School of Computing
FACULTY OF ENGINEERING AND PHYSICAL SCIENCE

UNIVERSITY OF LEEDS

Using AI to Solve Simon Tatham's Undead Game

Krishan Dipesh Patel

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

2019/20

40 credits

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Project Report	PDF	Minerva (30/04/20)
Code	Gitlab Repository	SSO (30/04/20)

Type of project: Exploratory Software

Summary

The purpose of this project is to create an artificial intelligence algorithm that is competent enough to solve the game of Undead in a reasonable time. The decision to choose Undead is because it differs to traditional puzzles of this type such (as Sudoku) and therefore adds an extra challenge.

Acknowledgements

The first person I would like to thank is my supervisor Dr. Brandon Bennett for the frequent meetings and support that he has provided throughout my project. I would also like to thank Dr. He Wang, my assessor, for the helpful feedback that was provided on my intermediate report.

Contents

Chapter 1

Introduction

1.1 Introduction

The study of solving puzzles using artificial intelligence (AI) has been of interest since the inception of AI. In particular the popular game of Sudoku has been studied extensively and many solvers for it have been created. The game of Sudoku comes under the category of puzzles called constraint satisfaction problems [17]. These are problems whose state must conform to and satisfy a number of limitations. The game of Undead, which will be the subject of this paper, is also a constraint satisfaction problem.

The study of games such as Sudoku has led to numerous successful research projects, which have discovered many techniques for solving puzzles using AI [12]. For example, you can model a problem as a graph and use a graph searching algorithm (such as depth first search (DFS)) in order to solve it. Other algorithms have been developed to look at how humans would solve the problems and then implement these more intuitive techniques algorithmically, so that an AI can use them to solve the problem. Sudoku has a number of features that help with its problem solving. One feature includes perfect information. This allows you to see the entire board at once and therefore know the complete state of the game [6]. As a result of this knowledge, you know how close you are to solving the puzzle.

Games such as Undead and Sudoku are considered to be NP-Complete [20]. This means that the problem has no one algorithm that can find a solution to that problem quickly, however, once a solution is found, the validity of the solution can be verified quickly. Therefore there are no known solutions for these problems that can be executed in polynomial time.

1.2 Aim of the project

The aim of the project is to create an AI that will read from a text file a series of different configurations for the Undead game and solve them in a time efficient manner. Currently, there is not a solution to this problem in the form of a dedicated solver, however there are websites and mobile apps of the game that can be played. The game should be scalable to a 5 x 5 board and be able to deal with any level of difficulty as long as the problem is solvable. This project will help advance my knowledge of the python programming language and improve my understanding of a number of different AI techniques that can be applied to other situations in the future.

1.3 Objective of the project

The overall objective of the project is set out below:

- Pursue and document my background research into the implementation of AI in games and puzzles

- Read in a string of characters from a text file and process these into a particular configuration of the Undead game.
- Form a path by iterating through adjacent squares and changing direction if a mirror is hit.
- Represent the board in a form that it can be solved in e.g. a list of paths
- Develop an AI algorithm to solve both 4x4 and 5x5 configurations of the Undead game
- Evaluate the performance of this AI on a number of different 4x4 and 5x5 boards.

Chapter 2

Background Research

2.1 Constraint Satisfaction

Constraint satisfaction is an area that is used in artificial intelligence when you need to make a number of decisions in parallel. It is used to model these decisions in terms of variables [10]. A constraint satisfaction problem is defined as such:

- A set of variables: $X_1, X_2, X_3, ..., X_n$
- Each variable X_i has a non-empty domain of possible values, e.g. $X_1 = [1, 2, 3, 4]$
- A set of constraints, these can be unary meaning they only involve one variable e.g. $X_1 \neq 3$, or binary meaning they can involve two variables e.g. $X_3 > X_4$.

A solution to a constraint satisfaction problem is found when each variable is assigned a value in its domain such that none of the constraints are broken [17].

2.2 Puzzle AI

In recent years, there have been many successes in the development of AI to solve various puzzles such as Sudoku, Karuko, 8 Queens and the 15 puzzle. These AIs are able to solve the problems mentioned faster and more efficiently than their human counterparts. This is because they can execute a greater number of computations in a fraction of the time. At the beginning of AI research, the study of problem solving was almost identical to the study of search algorithms [11]. The AI will follow an algorithm, such as depth first search (section 2.4.2), and simply execute that algorithm until the problem is solved. In contrast, humans take a more intuitive approach, normally using different techniques in order to eliminate possibilities and eventually reaching the correct solution. If a human attempted to execute a search algorithm by hand it would take much longer and it is more likely that they would make a mistake. These human techniques can also be implemented in conjunction with traditional AI techniques in order to eliminate certain states and therefore reduce the computation that the AI would have to do. This combination of techniques is how the Undead game will be solved in this paper.

2.2.1 8 Queens

The 8 Queens problem is a puzzle in which you must place 8 queens on a 8 x 8 chess board such that no two queens threaten each other [2]. A queen in chess is able to move anywhere along the horizontal, vertical or diagonal axis and is able to move in any direction. This means that you are unable to have two queens in the same row, column or diagonal. This problem can be expanded out into the n Queens problem in which you must place n queens on an n x n chess board such that no two queens threaten each other [9]. This puzzle is a classic constraint satisfaction puzzle as it has a clear list of constraints that have to be followed in order to find a solution.

This problem has been the subject of considerable research since it was introduced by chess composer Max Bezzel in 1848 and solved by Franz Nauck in 1850. There are a total of 92 solutions for the original 8 Queens problem, however, accounting for the fact that the same solution can be reflected on both the x and y axis or rotated, there are actually only 12 distinct solutions.

2.2.2 15 Puzzle

The 15 puzzle is a sliding puzzle that has 15 square tiles, numbered 1 - 15, on a 4x4 grid in a random order. This means that there is only one blank space. The aim of the puzzle is to get the tiles into ascending order starting from 1 in the top left corner.

Since the end state of the puzzle is known, this puzzle can be solved using a heuristic technique and this is how many AI researchers have solved the problem. Some common heuristics used for this problem are counting the number of misplaced tiles or finding the distance between its current position and its goal position. In particular, due to the problem being admissible (Section 2.4.4), the A* search algorithm has been used to ensure an optimal solution.

2.2.3 Peter Norvig Sudoku

Sudoku is a puzzle that consists of a 9x9 grid composed of 9 smaller 3x3 subgrids. The aim of the game is to fill in all 81 squares such that every row, column and subgrid contains the numbers 1-9 and none are repeated.

Peter Norvig created a famous Sudoku solver [12]. This used two main ideas, constraint propagation and search. The constraint propagation part of this solver uses two important strategies.

