School of Computing
FACULTY OF ENGINEERING AND PHYSICAL SCIENCE

UNIVERSITY OF LEEDS

Final Report

A Moderately Difficult Maze Game

Dante Saxton-Knight

Submitted in accordance with the requirements for the degree of BSc Computer Science

2020/21

40 credits

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Report	*Report (PDF)*	*Minerva (10/05/21)*
Project directory, containing game assets and python scripts	*Download URL*	*Supervisor, assessor (10/05/21)*
Video of final product	*URL (YouTube)*	*Supervisor, assessor (10/05/21)*

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) 

Summary

This project is an exploration into maze generation and video game difficulty. The intention is to create a maze-based game that is challenging but enjoyable for any player, regardless of their level of skill.

This is achieved by monitoring the player's performance and adjusting the game's parameters accordingly, such as the number of obstacles and the algorithm used for generating mazes.

Acknowledgements

Thanks to Tom Kelly, my supervisor for this project, and Brandon Bennett, my assessor, for their help and advice.

Thanks to my family for their love and support, and for patiently partaking in game testing.

Table of Contents

Chapter 1
Introduction

1.1 Motivation

Early single-player video games of the 1980s were punishingly difficult challenges made for a niche audience of dedicated players. As the target demographic for video games increased, developers needed a way of accommodating a wider range of players with varying levels of skill. The most common way this is done in modern games is by including multiple difficulty levels.

Most single-player games include the option to select a difficulty level at the beginning of the game. This usually affects the entire gameplay experience, making it more or less challenging. The problem with this is that the player cannot know which difficulty is best suited to them without first having played the game. The names of these difficulty options are usually vague, such as “easy” or “hard”, and what one player considers hard another may consider easy. Some players will under- or overestimate their abilities resulting in a boring or frustrating experience.

This problem can be addressed by using dynamic difficulty adjustment (DDA). In this system, instead of the player selecting a difficulty at the beginning, the difficulty changes in real-time depending on how well the player is performing [1]. The system measures the difficulty the player is facing at a given moment and adjusts the experience, increasing or decreasing the number of challenging features in response to how easy or difficult they are finding the game. The difficulty should eventually converge to a level according to the player’s level of skill, and this relative level of difficulty is determined by the developer; a game with a DDA system may challenge every player, or be a casual experience for every player, or be of moderate difficulty for every player. Finding this balance depends on the nature of the game and the parameters available for adjusting the difficulty. These could include the number of obstacles the player has to overcome, the duration of a time limit or the nature of the game’s environment.

1.2 Aims, Objectives and Deliverables

This project aims to create a maze-based game in which the difficulty level is automatically adjusted based on the player’s skill level. This will be achieved by adjusting the algorithm used to generate mazes and by changing the number of obstacles. A time limit will be specified, and the game will adjust the difficulty until the player matches this completion time. The game will be programmed in the Python programming language using Pygame [2].

The objectives of this project include:

1. Research maze generation algorithms and dynamic difficulty adjustment.
2. Become familiar with Pygame.
3. Create an algorithm for generating mazes of variable difficulty.
4. Create an interface for the game itself.
5. Experiment with DDA systems and expand on the game's features.
6. Collect data from players to investigate the efficacy of the DDA system.

Using these objectives as a guide, the intention is to explore various ideas through the challenges that arise when developing an enjoyable video game. The final product should demonstrate the effectiveness of the underlying maze generation algorithm and DDA system.

The deliverables of this project include:

1. The project files and source code.
2. The project report.
3. A video of the final product.

1.3 Project Methodology

This project is planned using a style similar to the Agile development process [3]. This style was chosen over a stricter model such as the Waterfall process [4], as this project is focused on exploring ideas and designs rather than implementing a set plan. The maze generation algorithms and the game itself will both need multiple revisions of design, and these two aspects of the project can be worked on individually.

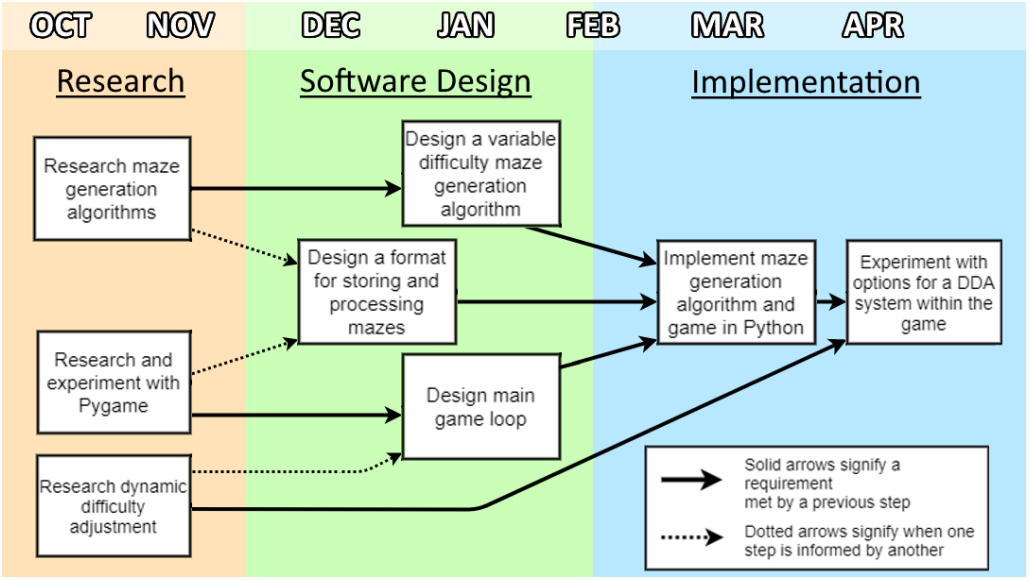


Figure 1: A general project plan. In practice, the software design and implementation stages will be informed by each other through multiple iterations.

Chapter 2
Background Research

2.1 Dynamic Difficulty Adjustment

A dynamic difficulty adjustment system detects whether a player is finding a game too easy or too difficult and adjusts the difficulty accordingly [1]. The design of a DDA system depends heavily on the nature of the game, but some general concepts can be applied to all DDA systems.

2.1.1 Flow

When a game is too difficult the player is likely to become frustrated, and when a game is too easy the player may lose interest. However, if the game is just challenging enough, the player will be satisfied and enter a state of full engagement. This is called a state of Flow [5], more commonly known as being “in the zone” [6].

If the player’s level of skill is plotted against a game’s difficulty, the region of low skill/high difficulty represents a state of frustration, and the region of high skill/low difficulty represents a state of boredom. The space between these regions is the Flow channel [1] and represents a state of satisfactory player engagement.

For a DDA system to move the player into the Flow channel, it must adjust the difficulty of the game so that it feels neither too hard nor too easy to the player. A simple method for moving a player into the Flow channel would be to first test what the skill level of the player is, by subjecting them to a sample of gameplay, and then use the results of this test to set the difficulty for the rest of the game.

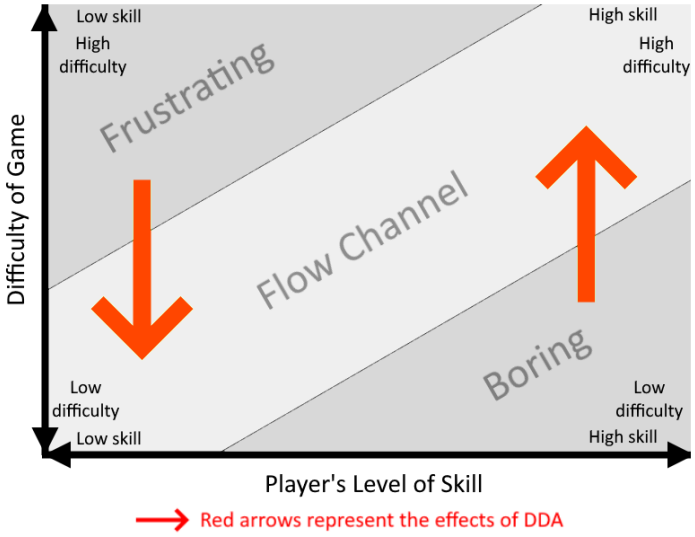


Figure 2: The Flow channel represents the fully engaged state between boredom and frustration. A DDA system should shift the player out of these undesirable states and into the Flow channel by adjusting the difficulty [1][5].

This fails to account for the fact that players will gain experience as they play and become more skilled at the game over time. If the difficulty remains constant and the player's skill increases, the player will eventually drift out of the Flow channel and become bored.

To combat this, the difficulty should be dynamically adjusted throughout the player's experience. As the player improves, the DDA system should direct the player up the Flow channel by continuously adjusting the difficulty [5].

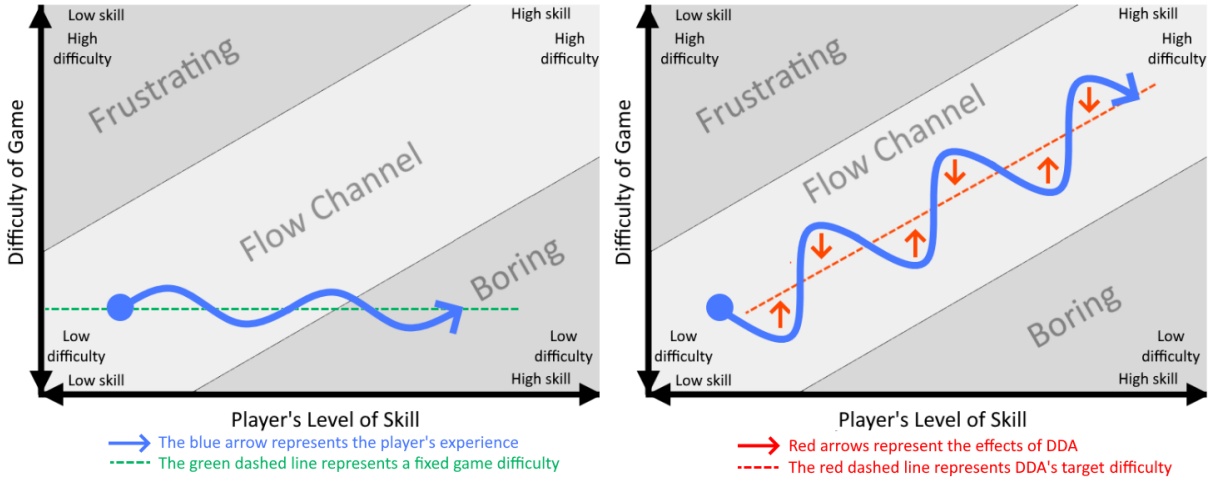


Figure 3: The DDA system should direct the player up the Flow channel as their skill improves, rather than letting them drift into the "boring" state.

This means an effective DDA system should have a method of measuring the difficulty faced by the player at any given moment.

2.1.2 Heuristic Challenge Function

Since the difficulty faced by a player is a subjective experience, it can be difficult to accurately measure. The reason that a game is challenging will often be due to a combination of factors, each contributing in various ways to the overall challenge.

One method of determining the level of difficulty faced by a player is by using a challenge function [7]. A challenge function is a type of heuristic function [8] which takes information about the current state of the game (such as how many obstacles the player has touched, or how much of a time limit remains), and selects the difficulty it thinks the player is facing.

This decision is made by following a decision tree, for example:

- If the player is running out of time and has received a penalty for touching an obstacle, then they are probably finding the game difficult.
- If they are running out of time but they haven't touched an obstacle, then they are probably finding the game moderately difficult.

- If they aren't running out of time and haven't touched an obstacle, then they are probably finding the game easy.

This could provide inaccurate results if, for example, one player's strategy involves intentionally running onto enemies. This strategy is commonly employed in games that grant the player a brief period of invincibility upon touching an enemy ("invincibility frames" [9]). In this case, touching an enemy is not a good indication that the player is finding the game difficult.

To know exactly how difficult a player is finding the game, they might need to be asked a long series of questions or be subject to a physical measurement such as a brain scan [7]. Heuristics provide a simple shortcut by assuming that certain game states will be more or less difficult for most players. This method presents a compromise between accuracy and efficiency [8]; by using more heuristics, the accuracy of our assumption may be improved at the expense of a more complex challenge function.

2.1.3 Hamlet System

The Hamlet system, designed by Hunicke and Chapman, controls the difficulty level by encouraging certain game states and discouraging others [1].

Most games can be abstracted into a set of states that the player transitions between in loops. For example, in a game such as Pac-Man, the player must eat every dot in a maze while being chased by ghosts. If the player eats a "power-up" then the roles are temporarily reversed; the player becomes invincible and the ghosts become edible, running from the player until the power-up is exhausted [10]. This gameplay can be abstracted to a set of looping states such as chasing ghosts, running from ghosts etc.

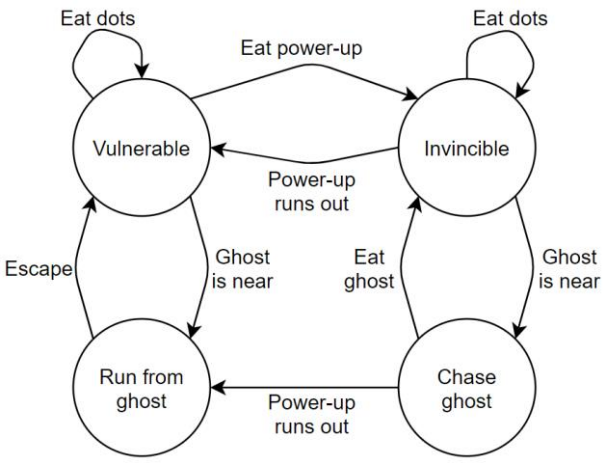


Figure 4: A simplified state diagram describing the possible states and transitions in the game Pac-Man.

