (C-Sharp), C++ or even Java. All these high-level languages also have accompanying game engines too, but due to the sophistication of the project, it is more logical to use a simpler game development environment like python to create this game.

A popular programming paradigm, Object Oriented Programming (or commonly referred to as OOP), uses classes to create objects for a program. This is a widely used programming structure due to the decomposition of code making it easier for the developer to construct their program. OOP uses instances of classes to create multiple objects have the same structure but could have different properties, for example, a class would be human, the object would be its name and the properties (generally unique to that object) would include data like age and gender perhaps. OOP uses these same principles in programming to make a more logical program. In its many adaptations, python can implement OOP structure through using classes and imbedded functions.

OOP would be relevant to my projects as there are many times I will need to create (possibly) thousands of instances of a class. The node structure in the abstract data structure graphs, requires the connection of lots of nodes that have all the same attributes but different positions. Using an object-oriented approach, I can make lots of nodes very quickly that all use the same structure but contain different data. The most logical choice for my game is to use an object-oriented approach as this massively reduces the complexities of my game.

**Technical Requirements**

*Python Version***:**

Development in Python 3.7.7

Application Deployment: 3.x.x

*Python Software:*

Development in Thonny: Version 3.2.7

Application Deployment: Any Capable Python Software that can run Python 3 and download libraries from Python Package Index (PyPI)

*Python Libraries:*

Pygame, Sys, Random

*Pygame Version:*

Development: 2.0.1

Application Deployment: Recommended Pygame 2.0+ (Earlier Pygame versions **not** tested)

*Platforms:*

Windows, macOS and Linux/UNIX

*System Requirements for Windows:*

Most Python Install Packages require Windows XP or later, however, Windows 7 or later is recommended.

*System Requirements for macOS:*

No Specified System Requirements for macOS, however, it is recommended that the Operating System is installed at the most recent stable software update.

*System Requirements for Linux/UNIX:*

No Specified System Requirements for Linux/UNIX, however, it is recommended that the Operating System is installed at the most recent stable software update.

The project is to be completed using Python 3.x (My project is completed in 3.7.7 however alternate python versions should be able to run the program). I chose to use python as this is the language, I am most competent in and able to complete this project in.

Due to the limited deployment in python and specifically pygame, the application is locally based and can only be run through a python shell with the source code. However, future developments could work on deploying­­­­­­­­ the game in a better way, but for the scope of this project, this will not be investigated.

It is essential for the python software to be able to download packages from PyPI as pygame is a library from PyPI and will mean the program will not be able to run without this.

**Algorithms and Data Structures**

**Graph and Tree Data Structures**

Data structures are a colossal part of computer science as these are ways developers can store data in a way that is efficient and works very linearly, meaning there is only one way they work enabling more developers to understand how the infrastructure of the code works. For this project, I will predominantly use graphs and trees.

A picture containing watch

Description automatically generatedGraphs are a non-linear data structure made of nodes and edges; see Figure 1 for an abstract diagram of a graph (GeeksforGeeks, 2021). The edges are the connections between nodes, representing that those nodes are inter-connected. This data structure involves the connection of nodes to represent maps, networks or (in my project) maze structures. This is the structure that will be used throughout the game and will allow the user and the computer to navigate the nodes.

Figure 1: A Graph Data Structure. A node is where the number is and an edge is the connections between them

A tree is a type of graph that contains no cycles, meaning there is only one way to get to every node in the structure. This is relevant because the recursive backtrack creates a tree data structure in my game where there is only one correct path to the end point. However, this may not apply in the maze creation as the user can decide whether to use a tree or graph, but to keep the program simple, I will keep this as a graph data structure. Maze solving will start as a graph data structure and then turn into a tree data structure once the algorithm has been applied to the structure.

Ensuring I follow the rules of the data structures and I ensure that my code is sticking to one data structure, I can easily implement these into my game.

Initially, the flow of data in this game is limited. The maze will be created through an algorithm based off the abstract data structure, graphs. Graphs consist of interconnected nodes and edges (lines connecting nodes to nodes). This will be the infrastructure of the maze and I aim to build around this underlying data structure.

As the main data structure of the project will be graphs, stack and queue operations are likely to be commonly used, when necessary. Nodes of the graph structure will be stored as objects that are stored in one-dimensional arrays.