1. If a square has only one possible value, then eliminate that value from the squares peers
2. If a unit has only one possible place for a value then put the value there.

If one of the above strategies then solves a square, the board will be updated according to the new information and then these constraints will be applied again.

The other route that was taken was search. In this technique the program will systematically try all possibilities until it hits one that works. To do this Peter Norvig uses depth first search recursively so that if there is an issue with a particular search tree, the program can backtrack to try different possibilities rather than having to start from the beginning each time. This technique is integral to solving puzzles and other AI problems [12].

2.3 Game Theory

Game theory is the mathematical study of strategic decision-making and the outcomes that are produced due to the decisions made. It is used to describe the logical decision-making that occurs in humans and computers, particularly in relation to games and puzzles. However, it can also be expanded to other areas such as economics and psychology. John Von Neumann began the idea of modern game theory with his research into zero sum games.

Since this project will be focusing on a puzzle, rather than a game of multiple players competing against each other, the strategies used may be quite different.

2.3.1 Zero Sum / Non-Zero Sum Games

A zero sum game is a game in which the payoffs always sum to zero [5]. This means that a particular player can only have a positive pay-off if another player has a negative pay-off. For many games this can be determined by who wins the game, with a victory counting for a $+1$, a loss counting as a -1 and a draw giving both players 0. Therefore if someone wins the game then you have $1 + (-1) = 0$ hence zero sum.

Alternatively, a non-zero sum game does not require the gain of one player to be counter balanced by another player's shortcoming.

2.3.2 Perfect Information / Imperfect Information

A game of perfect information is one whereby every player has full awareness of the current game state as well as all previous game states [15]. An example of this is the game of chess: each player can see the full board and see every move that their opponent makes from the beginning of the game. A game of imperfect information is the opposite of this, such that each player does not know the full state of the game at any given time. A good example of this is one where both players play simultaneously such as rock paper scissors. This is because each player holds information and cannot know the information held by the other player.

2.4 AI Solving methods

2.4.1 Brute Force

Brute force is the simplest AI method for solving problems. The point of this method is to try every possible option until one is found that works. It is undertaken by trying one permutation of a solved state and then checking it against the constraints. If it satisfies all of the constraints then it is a valid solve. If not, simply move on to the next permutation and repeat this until one works.

This can either be done completely randomly, by picking random states or it can be done with a form of logic behind it, in order to track the permutation. This method is primarily used due to its simplicity to implement; it works well on small problems that do not require a great deal of computation. However, the problem with this method arises when the problems scale up. Generally, a brute force approach will take a lot of computational power to complete and therefore has a high time complexity. This means that as the problems get larger this technique will get slower until it is infeasible to use.

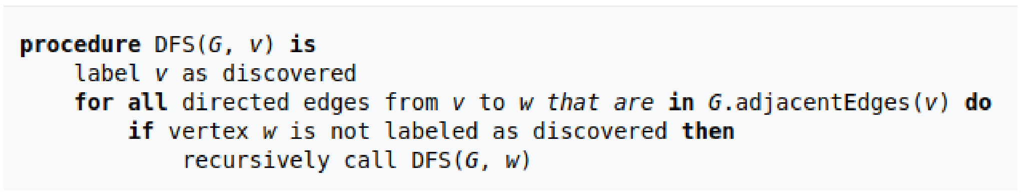
2.4.2 Depth-First Search

Depth-First Search (DFS) is a search algorithm that traverses graph or tree data structures [7]. The algorithm starts at the root node and traverses down a particular route until either completion or a rule is broken, at which point the algorithm begins backtracking back up the tree and tries other possibilities until there are no other options and the problem is solved. DFS uses a stack as the data structure which is last in, first out data structure allowing the algorithm to backtrack back through the graph by popping the most recent values off the stack.

One issue with DFS is that it could potentially never terminate. This will lead to an infinite loop and will crash any program that it is being used on. To rectify this you can give it a fixed depth that it can traverse down to in order to avoid overusing memory and disk space. DFS has a low time complexity, which is why it is very popular in terms of AI as it means that even as you scale problems up, the time to solve the problem will not increase significantly and the problem should still be solvable. The time complexity of DFS depends on the number of vertices and edges in the graph.

- Let V be the number of vertices in the graph
- Let E be the number of edges in the graph

The time complexity of DFS can then be written as $O(V + E)$ for explicit graphs that are traversed without repetition. However for implicit graphs with branching factor b that are searched to depth d the time complexity will be $O(b^d)$.



```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

Figure 2.1: DFS Pseudocode

2.4.3 Breadth-First Search

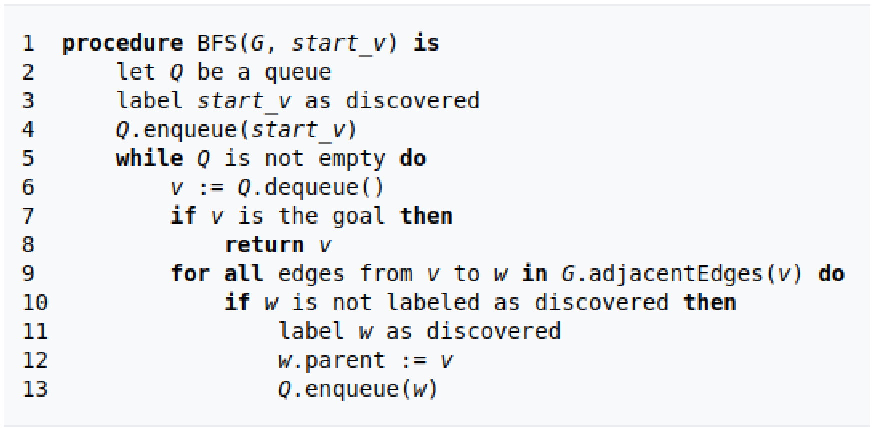
Breadth First Search (BFS) is a similar algorithm to DFS and is also used to traverse graph or tree data structures [7]. The difference between BFS and DFS is that, whereas DFS will traverse the entire way down a particular route, BFS will not proceed to the next depth level until it has expanded all of the nodes at the current level.

Another way BFS differs from DFS is that it uses a queue instead of a stack. A queue is a first in, first out data structure and this queue will store the frontier along which the algorithm is currently searching. Unlike DFS, BFS is complete. This is because BFS will always find a solution if one exists whereas DFS could get lost down a particular path in the graph that does not have a goal state and it will never return.[7]

2.4.4 Heuristic State

Heuristics are the principles and other pieces of commonsense that are applicable to computer systems, notably expert systems. They allow a reduction in the number of searches that need to be performed, in order to extract a solution [13].

The objective of a heuristic function is to find a solution to a problem in a reasonable time or to find an approximate solution to a problem if a solution cannot be found. It does not have to find the best solution, but it must find a good solution. They are used commonly in search problems in order to improve how optimal and how complete an existing search algorithm is. A