This type of model is called a finite-state machine [11]. The Hamlet system attempts to identify which loops keep the player in the Flow channel, and which loops lead to boredom or frustration. By simulating the progression through these states, the system identifies when the player is stuck in an undesirable loop and intervenes to move them into a more enjoyable one.

2.2 Maze Generation Algorithms

A maze is a structure filled with winding corridors and dead ends, usually in the form of a rectangular grid of navigable cells separated by walls. Mazes are often employed as part of a game or as a standalone puzzle. The aim is to get from the entrance to the exit and the challenge arises from traversing the maze without getting lost. To make this more challenging, there are usually no loops or alternate pathways; there is only one direct path from any point to any other point in such a maze.

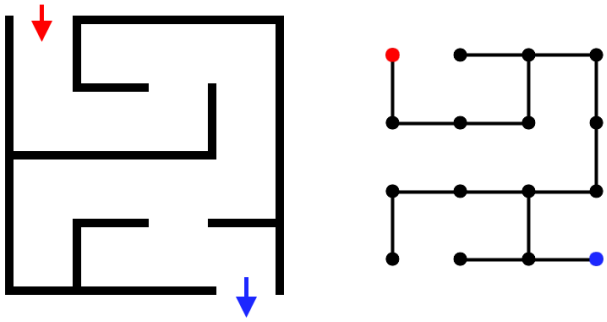


Figure 5: A typical maze and its graph representation.

These types of mazes, called “perfect mazes”, are isomorphic to spanning trees in square grid graphs, in which cells are represented by vertices and open passages are represented by the tree’s edges [11]. For any given size of grid, there are many possible spanning trees, but most spanning tree algorithms will systematically produce only a single tree. For the purpose of maze generation, it is desirable to have a wide variety of mazes with different characteristics. Most spanning tree algorithms can be adapted into maze generation algorithms with the addition of an element of randomness.

2.2.1 Recursive Backtracker/Depth-First Search

The recursive backtracker algorithm [12], also known as depth-first search (DFS), is one of the simplest and most effective algorithms for generating challenging perfect mazes. This algorithm uses a stack and records each cell as being visited until there are no unvisited cells left.

Given an initial grid of cells where each cell is surrounded by four walls, and starting with any cell, the algorithm can be described with the following steps [13]:

1. Push the current cell to the stack and mark it as visited.
2. If there are any unvisited neighbours of the current cell:
    a. Randomly select an unvisited neighbour.
    b. Remove the wall between the current cell and the selected neighbour.
    c. Make the selected neighbour the current cell, go to step 1.

3. Else, if there are no unvisited neighbours:
    a. Pop the stack and set the popped cell as the current cell, go to step 2.

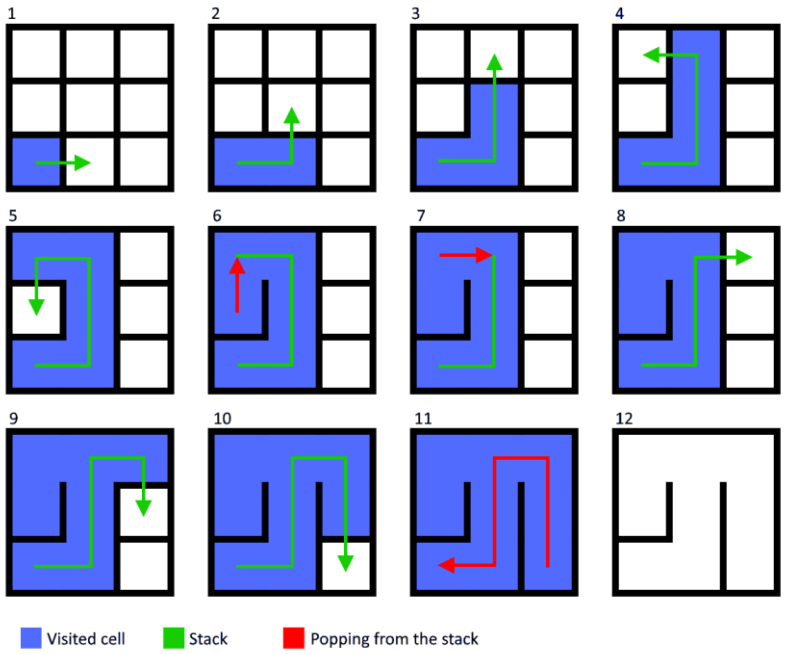


Figure 6: A graphical representation of DFS. A passage is carved until there are no unvisited neighbours (step 6) at which point the stack is popped until an unvisited neighbour is found (step 8). The final backtracking steps are condensed in step 11.

The mazes generated by DFS are usually quite challenging as they feature many long winding corridors that lead to dead ends [13]. This also typically results in an aesthetically pleasing maze.

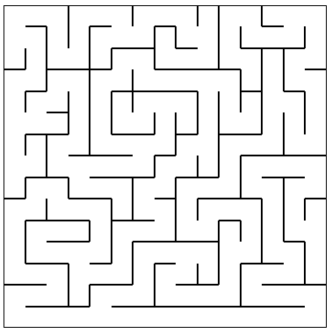


Figure 7: A 15-by-15 maze generated with DFS.

2.2.2 Prim's Algorithm

A randomised version of Prim's algorithm [14] can also be used to generate perfect mazes starting with an initial grid of walled cells. This algorithm keeps track of cells in the maze as a set. Starting with one cell in the set [13]:

1. Pick a cell at random from the set of cells in the maze.
2. If the current cell has neighbours that are not in the set:
    a. Pick a neighbour that is not in the set.
    b. Remove the wall between the current cell and the neighbour.

   c. Add that neighbour to the set, go to step 1.
3. Else, if there are no neighbours that are not in the set, go to step 1.

This algorithm produces mazes with many short passages and fewer long winding paths. This typically results in a maze that is easier to complete, as most dead ends are immediately obvious [13].

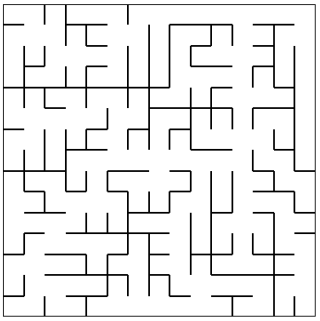


Figure 8: A 15-by-15 maze generated with Prim's algorithm.

2.2.3 Recursive Division

The recursive division algorithm [15] differs from DFS and Prim's algorithm in that it is not graph-based and involves adding walls rather than removing them. Starting with an empty grid and calling this space a "chamber" [13]:

1. Bisect the chamber into 2 smaller chambers with a randomly positioned dividing wall.
2. Remove a single section of this wall so that the 2 chambers are connected.
3. For both of these smaller chambers, go to step 1.

This is repeated until every chamber is a single cell wide or tall. This algorithm results in perfect mazes with long straight walls, which makes them typically easier to solve than DFS-generated mazes. They are usually trickier than mazes generated with Prim's algorithm as they feature fewer short dead ends [13].

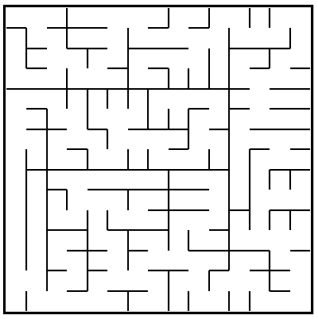


Figure 9: A 15-by-15 maze generated with recursive division.

2.2.4 Aldous-Broder Algorithm

The Aldous-Broder algorithm [16] is a simple algorithm for generating uniform spanning trees. Starting with any cell:

1. Select a random neighbour to the current cell.
2. If this neighbour is unvisited:
   a. Remove the wall between the current cell and the selected neighbour.
   b. Make the selected neighbour the current cell, go to step 1.
3. Else, make the selected neighbour the current cell, go to step 1.

This algorithm is unique in that it is equally likely to generate any spanning tree, and has no bias unlike the other algorithms mentioned. However, since it visits neighbours randomly without discriminating between visited and unvisited neighbours, this algorithm is very inefficient.

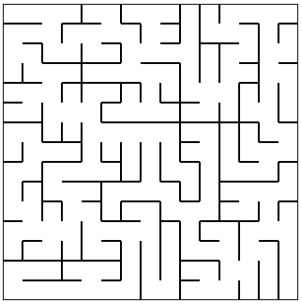


Figure 10: A 15-by-15 maze generated with the Aldous-Broder algorithm.

2.2.5 Hierarchical Mazes

A hierarchical maze is made up of many smaller mazes joined together [17]. If any two perfect mazes are placed next to each other and a single random wall section is removed from the border between them, this will connect the two mazes into a larger perfect maze.

These smaller mazes, or sub-mazes, can be arranged into a grid and connected to form a super-maze; as long as the smaller mazes are joined in such a way that they form a perfect maze of mazes, the overall maze will still be a perfect maze.

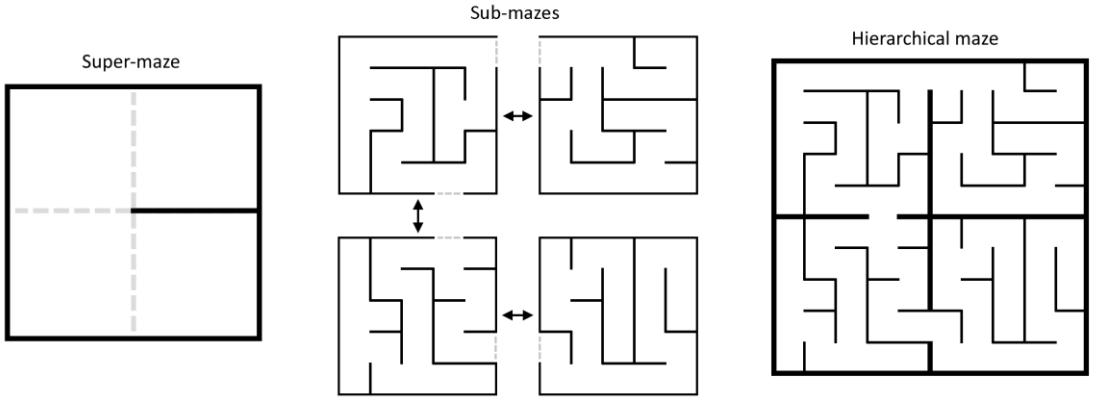


Figure 11: A super-maze of perfect mazes forms a larger perfect maze.

This is because each maze is a spanning tree. If two trees are joined by a single edge, then they form a larger tree. Any number of trees can be joined in this way, and if no cycles are formed the result will be a larger tree [18].

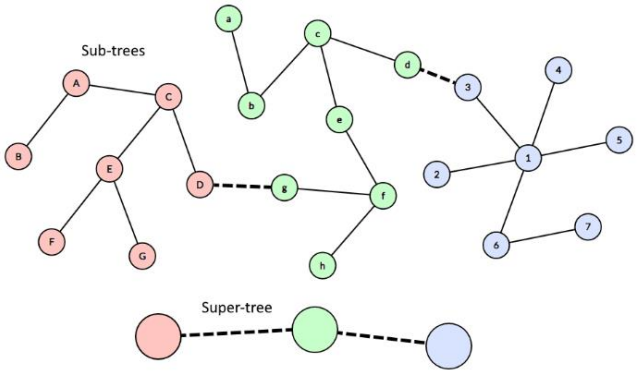


Figure 12: A tree of trees forms another tree.

Chapter 3
Making a Moderately Difficult Maze Game

3.1 Maze Generation Algorithm Design

This project requires an algorithm that can generate mazes with a specified level of difficulty. The brute-force method would be to generate random mazes until a suitably difficult one is found. While this can be practical for small mazes it becomes impractical with larger ones, as the domain of possible spanning trees on an *n*-by-*n* grid exponentially increases with *n* [17][19]. A more efficient solution would involve identifying characteristics that make a maze more or less difficult and incorporating these characteristics into randomly generated mazes.

3.1.1 As a Switch Statement

One possible approach would be to classify several pre-existing algorithms by the difficulty of the mazes they produce and simply select one of these algorithms based on the desired difficulty. This approach has a number of problems, one being that each algorithm produces stylistically different mazes and so it would be obvious to the player when the difficulty of the maze has been adjusted. There is also a lot of variation of difficulty between mazes generated by a single algorithm, making this method unreliable.

3.1.2 Using a Predetermined Solution Path

Another approach would be to create a solution path first and to generate a maze around it. This would limit the scope of the problem to producing paths of a specified difficulty. However, a new problem arises of generating an effective maze given the constraint of a predetermined path. This is incompatible with most pre-existing algorithms:

DFS starts by generating a path on the stack until a dead end is reached, and further branches are grown from this initial path. However, if the first path generated is the solution path, there will be no branches along the solution path and the maze will always be easy to complete. This is because the first branches will be formed at the very end of the solution path, and these branches will grow to fill all remaining space in the maze leaving no room for further branching.