**Maze Generation Algorithm**

The Maze Solver will use a maze generation algorithm to allow the user to solve the maze presented to them.

**Recursive Backtracking**

Recursive Backtracking is a type of depth-first search that is used on a graph data structure. It is an adaptation of the original algorithm that works by starting on a node and traversing to a neighbour that hasn’t been explored and removing the wall between them. The algorithm is fairly efficient and uses a time complexity of O(n2) (worst case). As this is a depth-first search, the algorithm reaches the deepest node and backtracks from there, meaning that the maze created changes its data structure from a graph to a tree. I believe this algorithm is perhaps the most ideal to use because of the simplicity of the algorithm and how easy it is to implement into my game.

**Recursive Division**

Recursive Division is a randomised maze generation algorithm. By splitting the maze grid with a wall, in half (either horizontally or vertically), the algorithm will split that wall at a point and then the algorithm will recurse but this time using a different wall. The algorithm will divide the maze grid up into sections and for every iteration or recursion, the maze grid is divided further. Whilst this is a very simple algorithm, I believe this would not be an ideal algorithm as the maze would be completely random and there is a possibility that a path is not defined between a start point and an end point.

After researching a few more algorithms, I found that recursive backtracking was the most ideal algorithm for my program. Most of the maze generation is randomised, however, the recursive backtracker can be manipulated easier to create harder mazes as the difficulty changes (due to the nature of the algorithm as it will naturally create dead-ends and lots of decision points). Furthermore, the algorithm will create a ‘perfect’ maze, meaning the maze contains only one path to any node.

**Maze Solving Algorithm**

Shortest Path Algorithms originated with Dijkstra’s Shortest Path in 1956, finding the fastest route between two points on a weighted graph. Shortest path algorithms became more efficient with an A\* pathfinding algorithm that took heuristics into account. My maze game will utilise a shortest path algorithm to find solve a maze created by the user.

**Dijkstra’s Shortest Path Algorithm**

Dijkstra’s Shortest Path works on a weighted graph by recording data about the graph in an adjacency matrix or adjacency list, depending on the size of the graph. The algorithm will consider every node and every edge in the graph and make a calculated decision on where to traverse by considering the path of the least cost. This will normally iterate meaning that the algorithm will keep trying to consider the nodes of least cost (the weight of the edge) and nodes that are unvisited. Whilst this will reach the goal node eventually, this algorithm is considered inefficient as lots of the decisions are based off the path of least cost instead of where the goal node is located.

**A\* Pathfinding Algorithm**

The A\* (a-star) Pathfinding Algorithm improves on Dijkstra’s shortest path and makes it more efficient.

The main difference is that A\* uses estimates or professionally, uses heuristics. These heuristics give an indication of where the end goal is located. Assuming the goal node and start node have a known location, a heuristic algorithm can be applied in 2 ways, Manhattan distance or Euclidian distance.

Manhattan Distance is a mathematical concept that can be used a heuristic value. For example, if we take 2 vectors, A and B in the coordinates [0,0] and [3,4] respectively, the Manhattan distance finds the sum of the absolute differences (Brownlee, 2020).

A picture containing text, clock, watch, gauge

Description automatically generated

Manhattan Distance would be |0-3| + |0-4| which results in 7

Euclidian Distance is another mathematical concept that is also used for heuristics. If I take the same 2 vectors A and B, the Euclidian Distance finds the direct distance between these two points (hypotenuse).

A picture containing text, clock, watch

Description automatically generated

Taking A and B, the Euclidian distance is 5.

For the A\* algorithm, it doesn’t matter which heuristic approach is used as they are both relevant estimates as to how far a node is from another. However, for my project I will use the Euclidian distance as this will return the hypotenuse distance between the two points regardless of whether the node has connections to the goal node. Most A\* algorithms use Manhattan however this is because they know all the connections of that node and of every node before the algorithm occurs. Unfortunately, this will not be known and will require too many calculations, reducing the overall performance of the program.

Additionally, the A\* makes decisions based off an F(x) value which is calculated by:

F(x) = G(x) + H(x); where G(x) is the cost between nodes, in my maze game this would be 1 as they are all next to each other, and H(x) which is the heuristic value shown earlier. A combination of this and a use of a priority queue in the A\* algorithm, shortest paths can be found quickly and efficiently.