```
procedure BFS(G, start_v) is
    let Q be a queue
    label start_v as discovered
    Q.enqueue(start_v)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered then
                label w as discovered
                w.parent := v
                Q.enqueue(w)
```

Figure 2.2: BFS Pseudocode

heuristic function is said to be admissible if it never over-estimates the cost of reaching the goal state. The A* algorithm is a heuristic search algorithm that was created by adding a heuristic element to Dijkstra's algorithm for the purpose of guiding its search which leads to a better overall performance.

2.5 Undead Game

The Undead puzzle is a game which consists of a board made up of grid of squares, some of which contain diagonal mirrors [16]. All squares that do not contain mirrors must be filled with one of three types of monster below:

- Ghost
- Vampire
- Zombie

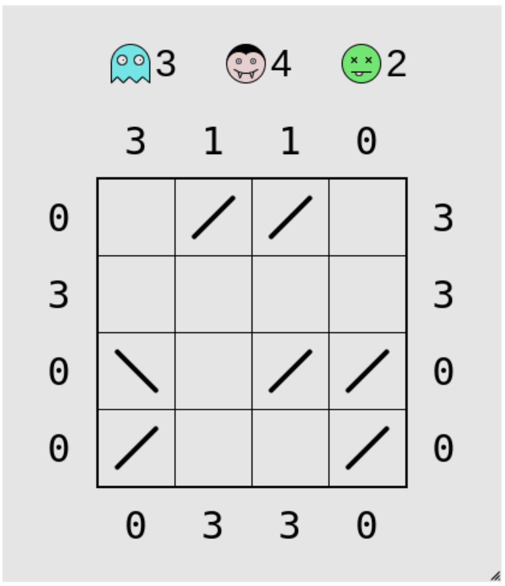


Figure 2.3: Undead Game

2.5.1 Rules

Before explaining the rules of the game it is important to understand the following definitions:

- Mirror: A mirror is represented by a diagonal line on the board and it can either be forwards or backwards. When viewing a path, if you hit a Mirror it will redirect you in another direction.
- Monster: One of Ghost, Vampire or Zombie
- Ghost: A Ghost cannot be seen directly but can only be seen in Mirrors
- Vampire: A Vampire can only be seen directly but not through a Mirror
- Zombie: A Zombie can be seen both directly and through a Mirror
- Path: A path is a series of squares from which you start from an outer square of the grid and traverse inwards into the board (bouncing off mirrors where necessary) until you exit the board
- Number Visible: Each path has a number visible associated with it. This number is the number of Monsters that can be seen from that position if you follow that path, subject to the rules for each Monster outlined above
- Sub Path: A sequence of squares that are a subset of a path.
- Zero Path: A path on which no Monsters are visible

For simplicity, the above definitions are incorporated into this report for its remainder. Now that the above definitions are clarified, the inner workings of the game can be investigated.

The board contains numbers along both ends of the rows and the columns, indicating how many Monsters you will see if you look straight (horizontally/vertically) into the board from that point [19]. A number is also allocated to each Monster displayed at the top of the board. These numbers indicate the maximum amount of times a Monster can be placed onto the board.

The game is solved when you can fill in the board without violating any of the constraints; meaning no more than the maximum amount given to each Monster is placed onto the board and every number around the outside accurately describes the number of Monsters seen from that point (remembering that Vampires can only be seen before mirrors and Ghosts can only be seen beyond a mirror).

2.5.2 Game Classification

Perfect information

The game of Undead can be considered a game of perfect information (as described in section 2.3.2). This is because it is a puzzle that is played by one person, therefore, they will always know every move that has been played. Furthermore, the player is able to see the entire board from the beginning of the game, with all information accessible to them. This allows the player to know the exact game state each time a monster is placed in a square.

Zero-Sum

The game of Undead can also be treated as a zero sum game (as described in section 2.3.1). In order to help solve the puzzle you can consider each path as a particular zero sum instance and use this to verify if it is valid. To calculate this you must take the number visible from each end of the path and subtract one from the other and take the absolute value. This can be represented as:

$$\text{Vis} = |\text{numVis1} - \text{numVis2}|$$

Looking at the path, consider **P1** to be the sub path before the first mirror hit and **P2** to be the sub path after the last mirror hit. For each of these sub paths, calculate their values subject to:

- Vampires are worth +1
- Ghosts are worth -1
- Zombies are worth 0

Finally **P** = |**P1 - P2**| from the other and if **P1** == **Vis** this is a valid path.

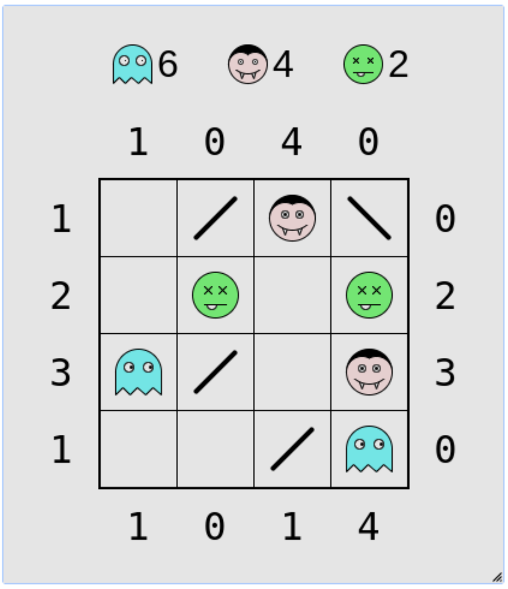


Figure 2.4: Zero Sum Example

If we look at figure 2.4 this may become clearer. In this example only one path is shown. For this path: numVis1 = 4, numVis2 = 3 therefore

$$\text{Vis} = |4 - 3| = 1$$

.

P1 is the sequence Ghost, Vampire, Zombie therefore

$$\text{P1} = 0 + 1 - 1 = 0$$

P2 is just a single Ghost therefore

$$\text{P2} = -1$$

.

$$\mathrm{P} = |0 - (-1)| = 1$$

Since **P** == **Vis** it is confirmed that this is a valid path.

2.5.3 Constraints

Undead can be represented as a constraint satisfaction problem. A constraint satisfaction problem can be described using three features [17]:

- The set of variables
- The set of domains
- The set of constraints that specify allowable combinations of values

In the case of the Undead game, the set of variables would be a set containing each square in the board. The set of domains would be the values that could go in each square e.g. $[g, v, z, \backslash, /]$. The set of constraints would contain all of the constraints on the board.

In this game there are two levels to the constraints. First, the global constraints which are the exact number of Ghosts, Vampires and Zombies that must be on the board. Once the board is filled, check that the number of Ghosts, Vampires and Zombies present on the board matches the number specified when the game was set up. The second level of constraints are the path level constraints. Each path has a number visible and for a path to be correctly filled in it must have the correct number of monsters visible subject to the rules of each monster, with regard to mirrors as specified in section 2.5.1.

2.5.4 Difficulty

When playing the game of Undead on Simon Tatham's website [18], you have the choice of playing different difficulties of games e.g. 4x4 Easy, 4x4, 5x5 Easy, 5x5. This raises the question: how is this difficulty determined? Rules on how the difficulties of the game are determined are not currently in existence, but there are some trends that can be observed. One simple observation is that easier puzzles will avoid long, twisty, reflected lines and therefore the maximum length of a path could be a metric of difficulty.