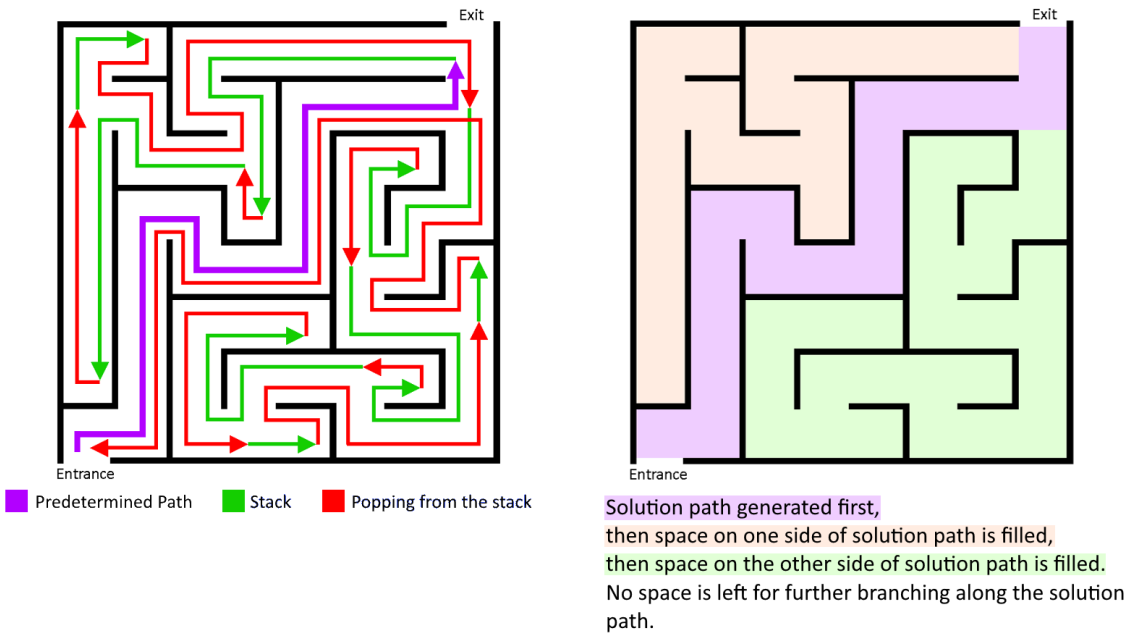


Figure 13: DFS with a predetermined solution added to the initial stack produces trivial mazes which lead the player directly to the exit.

Recursive division is based on iteratively dividing rectangular chambers rather than carving winding tunnels, so a predetermined path is incompatible with this algorithm.

A predetermined path could be successfully incorporated into Prim's algorithm by setting the initial set of cells to be the predetermined path. However, Prim's algorithm produces the easiest and least aesthetically pleasing mazes, so this solution is not ideal.

3.1.3 Using Hierarchical Mazes

Another solution would be to construct hierarchical mazes such that the sub-mazes are completely random, but the super-maze is generated to have a certain path length. As the super-maze is much smaller in dimensions than a complete maze, the domain of possible super-mazes is smaller [11] and the scope of the problem is decreased significantly. Assuming that the average difficulty of a random DFS maze is proportional to its size [11], the difficulty of the hierarchical maze should be roughly proportional to the difficulty of the super-maze (when using DFS-generated sub-mazes of constant size).

To expand on this, if the super-maze has dimensions of 4-by-4 and the sub-mazes are 5-by-5, then the hierarchical maze will consist of 16 sub-mazes (Figure 14). If the solution path of the super-maze involves solving 7 sub-mazes, then the total maze should be easier than if the solution path involves 11 sub-mazes. The sub-mazes are all the same size, so their average difficulty should remain approximately constant. This can be shown by the fact that the solution path length of the super-maze is approximately proportional to the solution path length of the entire hierarchical maze.

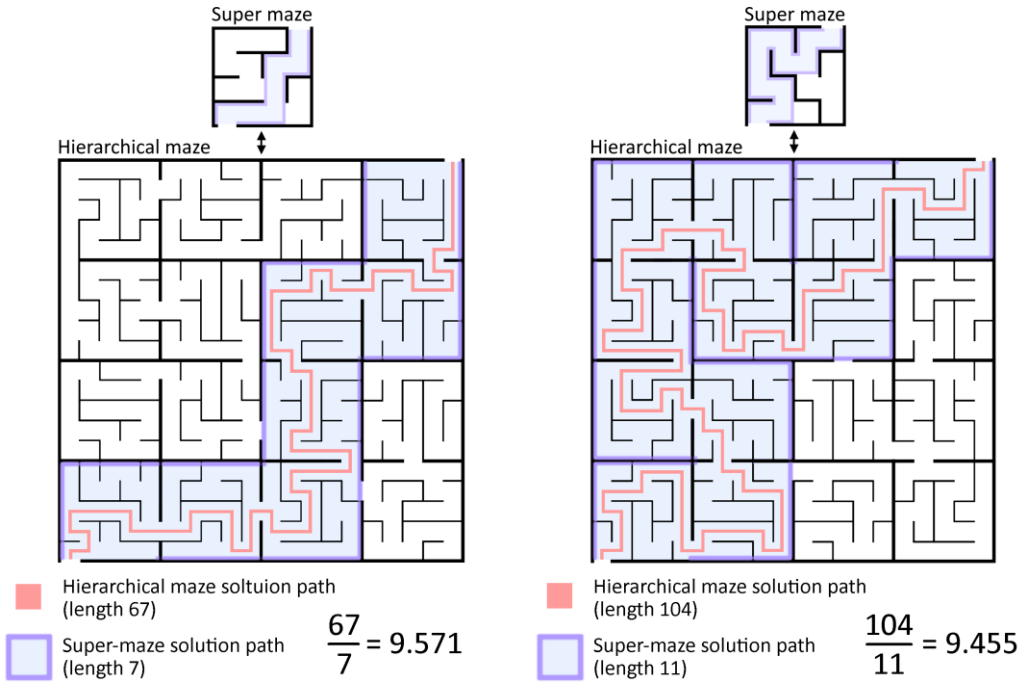


Figure 14: Hierarchical mazes consisting of randomly generated DFS mazes and their solution paths. The ratio of super-maze and sub-maze path lengths remain approximately constant between different hierarchical mazes. This ratio gives the average path length of the 5-by-5 sub-mazes.

This is assuming that the solution path length of a maze is a good indication of difficulty. However, there is a problem with this assumption which can be seen by taking the extreme case; if the solution path is maximal, then the entire maze consists of a long path to the exit with no branches. Solving such a maze would be tedious rather than challenging. Similarly, when the super-maze path length of a hierarchical maze is maximal, then the challenge would simply involve solving a series of very easy sub-mazes in a linear order.

There is also a problem inherent to all hierarchical mazes in that their structure can be quite obvious, so the player may be able to identify and solve the super-maze first. In this case, solving the hierarchical maze is only slightly more difficult than solving the super-maze.

These problems arise when trying to solve a hierarchical maze as a standalone puzzle. However, hierarchical mazes can be employed in a game with features that mitigate these issues.

3.2 Game Design

The game involves solving hierarchical mazes under a time limit. To increase the challenge, enemies are scattered through the maze which the player must avoid. The enemies move when the player moves, and their movement patterns are influenced by randomness.

3.2.1 Difficulty

The path length of the super-maze may not be a reliable indicator of difficulty in the case of solving a standalone hierarchical maze, but the addition of enemies in this game challenges the player's ability to strategically navigate mazes rather than just their ability to solve them.

A longer super-maze path length challenges the player to avoid more enemies. This is because the game involves circumventing enemies by waiting for them to wander into dead ends, allowing the player to pass without touching them. If the super-maze path length is greater then there are fewer dead ends for the enemies to wander into, and the player is forced to come into proximity to a greater number of enemies.

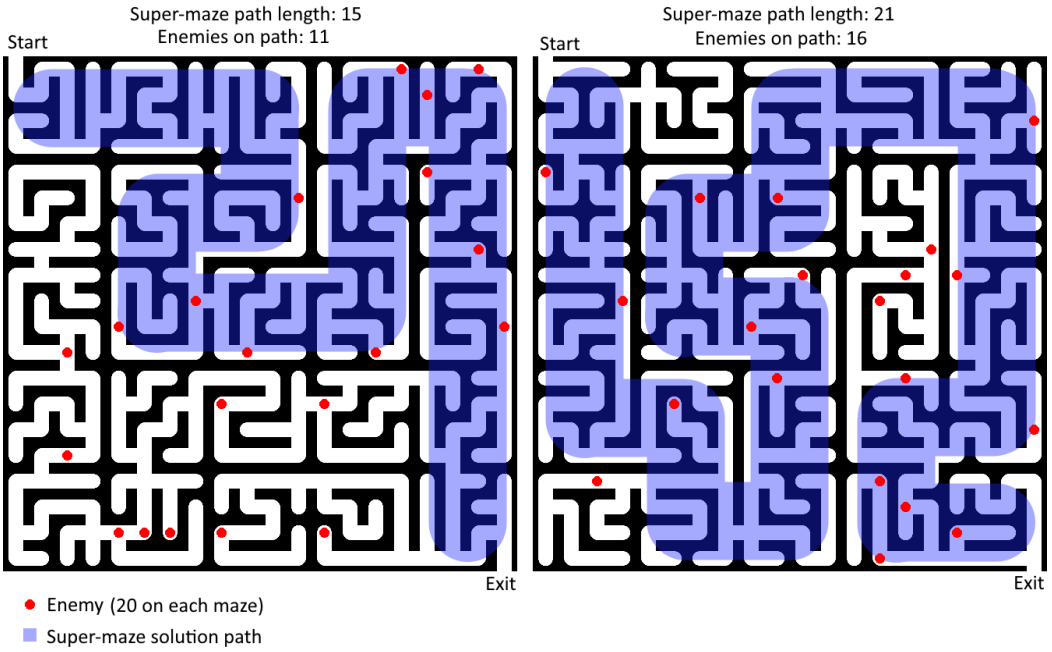


Figure 15: Two mazes with different solution path lengths and randomly placed enemies. A longer super-maze solution path forces the player to circumvent more enemies.

3.2.2 Enemy and Player Movement

The game is turn-based, in that enemies will only move once for every move the player makes; if the player is not moving, the enemies will not move. Enemies will explore the maze randomly and occasionally chase the player. Enemies can also be forced to advance a step without the player moving by pressing the space button. To prevent unfair situations where the player is immediately trapped, enemies cannot spawn near the entrance, and they can never enter the top-left super maze.

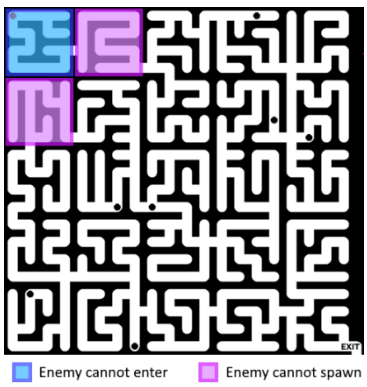


Figure 16: Enemies cannot enter the blue region or spawn in the blue/purple regions.

3.3 Implementation

The game consists of two Python files. One contains the algorithms for generating sub-mazes/super-mazes and stitching hierarchical mazes together. The other contains the interface and logic for the game which presents these mazes to the player and provides a means of interaction.

The sub-mazes are generated using the depth-first search algorithm. There are multiple options for representing a maze as a data structure, and this choice will determine how the algorithm is implemented.

3.3.1 Mazes as Graph-based Data Structures

Since mazes are a form of spanning tree, one method would be to use a graph-based representation where every cell is a vertex and every passage is an edge (Figure 5). Graphs can be stored in memory as an adjacency matrix [20]. However, this representation loses information about the relative position of each cell and it would not be possible to reconstruct the maze in a grid. This is because an adjacency matrix only describes the relationships between nodes and not their spatial positions.

One graph-based option which conveys spatial information is a quadtree [21] wherein each neighbouring cell is stored in a format similar to a linked list. Each parent node has four children denoting the neighbour to the north, east, south, and west. With this information, it is only possible to determine the *x* and *y* coordinates of a given cell by traversing the quadtree and keeping track of which nodes have been traversed. This would be inefficient for DFS as every step of the algorithm requires checking neighbouring cells, and this information is not conveyed directly in a quadtree.

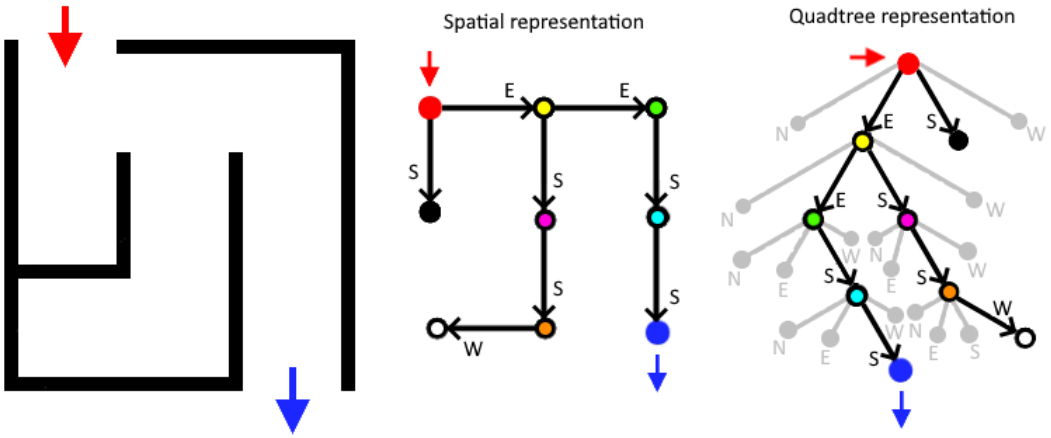


Figure 17: A maze represented as a graph with spatial information and as a quadtree. From the quadtree, it is not immediately obvious that the black and white nodes are neighbours, although this can be inferred by the sequence of directions.