Overall, I want to use the A\* pathfinding algorithm over Dijkstra’s shortest path algorithm as it is the faster algorithm to use and more logically built than Dijkstra’s shortest path which tends to be more undirected.

**Differentiating Difficulty**

After the maze has been generated, the maze needs to be set of a certain difficulty. This can be achieved through using a depth first search. After the starting and goal position has been added, a depth first search will be implemented to check the maximum number of nodes needed to reach the goal. I feel as this is a good algorithm to use as it roughly suggests how hard a maze is, by counting the number of nodes it goes through and calculating a percentage on how much of the grid is covered.

**Algorithm Breakdown**

For the three major algorithms that I will use, I will identify the algorithm and state the data structures used in the algorithm and some of the key features/functionality of the algorithm. The data structures are essential and considered the backbone for many algorithms as many algorithms manipulate these data structures to create

|  |  |
| --- | --- |
| *Algorithm* | *Data Structure and features* |
| *Recursive Backtrack* | *Graph – Non-linear data structure that will be represented through objects. Uses Booleans, lists and objects with a for loop to create a backtrack and a recursive technique to create the maze. Will convert a graph data structure into a tree data structure.* |
| *Depth first search* | *Tree – Non-linear data structure that is represented through objects. Contains integers, objects, lists, also uses a for-loop to search through neighbours, and uses a recursive method to iterate through the maze.* |
| *A\* pathfinder* | *Graph – Non-linear data structure that uses objects to represent nodes. It uses dictionaries, nested dictionaries, lists, stack functions to enable the algorithm to search through the maze. Lots of for-loops and while-loops are used instead of recursive techniques.* |

**Additional Features**

Upon the construction of the base algorithms, I need to create some of my own algorithms that will need to be used to ensure the game moves smoothly.

Detecting walls is one major algorithm that needs to be implemented. This is essential for the depth-first search so that it doesn’t count neighbours on opposite sides of the wall as an accessible neighbour. Additionally, this is required for movement of the user. As the maze designs will be different, this means this algorithm must work differently depending on the maze game mode the user is playing. If the user plays maze solving, the wall detection will have to check whether the wall is between two different nodes as the structure of the maze will be different. On maze creation, the object node, will have an attribute for wall, which is a Boolean, and states whether the node is a wall or isn’t.

Furthermore, more algorithms like verification algorithms to check whether mazes are valid; they have an existing start point and end point, the path between these covers at least a maximum of 90% of the maze. The algorithm to differentiate difficultly too, will have to be a section of these verifications to ensure the maze is correct and the game is running as it is supposed to.

Further technical challenges will include problems like compiling the entire game as one, ensuring the game runs smoothly. As lots of the game is split up into sections, I aim to package this into one function, where if this function is called, the entire sequence of events of the game starts running. Ensuring the game runs smoothing is using recursive techniques in places where for loops could be an example of how the program is optimised to run as efficiently as possible, maximising the processing power or different sections of the game.

**Maze Solving**

**Flowcharts**

Diagram

Description automatically generated

The Maze Solver starts with the program initialising a grid of nodes (in a graph data structure) that the maze will be built upon. The user then selects a difficulty of easy, medium and hard, and then the maze is generated using the recursive backtrack. This maze data is stored as an array and can be outputted onto the screen using one of the object’s methods. A verification occurs then to ensure that a maximum number of nodes is covered in this maze between the end point (at a random position) and the start point (at [0,0]). Following the verification, the user is presented with another interface where they can now solve the generated maze. The user can quit to the menu at any time during this part. Once the maze has been solved, the user is shown their time to complete the maze and can exit to the solver menu where they can do another maze, or they can exit to the main menu and play a new game mode perhaps.

*Diagram

Description automatically generated*This is how the recursive backtrack algorithm works in my program. The declaration of current Cell occurs outside of the function and is called in as a parameter instead. The process will use a for loop to cover every node and will recurse if that node has not been explored. If that node has not been explored, the wall needs to be broken between these nodes. This creates a perfect maze, one that has no cycles and contains only dead ends. This data structure switches the graph into a tree, because it no longer contains cycles.