- Let l = Maximum length of a path
- Let z, g, v = the number of zombies, ghosts and vampires respectively.
- Let n = the size of the board (n=4 indicates a 4×4 board).
- Let s= the sum of all the numbers around the edges of the board.
- Let m = the number of mirrors on the board.

Ghosts and Vampires have two different states whereas Zombies only have one (as they can always be seen). Taking this into account, if the number of Ghosts and Vampires are increased and Zombies are decreased then the difficulty will further increase. Therefore as $\frac{(g+v)}{z}$ increases, the difficulty increases.

Another potential metric is to use the sum of the numbers around the board. This would not give a perfect answer but it would give a sense of how difficult the game would be. A 4x4 board where all paths can see 0 monsters would give a total of 0 and a board where all paths can see 4 monsters would give a total of 64. These would both be trivial to solve. This leads to on average: $\frac{64}{2} - |\frac{64}{2} - s| = 32 - |(32 - s)|$ or in a more general form $2n^2 - |(2n^2 - s)|$ [19].

This metric can also be adapted for counting the number of mirrors on the board. A board with no mirrors and a board filled with mirrors are both trivial to solve therefore we can have a metric such as $n^2 - |(n^2 - m)|$.

These different metrics can be combined to come up with one equation for the difficulty of the board [19].

- Let $w1, w2, w3, w4$ = weights of the four metrics
- Let d = The overall difficulty of the board

$$d = (w1 * l) + w2\frac{(g + v)}{z} + w3(2n^2 - |(2n^2 - s)|) + w4(n^2 - |(n^2 - m)|)$$

2.5.5 Techniques for solving

Fill in Zero Paths

In order to speed up all the solving techniques, a function can be implemented that pre-processes the board and fills in all of the **zero paths**. If a path has zero monsters visible then it can be instantly filled in. This is because it is known that ghosts are not visible before a mirror and Vampires are not visible after a mirror; this means that all of the squares can be filled in before the first mirror as Ghosts and all squares after the first mirror as Vampires. By pre-processing the board first, the speed of the algorithms can be greatly increased.

Brute force

One approach that could be used is to brute force the solution as described in section 2.4.1. This would mean trying every possibility until a solution is found. This solution would be very quick and easy to implement, however, this would have a high time complexity and running time, and therefore, not an optimal solution.

Zero sum calculations

Another way of solving the puzzle would be to only try paths that conform to the zero-sum calculations described in section 2.5.2. This would reduce the number of possible paths to try thus decreasing the time taken to solve the puzzle.

DFS square by square

A good way of solving the problem could be to conduct a depth first search square by square with backtracking until the solution is found. This should produce a much better solution than the brute force approach as there would be structure to the search and removal of values previously tried.

DFS Path by Path

A potentially better version of depth first search for this particular problem would be path by path. This would be possible if every possible path is generated and then a DFS is executed by filling in an entire path and then endeavoring to fill the next one until a solution is found. This solution would require more coding, however it should solve the problem more quickly as it is only necessary to check possible paths rather than every possible value and once a path is filled in, it eliminates many other possibilities.

DFS Path by Path with tightened constraints

A way to improve this DFS could be to eliminate other possible paths before doing the search in order to reduce the search space as much as possible. Since the end of one path is the start of another, it is futile to try each possible path from both ends. Henceforth, the search space can be halved if only one side of the path is used. Furthermore, by comparing all of the possible paths from one end of the path with all of the possible paths from the other end, the total number of possible paths are reduced, as only the paths that appear in both sets are valid. By tightening the constraints in this way, the search space will be significantly reduced, ergo, reducing the branching factor. This unique solution means that the algorithm will finish more quickly and will perform more efficiently. This follows the idea that Peter Norvig implemented in his Sudoku solver of constraint propagation and search [12]. Therefore, this will be the optimal algorithm implemented in the final solution of the problem in this report.

Chapter 3

Analysis and Design

3.1 Language

This project is written in Python. This language has been used for both the AI and for the command line interface. My experience in using Python was the primary reason I chose this language for the project. I have used Python for a number of modules throughout my degree as well as for personal projects. Ultimately, Python is the language I am most comfortable with and, therefore, the one with which I believe I could produce the best project.

I have similar experience with both C and Java, however, I prefer to use Python was as it has a much simpler syntax making the code more concise and easier to follow. C and Java are both much faster languages than Python however, for a 4x4 game of Undead, there are 16 possible squares and three monsters that could be put in each square. With the assumption that there is an average of 4 squares per game already filled with mirrors, then there are 3^{12} possibilities. Similarly, for a 5x5 game, there are 25 possible squares and with the assumption that there is an average of 7 mirrors per game, that leaves 18 squares with 3 possible monsters each. Therefore there are $3^{18} \approx 3.87 * 10^8$. Although this is a very large number, it is not impossibly large and therefore the lower level speed benefits of C or Java are not necessary. Moreover, despite Python being slower, it still lends itself to comparing the various different solving algorithms explored in this project.