Although quadtrees can convey spatial information in two dimensions, they are better suited to representing subdivisions in planes [22] (e.g. recursive division [15]). While trees can be used to represent mazes abstractly, the grid-like construction of a maze is essential to its structure and simple graph-based methods fail to capture this information.

3.3.2 Mazes as Array-based Data Structures

A more literal representation of a maze that conveys this information directly is a two-dimensional array of cells. Since a cell is defined by the walls surrounding it, one method would be to store the state of each wall as a Boolean value. This would be useful when implementing the DFS algorithm as it requires removing walls, and this could be achieved simply by switching the state of a wall in the array.

This data structure becomes less useful when considering the game's graphics. When the game draws the maze to the screen it will do so with a pool of images, one for each type of cell. If the image for a vertical corridor cell has the filename "Corridor 1.png" and a horizontal corridor cell has the filename "Corridor 2.png" etc., the game will need a function that translates each cell object in the array into its corresponding image filename.

Another option would be to simply use the array to store these filenames.

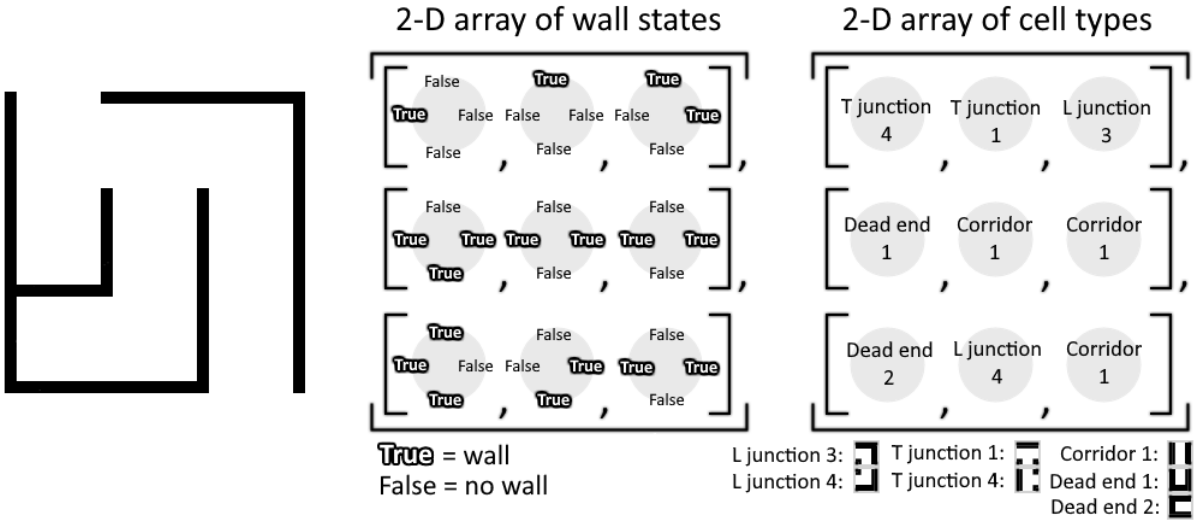


Figure 18: A maze represented as a 2-D array. The array of wall states stores four Booleans corresponding to each wall. The array of cell types stores filenames corresponding to each type of cell.

This way the array can be used as a lookup table by the game to immediately find which image it needs to use for each grid position. The problem with this format is that it compromises the efficiency of the maze generation algorithm. The information about which walls are present is lost, so these labels would have to be translated into a collection of wall states for the DFS algorithm to function. Since the maze drawing function and DFS algorithm

require these different formats, one solution would be to combine both ideas by having the filename contain Boolean information.

If each wall is assigned a binary digit, then each cell can be represented as a 4-digit binary number. For example, if the bits are ordered north wall – east wall – south wall – west wall, then the number 1000 represents a cell with a wall to the north ("T junction 1" in Figure 18), and 0110 represents a cell with a wall to the east and to the south ("L Junction 4" in Figure 18). This gives each cell type a unique ID number which also describes the arrangement of walls.

Instead of storing the numbers as integers in the array, they can be stored as strings. This way, the maze drawing function can simply append ".png" to the string returned by the array to get the corresponding filename. In this case, the image for a vertical corridor cell would have the filename "0101.png" and a horizontal corridor would have the filename "1010.png". Since strings are a type of array, using them instead of integers has the added benefit of transforming the two-dimensional array into a three-dimensional one. The DFS algorithm can change the state of a wall by changing the first, second, third or fourth character in the string from a "1" to a "0" or vice versa.

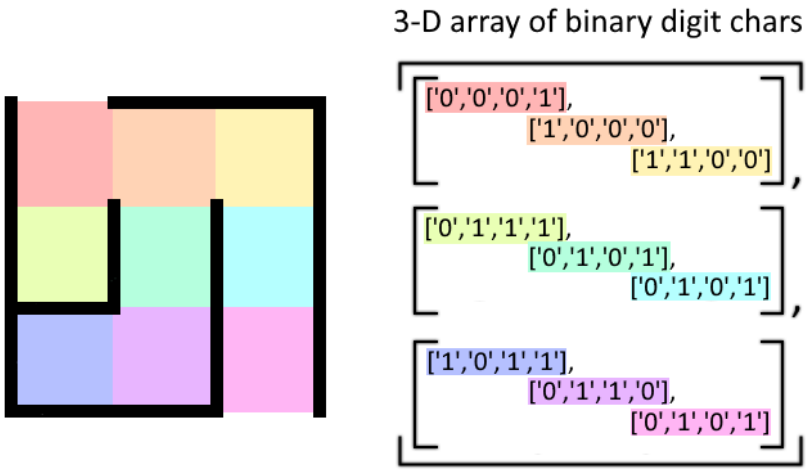


Figure 19: Each element in the 2-D array is a string, so it is equivalent to a 3-D array of chars. Each char represents a wall to the north, east, south or west of the cell. For clarity, each string has been colour coded to the corresponding cell in the maze.

This can also be used within the game logic to determine where a player can move. If the player is at $x = 2$ and $y = 1$ and wants to move north (Figure 20), the game can check the value of *array[1][2][0].* If this value is a "1", then there is a wall in the way. If the value is a "0" then there is free passage.

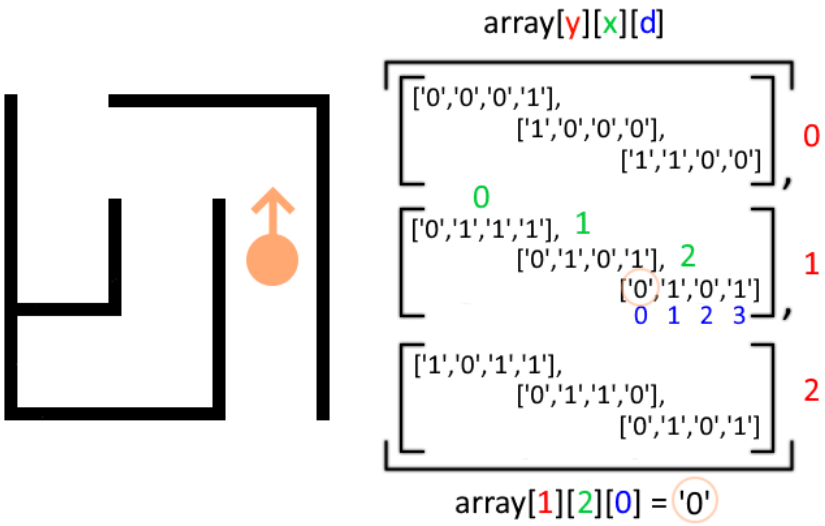


Figure 20: Coloured numbers indicate array indices. The value at array[y][x][d] gives the wall state of the cell at position (x, y) in direction d. Since array[1][2][0] = '0', the player at (2, 1) can move north.

3.3.3 Generating Mazes of a Specified Path Length

As described in sections 3.1.3 and 3.2.1, random sub-mazes can be arranged into a super-maze with a specified path length. This produces hierarchical mazes, the difficulty of which scale with the super-maze's path length when applied to a game with randomly placed enemies (Figure 15).

This algorithm requires a method for generating mazes of a specified path length, for use as the super-maze of the hierarchical maze. The simplest solution would be to employ a lookup table; store a pre-generated list of all possible mazes categorised by path length and return one at random. This lookup-table method improves computational efficiency at the cost of storage efficiency and becomes impractical for large mazes; by generating mazes systematically, it was found that the number of possible mazes in a 5-by-5 grid exceeds 200,000.

Another simple solution is the brute-force method: generate mazes randomly and stop when a maze with the specified path length is found. The efficiency of this method depends on the size of the maze and the path length specified; for an *n*-by-*n* grid, the number of possible mazes increases exponentially with *n* [17][19].

n	*n*-by-*n* mazes
2	4
3	88
4	3820

Table 1: Number of possible n-by-n mazes.

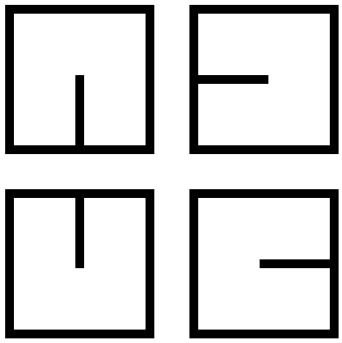


Figure 21: The four possible 2-by-2 mazes.

For 3-by-3 mazes with the entrance in one corner and the exit in the opposite corner, the only possible path lengths are 5, 7 and 9, providing three possible levels of difficulty. As path length *p* increases, the frequency of mazes with path length *p* decreases.

p	3-by-3 mazes with path length *p*
5	74
7	12
9	2

Table 2: Frequency of 3-by-3 mazes with path length p from one corner to the opposite corner.

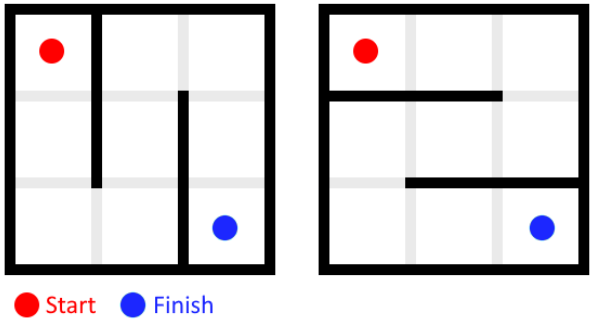


Figure 22: The two possible 3-by-3 mazes with path length 9.

This means the brute-force method is slow for large mazes and long path lengths, and the longest possible path length for a given maze presents the worst-case scenario.

This method relies on trial-and-error, so any maze that does not match the desired path length is discarded. One improvement to this method would be to keep every discarded maze in a list, along with its path length. This way, if a discarded maze of the specified path length has already been found in a previous run of the algorithm, it can be returned immediately.

For example, if the user specifies a 3-by-3 maze with path $p = 9$, there are only two out of 88 possible mazes that fit this criterion. It is likely that the algorithm will first generate multiple mazes with $p = 5$ and $p = 7$ before finding a maze with $p = 9$. These mazes with $p = 5$ and $p = 7$ can be saved in a list rather than discarded. If the user then specifies a path length $p = 7$ or $p = 5$, they can immediately be served a maze from the top of the list.

This solution is only effective for smaller path lengths as they are more likely to be discarded and added to the list, while larger path lengths see minimal improvement. To counter this, we could use the brute-force method for short path lengths and the lookup-table method for long path lengths; since the number of mazes with a longer path length is comparatively small (Table 2), all mazes of a sufficiently long path length can be pre-generated and stored in the list.

In other words, if the path length *p* is below a certain threshold, then the brute-force method will be used, but if *p* is above the threshold, then all mazes of path length *p* will be generated in advance. When this threshold is higher, computation is compromised for storage. When this threshold is lower, storage is compromised for computation.

The following table contains the average computation time over 100 trials for a 5-by-5 maze with path length *p*, and the frequency of 5-by-5 mazes with path length *p*. Mazes are generated using the DFS algorithm (Figure 6).

p	Average brute-force computation time for a 5-by-5 maze of path length *p* (seconds)	Frequency of mazes with path-length *p*
25	2.25	104
23	0.232	1884
21	0.0436	8288

Table 3: Average computation time and frequency of 5-by-5 mazes with path length p. 25 is the maximum path length for 5-by-5 mazes.

If we pick a threshold value of 22, the algorithm will use pre-generated mazes for *p* = 25 and 23, and brute-force for *p* = 21 or less, so the worst-case scenario has an average computation time of 0.0436 seconds. Since this computation time only affects the loading time between the game's levels, it is short enough for the purpose of the game.