*Diagram

Description automatically generated*

The depth first search uses the same structure as the recursive backtrack, however, this is a search and will look for the goal node instead. This also uses recursion and a for loop to ensure that the algorithm is depth first, so every dead end is explored. This is useful as it will explore the maximum number of nodes needed to find the goal, which will help me design the difficulty verification. After the goal node is found, the nodes travelled to are all stored in a list so the number of total nodes covered can be calculated; this is so that the verification algorithm can use this number to find a percentage of the maximum nodes explored.

**Class Diagram**

The program uses 2 main classes, the grid and the player (which is used for the player and to indicate the start point and end point).

Diagram

Description automatically generated

The class *Grid* declares following variables seen in the diagram above; x position, y position, neighbours\_near, wall and is\_visited. Neighbours\_near, wall and is\_visited are used in the classes’ methods, draw (takes a colour as a parameter) and get neighbours (takes the list of every node in the program as a parameter). The program uses instantiation to create hundreds of nodes that inherit all the attributes from the class *Grid*. Note, the only way to distinguish one node from another is to check their x and y position. For the draw method, the colour will be white to represent the walls, and a list of all instances of *Grid* is entered into a list where this is used as a parameter in get\_neighbours’ method.

On another hand, the class *Block* only has 2 attributes and like *Grid,* uses a draw method that takes a colour as a parameter. Player, which is controlled by the user, is green and has a changeable x and y position. Start point and end point are both very similar and are unable to change their x and y position, as this would obviously make the game impossible.

It is essential for the x and y position to be integers and not floats or real numbers as this means during calculations, the coordinate is a whole number not a number with several decimal places. Therefore, checks can be completed when the number is integer as all the numbers are also integers.

**Python**

*Text

Description automatically generated*

This is the Recursive Backtrack algorithm that will be used in the program. It takes the current node and the list of all the nodes in the graph as parameters. The node is set to as visited, and a list of the neighbours is generated. As the algorithm is random, the list must be shuffle and then a for loop iterates through those neighbours. If the node hasn’t been visited, the wall will be broken and then will recurse using that neighbour as the current cell. The algorithm finishes when all nodes have been checked through using the for loop and the recursion.

*Text, letter

Description automatically generated*

This is the depth-first search algorithm used in the maze solving section of my game, to find the maximum number of nodes between the start and end point, in order to set a generated maze of a certain difficulty. The algorithm takes a start node, end node, the list of all nodes in the graph and the path which is called visited. As my game uses 2 different coordinate systems, the start node will have coordinates that are relative to all other nodes (this would be [1,1], a coordinate relative to the number of rows and columns), not the pixel position coordinates (this would be [20,20], a coordinate relative to the width and height of the screen; these are measured in pixels) which display this on the screen. To remove this impediment, a quick function is used to get that node’s pixel coordinates instead of the other coordinates. Before the search begins, it completes a check to see whether that node is the end node. If it is not, that node is added to the visited list. The neighbours of that node are obtained and then the function iterates through all the elements in the list. If the neighbour is not in the visited list and if the check wall algorithm returns False (the algorithm checks the walls surrounding that node and decides whether that node can go to the next node without passing through a wall), the program iterates that neighbour as the new start node. Once the goal has been reached, this triggers the algorithm to stop recursing and then the program slowly backtracks as it will go through every dead end it has been through since reaching the goal, and finally returns the list of the maximum number of nodes explored to reach the goal.

**Maze Creation**

**Flowcharts**

*Diagram

Description automatically generated*

The Maze creation begins with the user choosing a grid size for the maze allowing the computer to generate a grid of walls. The user is also given the ability to break the walls and add walls using mouse presses. After adding a start point and end point, the user confirms the maze and now must choose whether the computer will solve it, or another user will. If another user is chosen, the maze is redrawn, and a timer is added to the game with the user now able to manually traverse the maze using the same mechanism used in maze solving. When the user solves the maze, the time taken is shown and the user can return to the maze creation menu or the main menu. On another hand, if the computer is chosen, the shortest path algorithm is implemented on the maze and the maze solved is shown if it is completed. The user is presented with the same options and can make another maze or return to the main menu.