The advantage of using Python is that it is portable, a high - level programming language and it can be extensible in other programming languages if the situation requires it. Furthermore, it is modular which helps the user perform different tasks and combine them with ease.

3.2 Program Flow

In order to make the program modular and easy to follow, it is split into distinct functions, each with their own parameters so that their use is simple and each function only performs one task. This will be helpful in the future, if creating a similar program, as there will already be written functions to work from allowing the algorithms used to potentially be applied to different puzzles.

To facilitate the comparison of how well different solving methods find a solution and how efficiently they work, the running of the program is split into 4 distinct stages in order to allow solving algorithm to be interchanged easily.

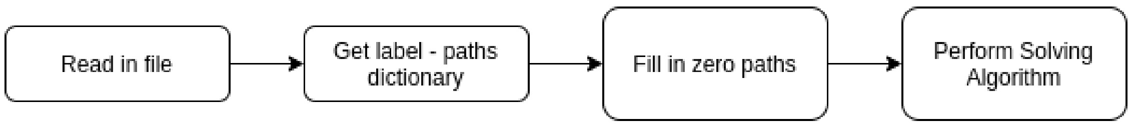


Figure 3.1: Program Flow

3.2.1 Reading from the file

Using the definitions in section 2.5.1 we can define a configuration of a particular game of Undead using the following:

- Three integers that describe how many of each monster are allowed on the board
- A board that is able to contain monsters and mirrors
- Integers surrounding all sides of the board describing how many monsters are visible on that particular path

The first process that occurs when the program is run is the board being read in from a text file. The board is represented as a single string of characters and once it is read into the program, it is processed into multiple data structures that represent the particular configuration of the game that can be used by the rest of the program. These data structures consist of:

- A matrix in the form of a 2D array to represent the board
- A string of numbers to represent the number of monsters visible down each path
- An integer to represent how many Ghosts are visible on the board
- An integer to represent how many Vampires are visible on the board
- An integer to represent how many Zombies are visible on the board

These data structures should be returned back to the main function.

3.2.2 Get Visible Monsters - Paths Dictionary

The matrix and the string of numbers discussed in the previous section are combined in a dictionary to form a new data structure that is used to check if the board has been solved correctly. A dictionary is used for this because the average cost of each look up is independent of the number of items that it holds therefore it has O(1) look up time which makes the solution more efficient. Unfortunately, the problem arises such that multiple paths could have the same value or the same number of monsters visible; this means that neither of these can be a unique key for the dictionary. To resolve this, the program labels each path with a letter and a number. The letter in the label of each path indicates the direction that it will start in, so R1 is the path starting from the top left square as it will be travelling in the right direction (figure 3.2). Two dictionaries are used:

- One to hold the label and the path e.g. ["R1" : "., ., /, ."] with a full stop representing a blank square and a slash representing a mirror
- One to hold the label and the number of monsters visible on that path e.g. ["R1": 2]

This ensures that the program has a quick way of finding how many monsters should be visible from a particular path.

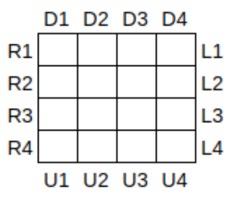


Figure 3.2: Board with Path labels

3.2.3 Fill in Zero Paths

The next step in the program is to fill in the paths where no monsters are visible (section 2.5.5). This is a quick process to carry out and reduces the number of unknown squares thereby reducing the computation of solving the entire board. If a path has no visible monsters, then due to the constraints on the monsters outlined in section 2.5.1, any blank square before the first mirror is hit must be a Ghost and any blank square after the first mirror is hit must be a Vampire. We can find these paths using the two data structures outlined in the previous step.

3.2.4 Perform Solving Algorithm

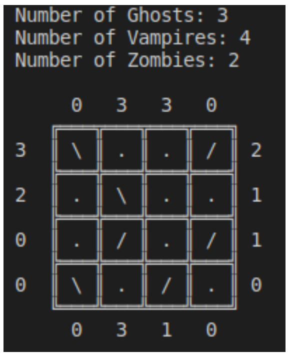
At this stage in the program, the board is partially complete from filling in the zero paths but the remaining squares still need to be solved. The partially solved board is then passed to the solving algorithm (e.g. DFS, brute force) and the state of the board is checked against the constraints (Section 2.5.3) after each step that the solving algorithm performs. If all of the constraints are satisfied then, the solved board is returned back to the main function and the program is complete.

Whilst solving this problem, a number of different techniques and algorithms were used in various iterations of the program in the search of finding the optimal solution. Although these algorithms are different, they are able to be interchanged easily without having to alter any of the rest of the program. This is undertaken by each algorithm taking in a 2D array that represents the board as its only parameter and subsequently calling other functions in the program in order to solve the puzzle.

3.3 Interface

Due to the nature of the program, the user will not need to interact with it once it is run. To display results to the user, the program will use a command line interface. This interface will display the unsolved board as well as the solved board and the time taken to solve it.

Originally, the intention was to use a graphical user interface (GUI), however, due to the time constraints it made more sense to opt for a command line interface. A GUI would have required more resources and taken longer to run which would have made the program less efficient; since this paper is evaluating different methods of solving this puzzle it is important that the program is as efficient as possible. Instead of printing out the board as a 2D array, a more user-friendly interface (figure 3.3) was created which mimics the original game (figure 2.3). This means that it can be directly compared to the original game in order to check for errors more efficiently and to give the user a better experience. Unicode characters were used to create a blank board and thereafter, all of the values were filled in.



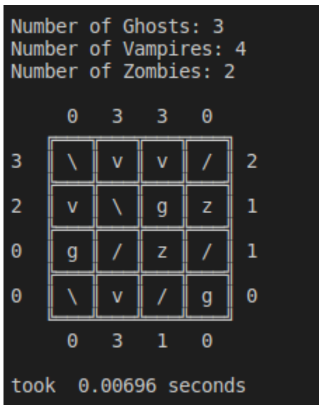
```
Number of Ghosts: 3
Number of Vampires: 4
Number of Zombies: 2