This requires storing a list of 1988 pre-generated mazes in memory, each as a three-dimensional array of chars. The total size of this list is only 8284 bytes. For comparison, one of the game's graphics has a size of 133 kilobytes.

To implement an algorithm that selects mazes based on path length *p*, an algorithm is needed for finding the path length *p* of any given maze. One method would be to use a maze-solving algorithm such as A* [23] to find the solution path and its length. This is an inefficient solution as only the path length is needed, not the solution path itself.

The solution path length can be found with a simple algorithm which "floods" the maze, starting in one corner and following every possible path until the opposite corner is reached. The number of steps taken gives the solution path length.

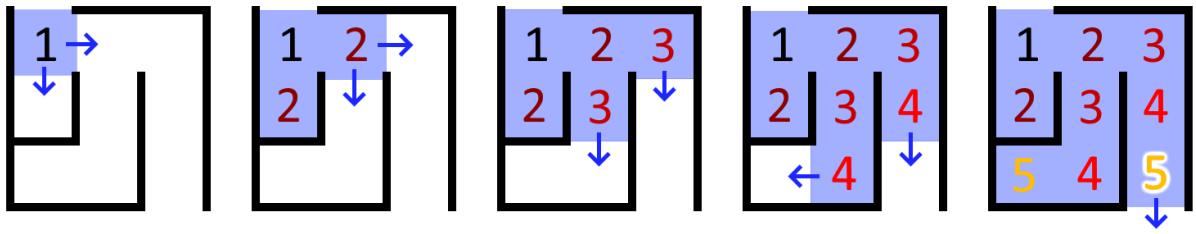


Figure 23: A maze with path length 5. "Flooding" the maze is an efficient method of getting the solution path length, but not the solution itself.

3.3.4 Constructing Hierarchical Mazes

The hierarchical maze algorithm should take an *n*-by-*n* maze (the super-maze), and a sub-maze size *m*, as input. It will generate a set of n^2 random DFS mazes (sub-mazes), each of size *m*-by-*m*, and connect them into the formation of the *n*-by-*n* super-maze.

This is done by first creating an *n*-by-*n* grid of *m*-by-*m* mazes. Since mazes are stored in arrays as stacks of rows (Figure 19), two mazes can be joined vertically by simply joining the arrays together.

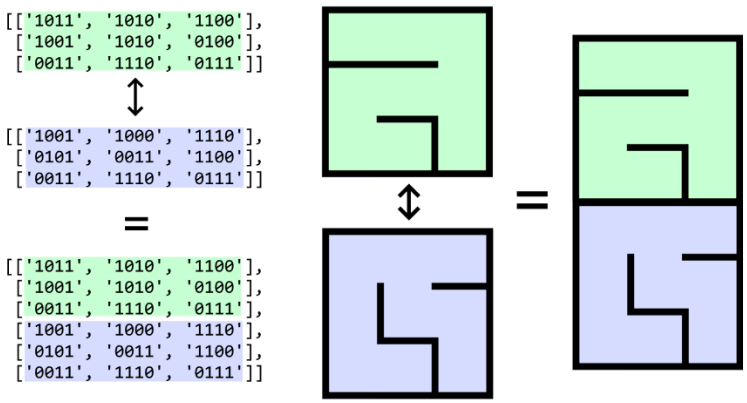


Figure 24: Mazes can be joined vertically by joining the arrays.

To join mazes horizontally, each row of the leftmost array must be individually joined to the corresponding row of the rightmost array.

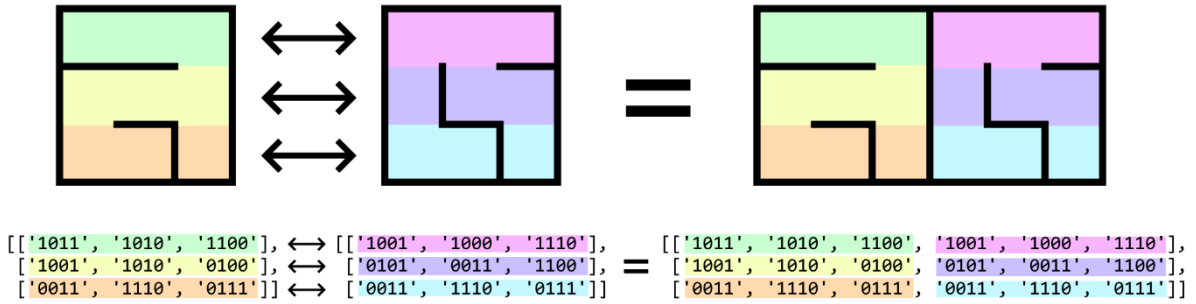


Figure 25: Mazes can be joined horizontally by joining each corresponding array row.

Once an *n*-by-*n* grid of sub-mazes has been created, the mazes are connected by removing a single random wall segment from the border that separates them. This is done according to the pattern of the input super-maze with the following method. For each cell in the super-maze:

- If the cell contains a northward passage, remove a random wall from the top edge of the corresponding super-maze.
- If the cell contains an eastward passage, remove a random wall from the right edge of the corresponding super-maze.

Since each sub-maze shares its top edge with the bottom edge of the sub-maze above it, and its right edge with the left edge of the sub-maze to its right, only the northward and eastward directions must be accounted for.

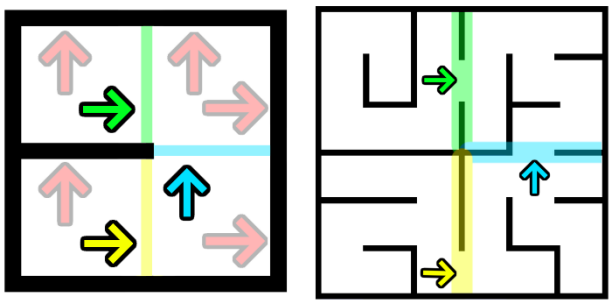


Figure 26: If a cell in the super-maze has no wall to the north or east, the corresponding sub-maze has a wall segment removed.

3.3.5 Enemies

For this algorithm to have a measurable effect on the game's difficulty, the game requires enemies that move around the maze. These enemies should provide a significant challenge to the player without frustrating them.

One simple algorithm for enemy movement would be to decide every move randomly:

1. The enemy picks a random direction to move.
2. If they cannot move in that direction, go to step 1.

This produces poor results, as enemies tend to move back and forth in small passages and have trouble exploring the rest of the maze.

For example, the following is a sequence of 10 randomly chosen directions (*N*orth, *E*ast, *S*outh, *W*est): *N, W, W, S, N, E, S, E, N, W*. If an enemy is in a horizontal corridor and follows this sequence of moves, it will move west twice, then east twice, then west once (as it cannot move north or south).

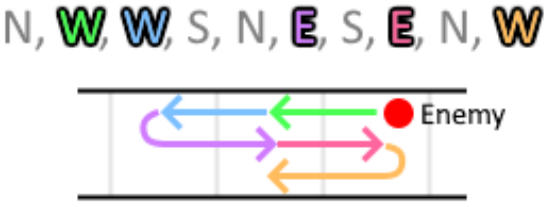


Figure 27: Enemies that move completely randomly tend to stay in the same general location for long periods.

This means enemies tend to block passages, and whether the player can pass is completely up to chance.

Another simple algorithm would be for enemies to move completely systematically. A common technique for solving perfect mazes is to constantly stick to the left- or right-hand wall, known as the wall follower algorithm [24]:

1. If there is no wall to the left, turn left and move forward.
2. Else, if there is no wall directly ahead, move forward.
3. Else, turn right and move forward.

For the purpose of solving mazes, this algorithm terminates when the exit is reached. If it does not terminate, enemies following this algorithm will continue to circulate the maze systematically, visiting every cell in the maze at least once before repeating the cycle.

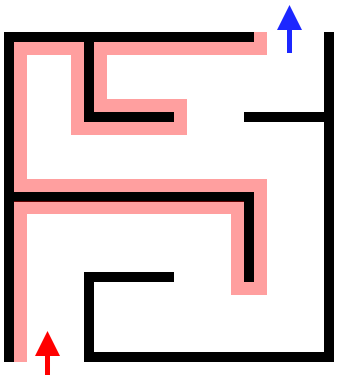


Figure 28: The wall follower algorithm using the left-hand rule.

This means enemies will always move through passages rather than blocking them. It also allows for situations where enemies can “chase” the player, providing more excitement, and allows the player to “chase” enemies by strategically following behind them.

While this is method is significantly more effective than always choosing random moves, the enemies are now entirely predictable which may leave players feeling under-challenged and removes any element of risk from the game. Ideally, enemies should move through corridors without suddenly turning around (such as in the wall follower algorithm) but they should be influenced by some randomness to make the game interesting.

This can be achieved by employing the random mouse algorithm [24]. With this algorithm, enemies always move forward through the maze until they reach a junction of three or more passages, at which point they make a random choice (choosing anywhere except the direction they came from). Enemies will never reverse direction unless they reach a dead-end.

This means enemies will chase the player/be chased by the player through passages and won’t linger in the same location for long periods. It also means the player knows exactly when an enemy will make a random choice, allowing for the player to plan their moves and make risk vs. reward decisions.

3.4 Experimenting with Dynamic Difficulty Adjustment

A DDA system requires a method of changing the difficulty of the game [1]. This will be done by defining 9 difficulty levels. The current difficulty level determines the super-maze path

length and the number of enemies; level 1 features the minimum path length (9 for a 5-by-5 super-maze) and 1 enemy, while level 9 features the maximum path length (25) and 9 enemies. Since a 5-by-5 super-maze has 9 possible path lengths, the path length p for a given difficulty level L can be calculated using the formula $p = 2L + 7$. The number of enemies is equal to the current difficulty level L.

3.4.1 Heuristics for Increasing/Decreasing Difficulty

A DDA system also requires a method of monitoring how well the player is performing in the game [1]. One simple heuristic method would be to consider only whether the player has failed or completed the previous maze:

- If the player loses by coming into contact with an enemy or running out of time, the difficulty level is decreased.
- If they win by completing the maze within the time limit, then the difficulty is increased.

The following data (Figure 29) was collected from a volunteer playing 25 consecutive mazes using this set of rules. The game starts at level 5 and the player has 80 seconds to complete each maze. The maze consists of a 5-by-5 super-maze with 4-by-4 sub-mazes (Figure 15).

Since the difficulty always starts at level 5, the first five results are ignored for the mean and standard deviation calculations (indicated by the green line). This is to give the system a chance to adjust the difficulty from its starting point before the data is analysed.

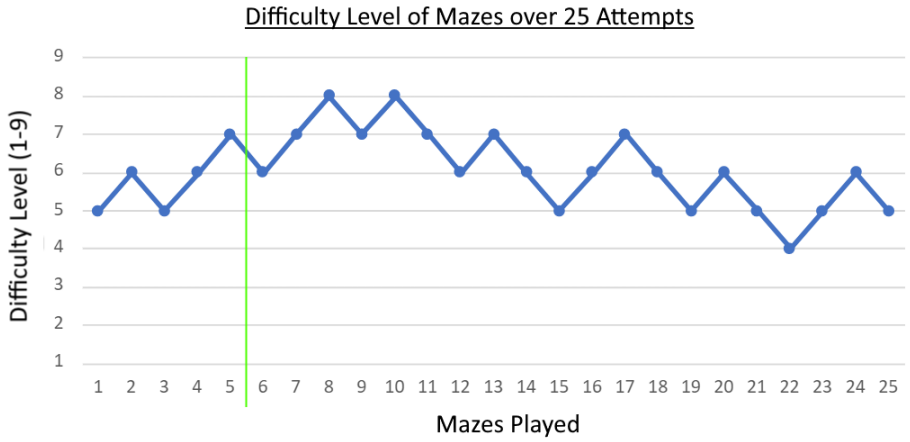


Figure 29: A graph showing how the level of difficulty changed for a player using this system. For the last 20 results: mean average = 6.1, standard deviation = 1.044.

This system results in the difficulty fluctuating constantly, as there is no way for the difficulty to stay the same. If the player keeps completing mazes, then each subsequent maze will only become more difficult, and the player is doomed to eventually fail. This is because the only way for the difficulty to decrease is if the player fails. This is not an ideal solution as it

requires waiting for the game to become too difficult before making it easier, which fails to keep the player within the Flow channel.

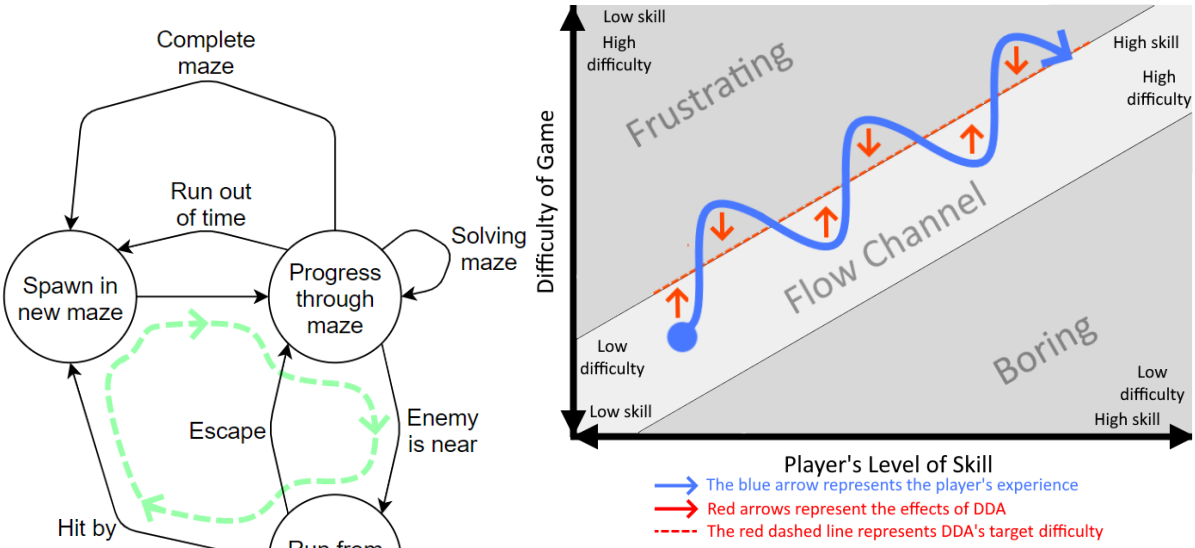


Figure 30: This system fails to keep the player within the Flow channel.