Diagram

Description automatically generatedThe following is a rough A\* pathfinding algorithm, lots of data about the nodes must be obtain before the conditional statements are used. This is the f(x) value that must eventually be calculated for every node. This A\* uses iterations instead of a priority queue or perhaps recursion There are many different forms of an A\* algorithm and this is one of many adaptations. Personally, I will use an iterative approach over a recursive approach as I have more control over the algorithm, however, this comes at the cost of a higher time complexity considering this is an efficient algorithm. This could be adapted to become more efficient, although for this project, any shortest path algorithm can be used, regardless of using optimisation techniques.

**Class Diagram**

A picture containing text, receipt

Description automatically generated

The classes for the Maze Creation are very similar to the class structure in maze solving. The grid class is slightly different as instead of creating lines, the program creates nodes as squares instead of intersections, so the pathfinding algorithm is calculated faster.

Additionally, only once instance of the Player class is created because the class is essentially the grid class, but only contains attributes for x and y coordinates. The start and end points use the grid class and are indicated by the colours blue and red respectively.

**Python**

*Text

Description automatically generated*

The A\* pathfinding algorithm works like the following: a graph of the maze must be created using a dictionary data structure where the key is a node followed by its neighbours as the values. Then, it will sort the nodes by their fx value with the smallest fx value first. Then for the smallest node’s neighbours, if that neighbour is in the graph and has a larger fx value than the smallest node’s gx plus that neighbour’s heuristic value, then the neighbours fx is calculated. Eventually, path will leave the values with the lowest fx values, which can be resolved to find the shortest path in order. This approach is effective for finding values if the start and goal nodes are far apart too. Although traditional A\* pathfinders use recursion, I believe iterations work well too to achieve the shortest path, despite having a higher time complexity due to the increased number of iterative functions.

Upon the algorithm overview, the hardest algorithm to make is the A\* pathfinder. The algorithm uses lots more processes, the heuristics add to the complexity of the algorithm. After the creation of this algorithm, I can confirm that this was true, and was easily the most complex and hardest to fully understand the scope of the algorithm.

**Architecture Description**

The program uses OOP features like instantiation and inheritance, however, to allow the game to run properly, the program needs to adapt to a procedural approach. The game begins with an initialiser function which handles the menu screen and uses global variables to ensure that other aspects of the game can use them. Global variables are considered bad practise, however, in this case, the variables are only changed once in the initialiser function where the user has to set/declare sizes for the maze. Originally this was kept outside of the function, although to allow the user to play more than one game, the program needed a way to repeat itself, meaning all variables and attributes must be set to their default values.

Furthermore, the program will perform a ‘handoff’ to ensure the program runs smoothly. For example, if the player reaches the goal node, that function will return a Boolean value indicating whether the player has reached the position and will consequently cause the program to run the win screen function which displays the win screen and its necessary information.

I believe the procedural approach to the game works well as I can test individual parts of the code if I need to which is very important when the project goes into testing.

**User InterfacesGraphical user interface, text, application

Description automatically generated**

Figure 2 Main Menu

Graphical user interface, text

Description automatically generatedGraphical user interface, text

Description automatically generatedText

Description automatically generated with medium confidence

Figure 4 Create Menu

Figure 3 Solve Menu

Text

Description automatically generatedA picture containing diagram

Description automatically generated

Figure 6 Solve Win Screen

Figure 5 An example of Solve: Option 2

A screenshot of a computer

Description automatically generated with low confidenceChart

Description automatically generated

Figure 8 Create Option 3: Completed Maze

Figure 7 Create Option 3

A picture containing text, crossword puzzle

Description automatically generatedGraphical user interface, application

Description automatically generated

Figure 10 A Computer Visualisation of their solution

Figure 9 Create Decision on who will Solve

Text

Description automatically generated

Figure 11 Computer Option Win Screen

A screenshot of a computer

Description automatically generated with low confidence

Figure 12 User Option Chosen at Decision

Text

Description automatically generated

Figure 13 User Win Screen in Creation

The user interfaces represent a basic human computer interaction, with hints and controls that allow the user to navigate the program. Although the UI could be more complex, this could be something that can be explored later as it is outside of the scope of the project. I believe this is a very simplistic design and works exactly as it is supposed to.