     0   3   3   0

3  | \ | . | . | / |  2
2  | . | \ | . | . |  1
0  | . | / | . | / |  1
0  | \ | . | / | . |  0

     0   3   1   0
```

(a) Unsolved Board



```
Number of Ghosts: 3
Number of Vampires: 4
Number of Zombies: 2

     0   3   3   0

3  | \ | v | v | / |  2
2  | v | \ | g | z |  1
0  | g | / | z | / |  1
0  | \ | v | / | g |  0

     0   3   1   0

took  0.00696 seconds
```

(b) Solved Board

Figure 3.3: Command Line Interface

Chapter 4

Planning

4.1 Initial Planning

The project will be split into four main areas:

- Report writing: a comprehensive report which introduces the concept of AI, the project that I am pursuing and detailed documentation in relation to the project
- Setting up the game: implementation of a program that will read in a configuration of the game of Undead and parse it into a form where the AI can work on it.
- Implementing the AI: implementation of the AI to solve the games of Undead provided in a timely manner and allow for scalability.
- Testing: Inclusion of testing to make sure any bugs are removed and the game is able to run well. This will include proof and documentation of testing.

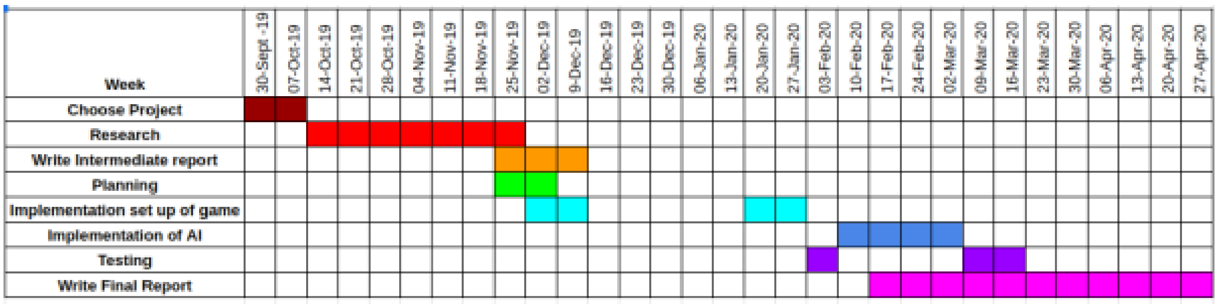
Based on the above the report will be updated to include details of the development of the project. This will ensure complete clarity in relation to the functioning of the game and how the game has developed to the final product.

4.2 Overall project methodology

A waterfall based approach [3] has been chosen for this project. This is a rigid method which consists of taking clear steps in pursuit of the final product. These steps include analysis, design, coding, testing, implementation. The analysis and design phases are described in the first three chapters of this report. Although the agile method [1] method is the method primarily used in industry, I believe this method is best utilised for group projects and for products that need to be delivered quickly. However, for this project I will be working alone and have been given ample time, therefore I believe that the waterfall method will be the most beneficial in organising my project workflow. Furthermore, for this project I am required to produce documentation in the form of a report and the waterfall method enables this reporting format, rather than the agile approach. Therefore, in light of this I have selected the waterfall project methodology.

4.3 Timeline

I began this project on the 30th September 2019. The first two weeks of the project were used to choose which project I was going to complete and finalise any details regarding this. Following this, I began my research to enable me to write the intermediary report. In the remainder of the first semester (the weeks consisting of 30th September to 16th December) I began to explore possibilities for implementation of the project as well as writing the intermediary report. At the beginning of the second semester (the week starting the 20th January) I began full implementation of the project and started to write my final report.



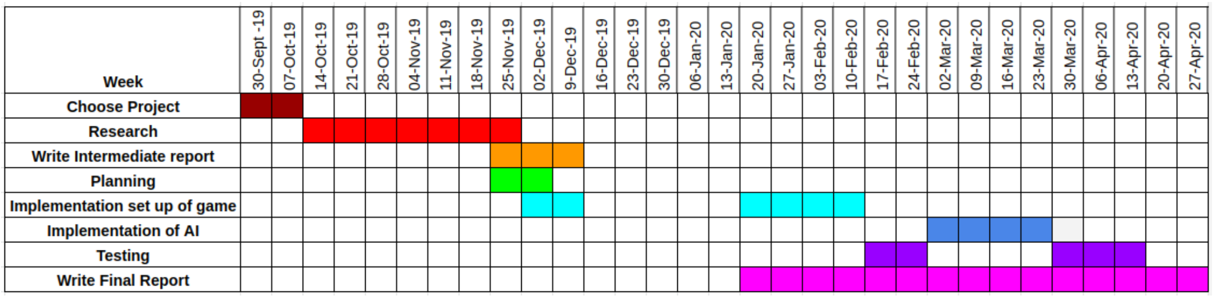
Week	30-Sept -19	07-Oct-19	14-Oct-19	21-Oct-19	28-Oct-19	04-Nov-19	11-Nov-19	18-Nov-19	25-Nov-19	02-Dec-19	9-Dec-19	16-Dec-19	23-Dec-19	30-Dec-19	06-Jan-20	13-Jan-20	20-Jan-20	27-Jan-20	03-Feb-20	10-Feb-20	17-Feb-20	24-Feb-20	02-Mar-20	09-Mar-20	16-Mar-20	23-Mar-20	30-Mar-20	06-Apr-20	13-Apr-20	20-Apr-20	27-Apr-20
Choose Project	■	■																													
Research			■	■	■	■	■	■	■																						
Write Intermediate report									■	■	■																				
Planning									■	■																					
Implementation set up of game										■	■						■	■													
Implementation of AI																				■	■	■	■								
Testing																			■					■	■						
Write Final Report																					■	■	■	■	■	■	■	■	■	■	■

Figure 4.1: Initial Gantt Chart

4.4 Risk Mitigation

This plan was designed such that all of the research and design will be completed in the first semester, because both steps must be completed, before the coding can begin. This provides plenty of time in the second semester for the coding of the project to be the main focus. The implementation of the game was expected to be completed by January 27th 2020, when teaching began for the second semester, because workload at this time gave me the opportunity to focus on this project. I was aware that from the week of March 9th 2020, other modules had coursework due which would demand a large proportion of my time; I wanted all coding to be complete by the week of March 9th so that the report would be my sole focus for the remainder of the time.

4.5 Revised plan



Week	30-Sept -19	07-Oct-19	14-Oct-19	21-Oct-19	28-Oct-19	04-Nov-19	11-Nov-19	18-Nov-19	25-Nov-19	02-Dec-19	9-Dec-19	16-Dec-19	23-Dec-19	30-Dec-19	06-Jan-20	13-Jan-20	20-Jan-20	27-Jan-20	03-Feb-20	10-Feb-20	17-Feb-20	24-Feb-20	02-Mar-20	09-Mar-20	16-Mar-20	23-Mar-20	30-Mar-20	06-Apr-20	13-Apr-20	20-Apr-20	27-Apr-20
Choose Project	■	■																													
Research			■	■	■	■	■	■	■																						
Write Intermediate report									■	■	■																				
Planning									■	■																					
Implementation set up of game										■	■						■	■	■	■											
Implementation of AI																							■	■	■	■					
Testing																					■	■					■	■	■		
Write Final Report																	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

Figure 4.2: Revised Gantt Chart

The general guide-line that the Gantt chart provides was followed closely and assisted in keeping the work on schedule; having distinct sections allowed me to focus on only one task at a time. This worked well during the first semester as the correct work was undertaken every week as planned. However, once the coding began some alterations to the plan had to be made, as implementing the set-up of the game and processing the board into data structures proved to be more difficult than anticipated. Furthermore, it took an additional two weeks to debug all of the issues and get a working solution with a brute force method. Therefore, I decided to revise the plan and began writing the report much earlier, concurrently with the code, giving me more time to edit the report and allow for more iterations. Furthermore, an extra week of testing was required in the week of April 13th as the effectiveness of the final solution needed to be compared with the other AI solutions that were created throughout the semester. Despite these changes, the Gantt chart has been an effective tool and enabled me to be on track throughout my project.

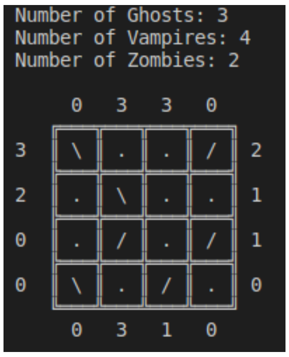
Chapter 5

Making the AI

This section is focused on the AI of the game and how it was used to achieve the objectives of the project. This section will also discuss the set-up of the game and how it represents the state of the game at each stage in order for the AI to perform.

5.1 Game State Representation

To allow representation of the state of the game, the program must first be provided with some data in order to form the initial state. This data is given in the form of a text file with a string of characters that represent the entire board and are stored in data structures as described in section 3.2.1.