Figure 31: A simplified state diagram of the game, with an undesirable loop shown in green.

If the game is modelled as a finite-state machine (FSM) [25], this DDA system guarantees that players will periodically fall into the loop shown in green (Figure 31) when the difficulty becomes too high, which eventually leads to the player becoming frustrated. An effective DDA system will try to keep the game at a moderate difficulty for as long as possible [1].

3.4.2 Heuristics for Keeping Difficulty Constant

A maze that the player fails to complete can be seen as too difficult, but it is less clear how to define when a maze is too easy and when it is of moderate difficulty. One method would be to consider how much of the time limit remains when the player completes a maze; if there is plenty of time left then the maze was easy, but if the player completes the maze only moments before running out of time then the maze was moderately difficult:

- If the player completes the maze in under 50 seconds, the difficulty is increased.
- If they complete the maze in between 50 and 80 seconds, then the difficulty remains the same.
- If they lose by coming into contact with an enemy or running out of time, the difficulty level is decreased.

The following data (Figure 32) was collected from a player over the course of 25 mazes using this set of rules.

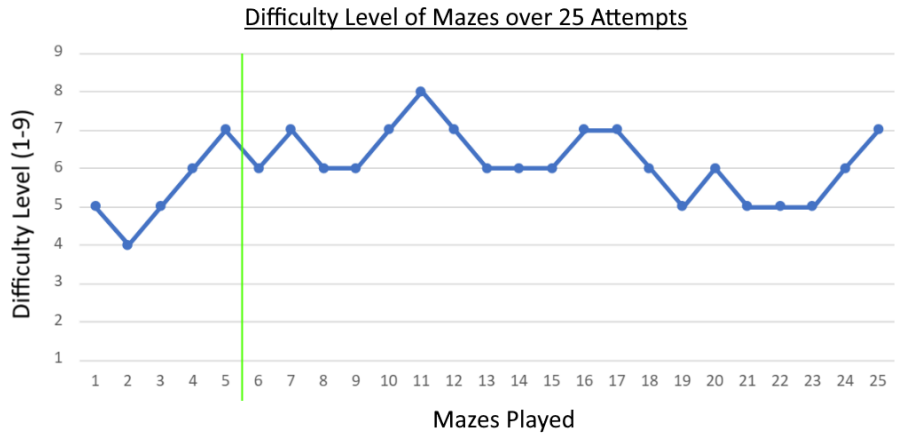


Figure 32: A graph showing how the level of difficulty changed for a player using this system. For the last 20 results: mean average = 6.2, standard deviation = 0.812.

With this change, the player faces roughly the same average difficulty but with fewer failures; with the previous system the player failed 12 mazes, compared to 8 with this system (shown in Figure 29 and Figure 32 as a downward slope). The difficulty is also more consistent as the standard deviation is reduced significantly.

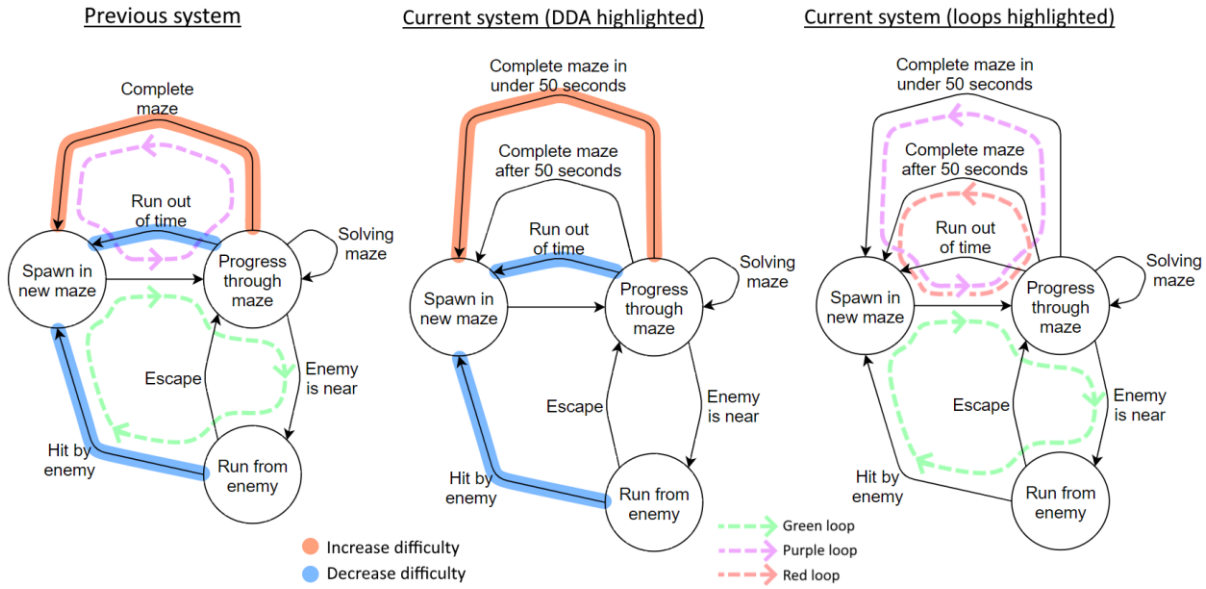


Figure 33: State diagrams for previous system (left) and current system. Two views of the current system are given, one with DDA's effects shown in orange and blue (centre), and the other with certain loops highlighted (right).

In this FSM model, DDA's effects are shown in orange and blue. The green loop represents a cycle of the player losing by hitting an enemy, and the purple loop represents a cycle of the player completing a maze.

In the previous system, the player always alternates between being in the purple and green loops (Figure 33, left), because staying in the green loop eventually forces the player into the purple loop and vice versa, due to the effects of DDA.

This system provides a new red loop (Figure 33, right) and staying in either the purple or green loops will eventually force the player into this red loop due to DDA. The red loop represents a cycle in which the player completes a maze with the time limit close to being exhausted ("Complete maze after 50 seconds"), which is typically when the player is the most focused and has entered the Flow channel.

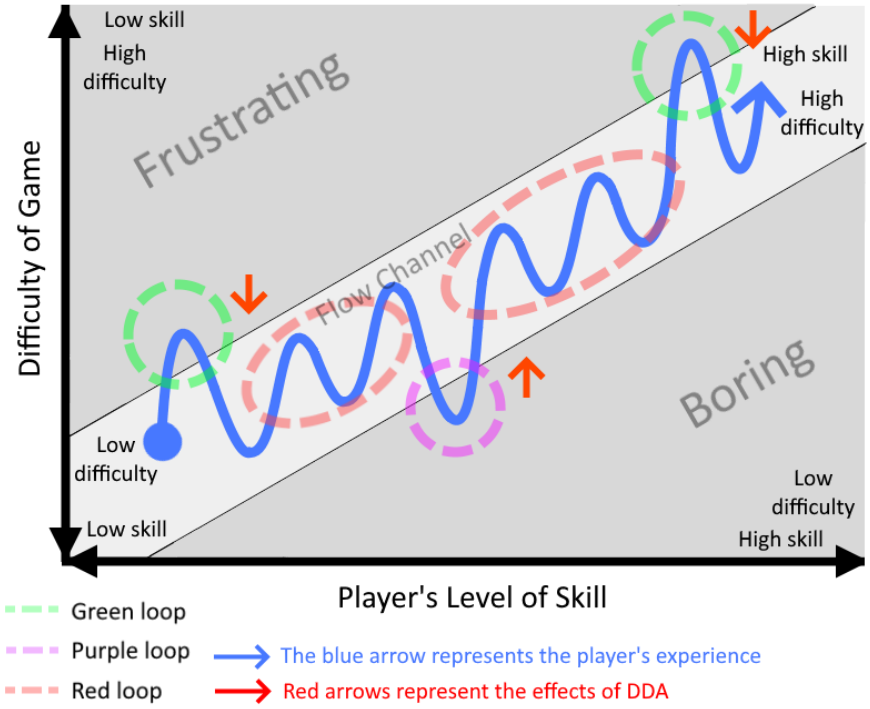


Figure 34: This system tends to move the player into the red loop, which keeps them in the Flow channel for a longer period.

While this is an improvement, the difficulty level still fluctuates considerably (Figure 32). There is also a problem with these systems in that the time limit has too little an effect; in all 20 cases of the player failing to complete a maze, it was because the player came into contact with an enemy, and not because they ran out of time. This could be because the time limit is too long, or because enemies are too challenging and cause the player to lose early. This could be mitigated by lowering the time limit and by lessening the effect that enemies have.

3.4.3 Extra Lives

One method of lessening the effect of enemies and keeping the player in a given maze for longer is to give the player an extra life. This means the player can touch at least one enemy without losing and can continue playing the maze. This extra parameter also provides a new heuristic that can be used to measure difficulty:

- If the player completes the maze in under 50 seconds and without losing a life, increase the difficulty.
- If they complete the maze after losing a life, the difficulty remains the same.
- If they complete the maze in between 50 and 70 seconds, the difficulty remains the same.
- If they fail by running out of time or losing all of their lives, the difficulty is decreased.

The following data (Figure 35) was collected from a player over the course of 25 mazes using this set of rules.

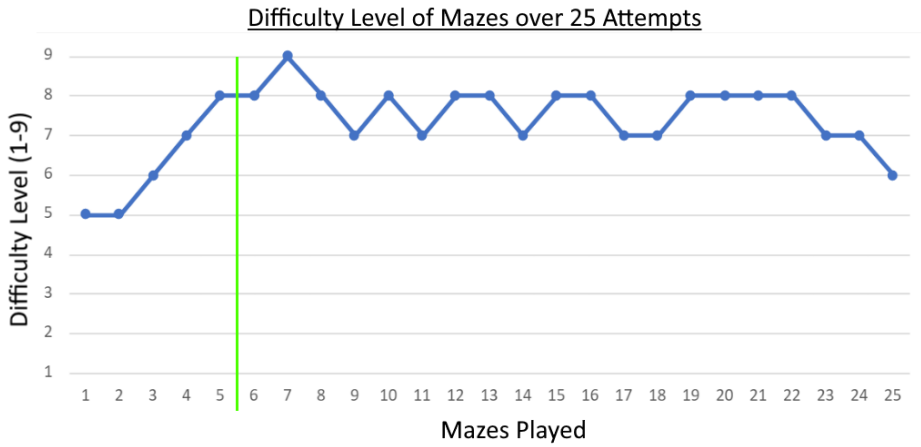


Figure 35: A graph showing how the level of difficulty changed for a player using this system. For the last 20 results: mean average = 7.6, standard deviation = 0.663.

With this system the standard deviation sees another significant reduction, indicating an improved consistency. This is due to the extra condition which allows the difficulty to stay the same. The average difficulty level has also increased, implying a change in overall difficulty; lower skill players can reach higher difficulty levels.

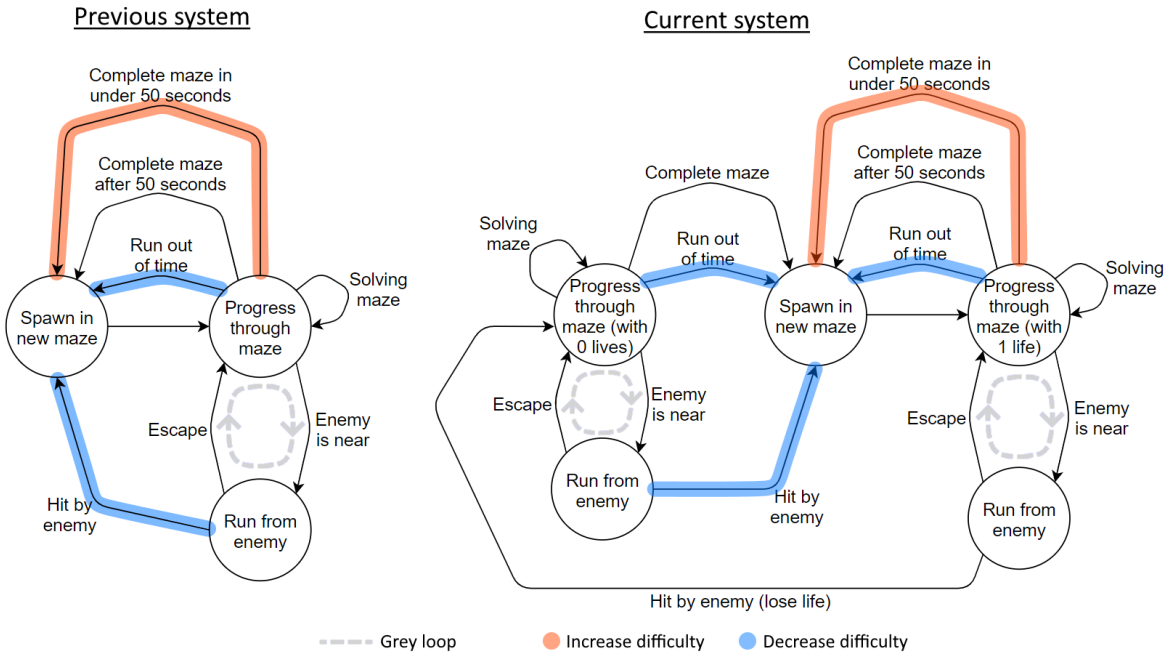


Figure 36: State diagrams for the previous system (left) and current system (right).

This system also encourages players to enter the grey loop, shown in Figure 36, more so than previous systems. This loop represents the case when the player is close to an enemy and must narrowly avoid it to progress through the maze. This is exciting for the player, as it presents a risk vs. reward scenario; the player can risk losing a life to make quicker progress. Avoiding enemies takes a lot of focus, so encouraging this loop should keep players within the Flow channel.

In previous systems this loop is discouraged; if the player touches an enemy once, they lose (Figure 36, left) which makes the risk greater than the reward and slows progress through the maze. In the current system, since the player has two lives, they get an extra chance to enter this loop if they touch an enemy (Figure 36, right). This loop is also encouraged by the fact that the average difficulty level under this system is higher than previous systems (Figure 35), which causes more enemies to appear in the maze for a player of a given skill level.

Chapter 4
Final Product

4.1 Presentation

Upon first loading, the game presents the player with an introduction and instructions for how to play the game. The player must get from the top left corner of the maze to the bottom right corner before running out of time.

The time limit is indicated by a bomb with a burning fuse to the right side of the screen. The spark on the fuse rotates to catch the player's attention, and the bomb glows red when the player only has 20 seconds remaining.

Enemies are represented as black circles, and the player as a red circle. If the player loses a life, this is indicated by a white spot appearing in the centre of the player.

Figure 37: The game's start-up screen.

If the player completes a maze, the screen flashes blue and a sound effect is played to indicate success. If the player fails a maze, the screen flashes red and an explosion sound effect is played. In both cases, the difficulty is adjusted, and a new maze is presented. The game features two pieces of looping background music.

Figure 38: The bomb starts glowing after 50 seconds.

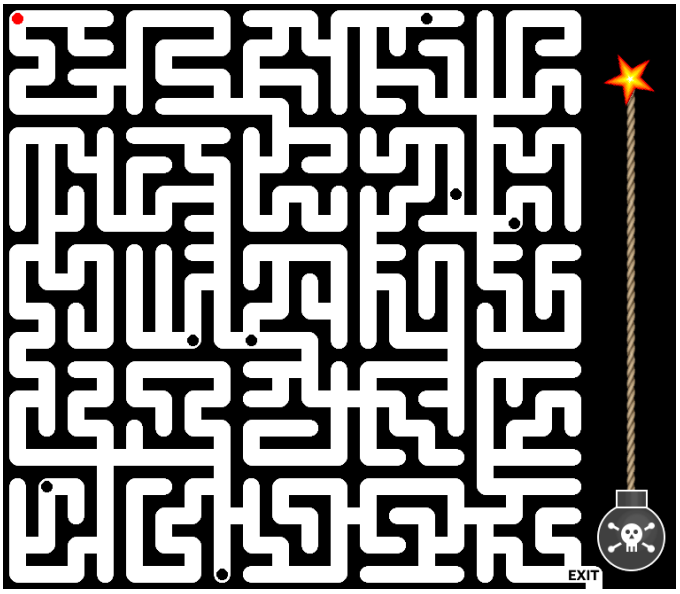


Figure 39: A level 7 maze (7 enemies, super-maze path length = 21).

4.2 Testing

After completing development of the game, it was presented to a sample of players to evaluate the efficacy of the DDA system and to compare the effects on players when the DDA system is used vs. using a linear difficulty.

4.2.1 Experiment Design

An experiment was required to collect data about how players felt during and after their experience playing the game.

As the DDA system always starts with the same initial difficulty, it takes a certain number of iterations for the system to adjust before finding the player's relative "moderate" difficulty (As demonstrated in Figure 35). During this adjustment period, the player may find mazes too easy or difficult. Because of this, it would be informative to question players about how frustrated or bored they felt playing the first few mazes and compare this to how they felt playing the following mazes.

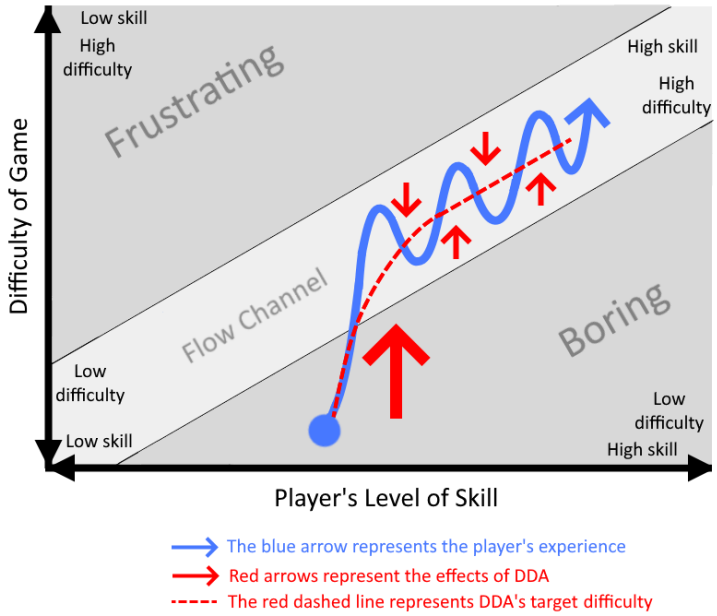


Figure 40: The DDA system starts at a fixed difficulty, so it must first push the player into the Flow channel, and then make minor adjustments to keep them there.

Furthermore, if the player is questioned about how difficult they thought the game was during the middle of the experience and the end of the experience, the consistency of the DDA system and the effectiveness over time could be fairly assessed.

To properly evaluate the effects of DDA, it is necessary to compare it with a linear difficulty. Players will be subjected to two trials, one in which the DDA system is active starting at difficulty level 5, and one in which the difficulty is always fixed at level 5. These trials will be taken in a random order, and the player will not be informed about which trial was using DDA until after the experiment is finished to reduce bias in the player's responses. Players will also be given a long break between both trials.

The system works to keep the player in the Flow channel, so it would be disruptive to question the player in between every maze; instead, the game will be paused after every 6

mazes and the player will fill in a short form. This will be repeated 3 times for a total of 18 mazes. They will be asked to rate how easy/difficult the game felt on a scale from 1 to 5. They will also be asked to comment briefly on how engaging, boring, or frustrating the experience was; as these are subjective emotional responses, it is more valuable to collect this data qualitatively. During the trial with DDA active, the current difficulty level of each maze is recorded by the game and stored in a text file.

Play tester: .....

On a scale of 1 to 5, how difficult are you finding the game? (tick one [✓])

Mazes	1 (easy)	2 (moderately easy)	3 (moderate)	4 (moderately hard)	5 (hard)
1 - 6					
7 - 12					
13 - 18					

How frustrating, boring or engaging are you finding the game and why?

Mazes	Comment
1 - 6	
7 - 12	
13 - 18	

Figure 41: A form for collecting data from players.

4.2.2 Results

Four players underwent the experiment, and their responses were collated and assessed. The data from the first trial (DDA active) is as follows:

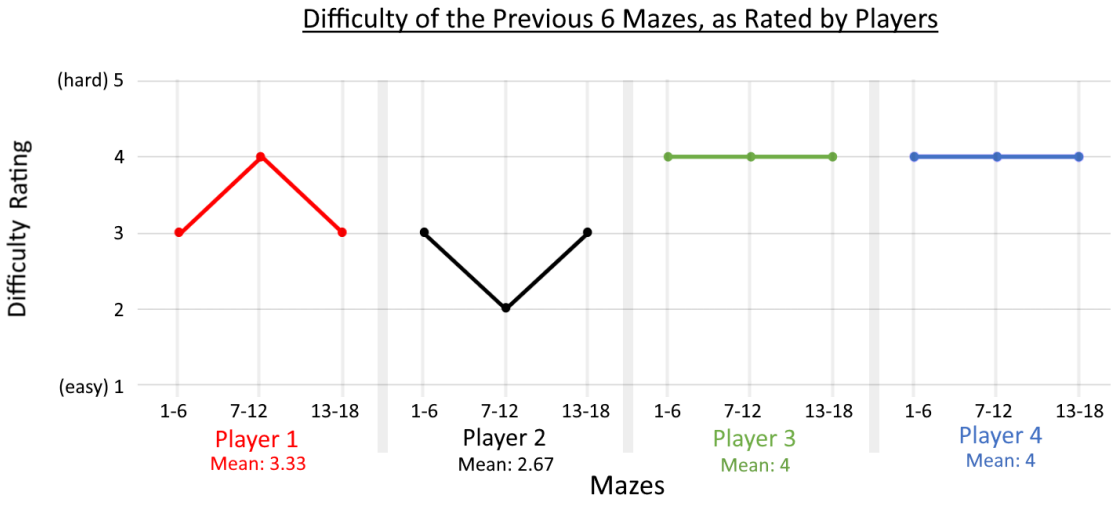


Figure 42: Results from the first question on the form: "On a scale of 1 to 5, how difficult are you finding the game?". DDA was active.

This graph shows the relative difficulty experienced by each player. Player 1 and Player 2 experienced a fluctuation in difficulty, while Player 3 and Player 4 felt that the game's difficulty remained constant throughout the experience.

It was expected that players may find the first set of mazes too difficult or easy due to the adjustment of DDA, followed by a consistently moderate difficulty. However, this conclusion cannot be drawn from the data, as each player rated the beginning and end of their experiences with the same difficulty rating. Dividing the test into smaller sets or increasing the length of the trial may have provided more data points to analyse how the system changed over time.

The data from the second trial (linear difficulty) is as follows:

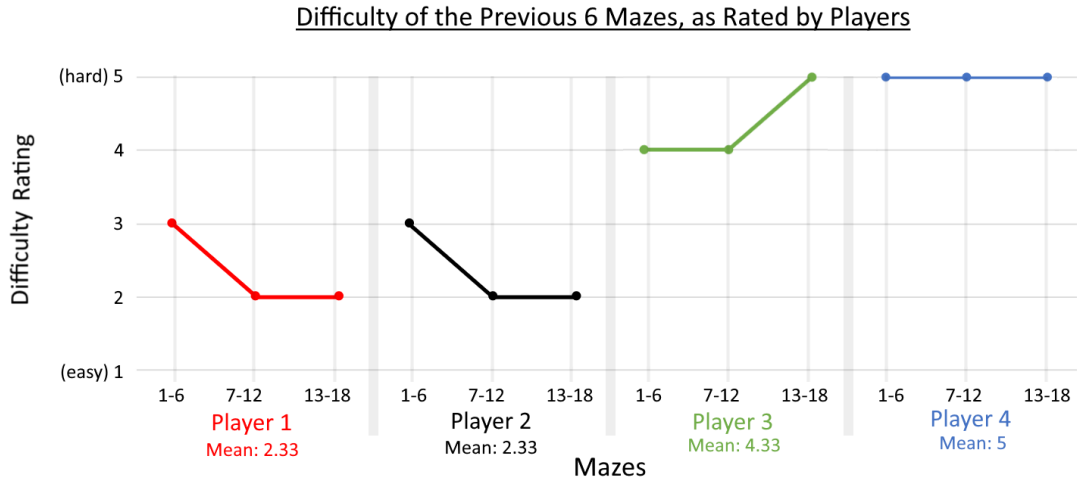


Figure 43: Results from the first question on the form. DDA was inactive.

Despite the difficulty level never changing, all but one player felt a change in perceived difficulty. This could be explained by a number of factors, including:

- the player's skill level increasing over time due to gaining experience, resulting in the game feeling easier,
- the player fatiguing and losing focus, resulting in the game feeling harder,
- the inherent variation in the difficulty of the mazes due to the randomness of the DFS algorithm,

or various other external factors, such as environmental distractions.

Comparing the datasets, every player's difficulty rating from the DDA trial was within the bounds of 2-4, whereas two players from the linear trial gave ratings of 5 (hard). Also, every player's mean difficulty rating was closer to 3 in the DDA trial than it was in the linear trial. This suggests that the DDA system is successful at keeping players closer to a moderate relative difficulty and that it prevents the game from becoming too hard or too easy for each player.