```
Number of Ghosts: 3
Number of Vampires: 4
Number of Zombies: 2

    0   3   3   0
3   \   .   .   /   2
2   .   \   .   .   1
0   .   /   .   /   1
0   \   .   /   .   0
    0   3   1   0
```

Figure 5.1: Board to be represented

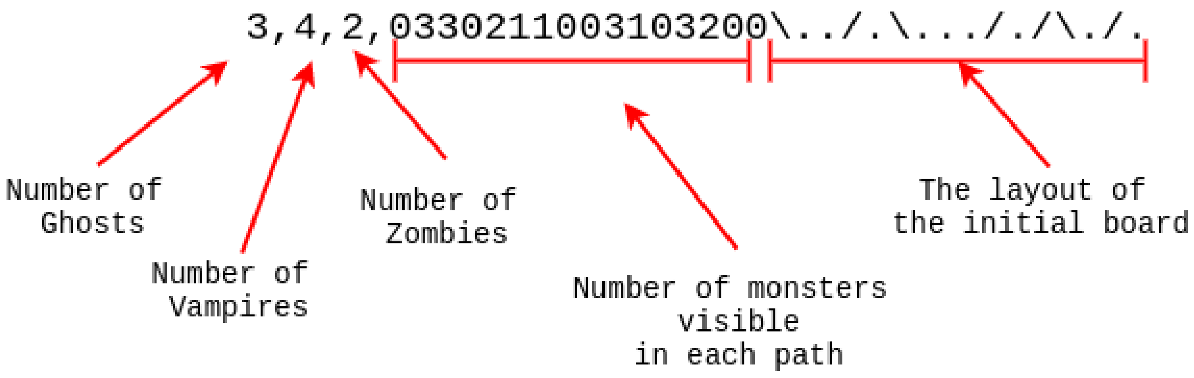
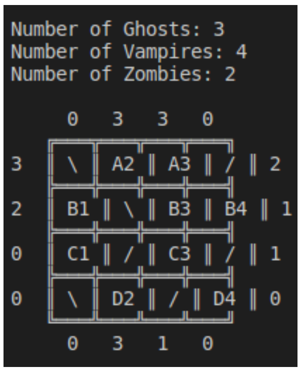


Figure 5.2: Text Representation

This will present the board in the structure of a 2D matrix and from this matrix we need to be able to trace the different paths.This is more challenging than initially expected as the path cannot be found by iterating through in one direction due to the fact that mirrors on the board cause changes in direction. Initially, each square in the board needs to be able to be labelled so that it can be uniquely identified. This is done by assigning a number to each column and a letter to each row. For the above board, the equivalent labelled board would look like this:

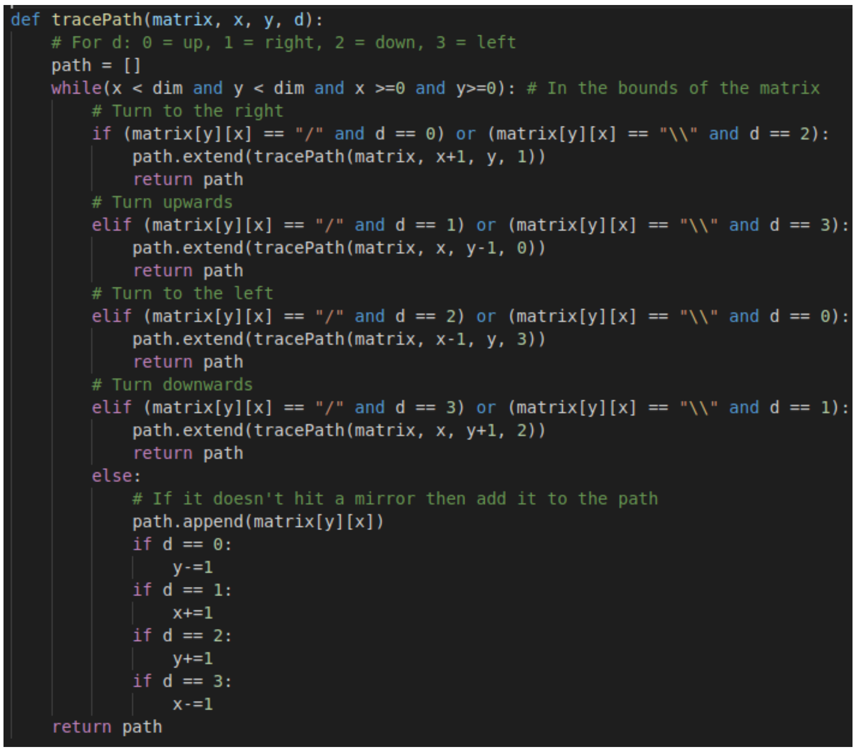


```
Number of Ghosts: 3
Number of Vampires: 4
Number of Zombies: 2

     0   3   3   0
   ╔═══╦═══╦═══╦═══╗
3  ║ \ ║ A2 ║ A3 ║ / ║ 2
   ╠═══╬═══╬═══╬═══╣
2  ║ B1 ║ \ ║ B3 ║ B4 ║ 1
   ╠═══╬═══╬═══╬═══╣
0  ║ C1 ║ / ║ C3 ║ / ║ 1
   ╠═══╬═══╬═══╬═══╣
0  ║ \ ║ D2 ║ / ║ D4 ║ 0
   ╚═══╩═══╩═══╩═══╝
     0   3   1   0
```

Figure 5.3: Labelled Board

To implement the tracing of the paths, a recursive function is applied that starts at a particular edge square, represented by its x and y coordinate, with a particular direction where 0 = up, 1 = right, 2 = down, 3 = left. If a mirror is hit on this path, the function will be recursively called with a new direction depending on which direction the mirror is facing and the x and y positions of the current square. Once the path has exited the matrix from another square, the path is then returned from the function.

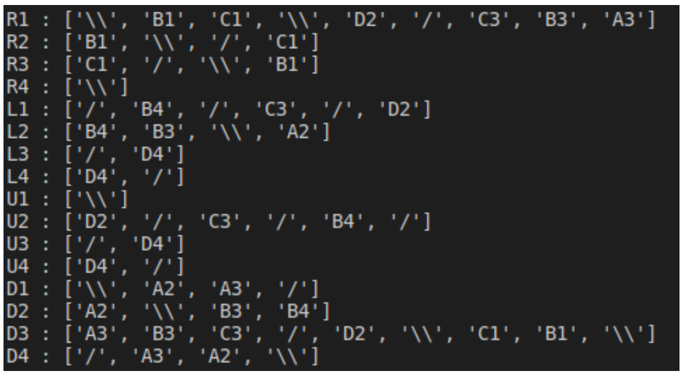