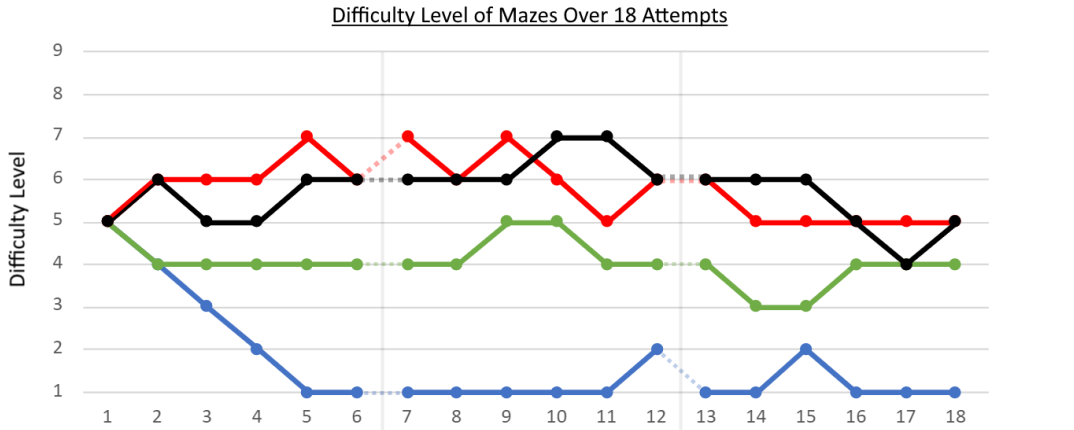
A general picture of the player's attitudes can be gathered by assessing the responses to question 2 in the form (Figure 41):

During the DDA trial, Player 1 and Player 2 both found the game to become more enjoyable as they played. This was because they felt that they were improving, but that the game was providing a consistent challenge. To begin with, Player 3 and Player 4 both found that the game was frustrating; Player 4 felt that the game became more rewarding as they played, while Player 3 felt that the game made them feel tenser as they played.

During the linear difficulty trial, Player 1 found the game enjoyable to start with but thought that it eventually became monotonous. Player 2 found the game less enjoyable overall compared to the DDA trial. Player 3 and 4 found the game consistently more frustrating, and Player 4 had become fatigued by the final set of mazes.

This data strongly suggests that the game was more engaging during the DDA trial.

The change in the game's difficulty level for each player during the DDA trial was as follows:



	Mazes 1-6		Mazes 7-12		Mazes 13-18		Overall
	Mean	S.D.	Mean	S.D.	Mean	S.D.	mean
Player 1	6	0.577	6.17	0.687	5.17	0.373	5.78
Player 2	5.5	0.5	6.33	0.471	5.33	0.745	5.72
Player 3	4.17	0.373	4.33	0.471	3.67	0.471	4.06
Player 4	2.67	1.49	1.17	0.373	1.17	0.373	1.67

Figure 44: How the difficulty of each maze changed throughout the DDA trial for each player.

From this data, it can be seen that Player 1 and Player 2 were both of a similarly high skill level, followed by Player 3, while Player 4 had the lowest skill level. Comparing this data to the graph shown in Figure 42, there seems to be a correlation between the player's skill level and their perceived difficulty; despite the DDA system's effects, higher-skilled players still found the game easier and lower-skilled players still found the game harder.

Overall, the DDA system was found to have a measurable positive effect on players' experiences, despite not perfectly adjusting the game to a relatively moderate difficulty.

Chapter 5
Conclusion

5.1 Review of Aims and Objectives

In section 1.2, six main objectives were set. To evaluate this project's success, each one of these objectives is reviewed:

1. ***Research maze generation algorithms and dynamic difficulty adjustment.***

   Chapter 2 reviews a range of subjects relating to engagement and video game difficulty, as well as the nature of mazes and methods used to generate them. Most of these concepts were further explored throughout the project and served as inspiration for the original solutions presented in Chapter 3.

2. ***Become familiar with Pygame.***

   By following online guides and resources, I learned how to create a fully functioning video game using the Pygame library. This includes constructing a central game loop [25], handling player input, drawing graphics to the screen and creating the internal logic of the game. This process was not detailed in the report due to its highly involved and off-topic nature, though it is evidenced by the final product and commented source code.

3. ***Create an algorithm for generating mazes of variable difficulty.***

   A variable-difficulty maze generation algorithm was designed and outlined in sections 3.1.3 and 3.2.1, and the implementation of the algorithm is detailed in section 3.3. Within the context of the game, this algorithm succeeds at generating mazes of a specified difficulty, as demonstrated in section 4.2.

4. ***Create an interface for the game itself.***

   The game's interface is presented in section 4.1. It features several quality of life elements [26], which are features that improve the "playability" of a game without changing the core gameplay. These include animations, sound effects and controls which make the game easier to understand and play. The final product is a polished and enjoyable game.

5. ***Experiment with DDA systems and expand on the game's features.***

   As detailed in section 3.4, various DDA systems were devised, tested, and modified iteratively. The theory behind them is explored using various conceptual models such as finite-state machines and Flow channel diagrams.

6. ***Collect data from players to investigate the efficacy of the DDA system.***

   In section 4.2, a randomised single-blind study was designed and conducted wherein the DDA system was compared to a linear difficulty. The results of this experiment suggest a positive correlation between the quality of the player's experience and the use of a DDA system, and demonstrate that the DDA system prevents players from finding the game too easy or too difficult.

Each objective served as a point of exploration for a wide range of possible solutions and ideas. Overall, I believe the project was very successful in meeting the objectives, and in delivering the product outlined in the project's aims.

5.2 Reflection

This has been my first attempt at undertaking a large academic project and entering the world of formal research. While at times exhausting, working on this project has been an invaluable and incredibly enjoyable experience.

Planning, designing, and implementing the game has given me a wealth of hands-on experience, and has introduced me to concepts and techniques that I will continue to employ in all of my future projects. These include the ability to understand and utilise research papers, effective project management, and a better grasp of video game development and programming in general.

One of the most valuable lessons I learned was to take project management seriously. While an initial project plan was outlined in section 1.3, this schedule was followed quite loosely which resulted in a lot of work piling up until close to the project's deadline. This is a recurrent problem of mine, and one I hope to mitigate in the future.

Some of the most valuable experience I gained was through using the Pygame library and solving problems specific to creating a video game; I have always aspired to learn the art of video game development. This project has provided me with the perfect motivation, and has pushed me to create something that I'm proud of.

5.3 Further Work

This project was of a very exploratory nature, so many of the concepts presented have potential for development and improvement.

One of these is the novel hierarchical maze technique described in section 3.1.3. This algorithm proved effective for the specific use case of the game, but it could also be employed in other contexts and modified in many ways. One shortcoming is described on

page 12, which occurs when the super-maze path length is too long. In these cases, adding alternate pathways (cycles) to the maze may increase their difficulty.

Several improvements could be made to the DDA system. As described in section 4.2.1, the system takes several iterations to initially adjust the difficulty level. This might be improved by adding a form of “acceleration” to the system’s adjustments so that larger differences in player skill and game difficulty can be adjusted in a smaller number of iterations. Incorporating adjustments to the time limit and the size of the mazes may also provide more control over the game’s difficulty.

5.4 Legal, Ethical, Social & Professional Issues

A dynamic difficulty adjustment system could be viewed as a manipulative tool used to keep the player engaged with the game for the maximum amount of time possible. This is similar to the concept of a compulsion loop, wherein a player is conditioned into repeating a series of activities in order to motivate them to continue playing, using positive reinforcement to trigger the release of dopamine in the brain. Such principles can lead to video game addiction and, in the case where real-world purchases are part of this loop, habitual spending [27].

One difference between a compulsion loop and a DDA system is that in the former, the game conditions the player to expect a certain “reward” upon completing a task. This is usually done with a variable-ratio reinforcement schedule [28], which is similar in principle to how a slot machine awards prizes. This expectation motivates the player to perpetuate the loop, and it becomes a compulsive activity. A DDA system does not “reward” the player for staying in certain loops, as the player does not directly control which loop they enter. This means the player does not associate a certain gameplay loop with a certain action, and the system does not condition them into acting compulsively.

Another difference is that a compulsion loop is usually employed with the express purpose of exploiting a player. It could be argued that a DDA only aims to regulate the difficulty of the game and thus maximise the enjoyment experienced by the player. A compulsion loop can result in a dependency that isn’t necessarily motivated by enjoyment, while a DDA system serves as a means of increasing the quality and accessibility of a game.

Despite this, an effective DDA system is usually hidden from the player, which in itself could be seen as an ethical concern. To address this, all players who partook in testing were informed about the nature of the DDA system and gave consent for their data to be collected and used anonymously.

All assets used in the game and figures featured in the report are original. All resources used to create the solution have been cited.

List of References

1. Hunicke R. The case for dynamic difficulty adjustment in games. In: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology [Internet]. New York, NY, USA: Association for Computing Machinery; 2005 [cited 2020 Nov 18]. p. 429–33. (ACE '05). Available from: https://doi.org/10.1145/1178477.1178573

2. Pygame website [Internet]. [cited 2020 Nov 9]. Available from: https://www.pygame.org/wiki/about

3. What is Agile Software Development? [Internet]. Agile Alliance |. 2015 [cited 2021 Jan 4]. Available from: https://www.agilealliance.org/agile101/

4. Waterfall Software Development Model | Oxagile [Internet]. 2014 [cited 2021 Jan 4]. Available from: https://www.oxagile.com/article/the-waterfall-model/

5. How to Keep Players in Their Flow Channel – What's in a Game? [Internet]. [cited 2020 Nov 21]. Available from: http://whats-in-a-game.com/controlling-flow/

6. Chen J. Flow in games (and everything else). Commun ACM. 2007 Apr;50(4):31–4.

7. Shakhova M, Zagarskikh A. Dynamic Difficulty Adjustment with a simplification ability using neuroevolution. Procedia Comput Sci. 2019 Jan 1;156:395–403.

8. Pearl J. Heuristics: Intelligent search strategies for computer problem solving. 1984 Jan 1 [cited 2021 Feb 4]; Available from: https://www.osti.gov/biblio/5127296

9. Pichlmair M, Johansen M. Designing Game Feel. A Survey. IEEE Trans Games. 2021;1–1.

10. Pac-Man - Videogame by Midway Manufacturing Co. [Internet]. [cited 2021 Feb 2]. Available from: https://www.arcade-museum.com/game_detail.php?game_id=10816

11. Finite State Machines | Brilliant Math & Science Wiki [Internet]. [cited 2021 May 9]. Available from: https://brilliant.org/wiki/finite-state-machines/

12. Karlsson A. Evaluation of the Complexity of Procedurally Generated Maze Algorithms.

13. Buckblog: Maze Generation: Recursive Backtracking [Internet]. [cited 2020 Dec 18]. Available from: https://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking

14. Kozlova A, Brown J, Reading E. Examination of Representational Expression in Maze Generation Algorithms. In 2015.

15. Buckblog: Maze Generation: Prim's Algorithm [Internet]. [cited 2020 Dec 18]. Available from: http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm

16. Buckblog: Maze Generation: Recursive Division [Internet]. [cited 2020 Dec 18]. Available from: http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm

16. Buckblog: Maze Generation: Aldous-Broder algorithm [Internet]. [cited 2020 Dec 19]. Available from: http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm

18. Kim PH, Grove J, Wurster S, Crawfis R. Design-centric maze generation. In: Proceedings of the 14th International Conference on the Foundations of Digital Games [Internet]. San Luis Obispo California USA: ACM; 2019 [cited 2020 Dec 18]. p. 1–9. Available from: https://dl.acm.org/doi/10.1145/3337722.3341854

19. Some Basic Theorems on Trees [Internet]. GeeksforGeeks. 2018 [cited 2021 May 9]. Available from: https://www.geeksforgeeks.org/some-theorems-on-trees/

20. Chakraborty M, Chowdhury S, Chakraborty J, Mehera R, Pal RK. Algorithms for generating all possible spanning trees of a simple undirected connected graph: an extensive review. Complex Intell Syst. 2019 Oct 1;5(3):265–81.

19. Weisstein EW. Adjacency Matrix [Internet]. Wolfram Research, Inc.; [cited 2021 Mar 2]. Available from: https://mathworld.wolfram.com/AdjacencyMatrix.html

22. Samet H. The Quadtree and Related Hierarchical Data Structures. ACM Comput Surv. 1984 Jun;16(2):187–260.

21. 15.3. The PR Quadtree — CS3 Data Structures & Algorithms [Internet]. [cited 2021 Mar 2]. Available from: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/PRquadtree.html

24. Liu X, Gong D. A comparative study of A-star algorithms for search and rescue in perfect maze. 2011 Apr 1;

25. Niemczyk R, Zawiślak S. Review of Maze Solving Algorithms for 2D Maze and Their Visualisation. In: Zawiślak S, Rysiński J, editors. Engineer of the XXI Century. Cham: Springer International Publishing; 2020. p. 239–52. (Mechanisms and Machine Science).

26. Conci A. Real time game loop models for single-player computer games. 2005.

27. Bycer J. Playability in Game Design [Internet]. Medium. 2019 [cited 2021 May 8]. Available from: https://superjumpmagazine.com/playability-in-game-design-310e94c4e88e

28. The Compulsion Loop Explained [Internet]. [cited 2020 Dec 18]. Available from: https://www.gamasutra.com/blogs/JosephKim/20140323/213728/The_Compulsion_Loop_Explained.php

29. Gatto J, Patrick M. Are Loot Boxes An Illegal Gambling Mechanic? :7.