```python
def tracePath(matrix, x, y, d):
    # For d: 0 = up, 1 = right, 2 = down, 3 = left
    path = []
    while(x < dim and y < dim and x >=0 and y>=0): # In the bounds of the matrix
        # Turn to the right
        if (matrix[y][x] == "/" and d == 0) or (matrix[y][x] == "\\" and d == 2):
            path.extend(tracePath(matrix, x+1, y, 1))
            return path
        # Turn upwards
        elif (matrix[y][x] == "/" and d == 1) or (matrix[y][x] == "\\" and d == 3):
            path.extend(tracePath(matrix, x, y-1, 0))
            return path
        # Turn to the left
        elif (matrix[y][x] == "/" and d == 2) or (matrix[y][x] == "\\" and d == 0):
            path.extend(tracePath(matrix, x-1, y, 3))
            return path
        # Turn downwards
        elif (matrix[y][x] == "/" and d == 3) or (matrix[y][x] == "\\" and d == 1):
            path.extend(tracePath(matrix, x, y+1, 2))
            return path
        else:
            # If it doesn't hit a mirror then add it to the path
            path.append(matrix[y][x])
            if d == 0:
                y-=1
            if d == 1:
                x+=1
            if d == 2:
                y+=1
            if d == 3:
                x-=1
    return path
```

Figure 5.4: Python Code for tracing a path

This path is stored in a dictionary with the label of the path as the key as described in section 3.2.2. For the board in figure 5.3, the dictionary would be as shown in figure 5.5. Backwards facing mirrors are represented as "\\" instead of "\" as one backslash is an escape character; to process it two backslashes were required.

As described in section 3.2.1, a dictionary is needed that pairs up the labels for these paths with the numbers of monsters visible on said path. This is done by iterating through the numbers given with the input and matching them up with the correct labels.



```
R1 : ['\\', 'B1', 'C1', '\\', 'D2', '/', 'C3', 'B3', 'A3']
R2 : ['B1', '\\', '/', 'C1']
R3 : ['C1', '/', '\\', 'B1']
R4 : ['\\']
L1 : ['/', 'B4', '/', 'C3', '/', 'D2']
L2 : ['B4', 'B3', '\\', 'A2']
L3 : ['/', 'D4']
L4 : ['D4', '/']
U1 : ['\\']
U2 : ['D2', '/', 'C3', '/', 'B4', '/']
U3 : ['/', 'D4']
U4 : ['D4', '/']
D1 : ['\\', 'A2', 'A3', '/']
D2 : ['A2', '\\', 'B3', 'B4']
D3 : ['A3', 'B3', 'C3', '/', 'D2', '\\', 'C1', 'B1', '\\']
D4 : ['/', 'A3', 'A2', '\\']
```

Figure 5.5: Label Path Dictionary

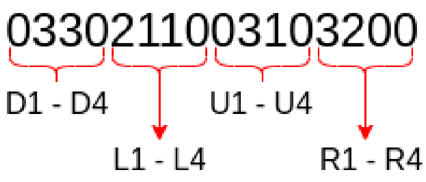
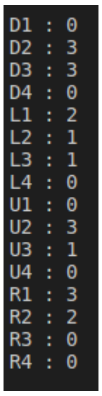


Figure 5.6: Label - Number of Monsters Visible Text

Once the game is fully represented in the data structures shown above, the coding of the AI of the project can begin.

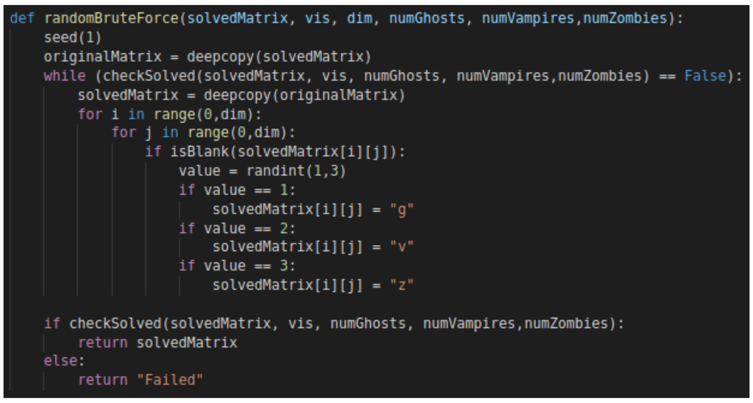
5.2 Basic Strategy - Random Brute Force

Before beginning the AI, an initial naive approach in the form of a random brute force was implemented. As this method can be implemented very quickly and easily once the game is set up, the result was that there was at least one solving method working. This solution allowed functions to be written that facilitated the verification that the final board was solved; the rest of the program could also be tested to ensure all of the data structures were being created and processed correctly before the implementation of more advanced strategies. Moreover, this created a baseline performance that permitted measuring the effectiveness of future solutions by comparing the times taken to solve the same puzzles. This was implemented by doing a while loop that checked if the board was solved on each iteration. Inside of this while loop, 2 for loops are contained in order to iterate through every square on the board, then for each square a random number from 1-3 was generated. If a 1 is produced then the square is filled with a Ghost, likewise if a 2 is produced then the square is filled with a Vampire and if a 3 is produced then the square is filled with a Zombie. Every square is filled in this way until the end of the matrix is reached, at which point it is checked. If it is solved, the matrix is returned and if not, the matrix is reset to its original state and the loop restarts.



```
D1 : 0
D2 : 3
D3 : 3
D4 : 0
L1 : 2
L2 : 1
L3 : 1
L4 : 0
U1 : 0
U2 : 3
U3 : 1
U4 : 0
R1 : 3
R2 : 2
R3 : 0
R4 : 0
```

Figure 5.7: Label - Number of Monsters Visible Dictionary



```python
def randomBruteForce(solvedMatrix, vis, dim, numGhosts, numVampires,numZombies):
    seed(1)
    originalMatrix = deepcopy(solvedMatrix)
    while (checkSolved(solvedMatrix, vis, numGhosts, numVampires,numZombies) == False):
        solvedMatrix = deepcopy(originalMatrix)
        for i in range(0,dim):
            for j in range(0,dim):
                if isBlank(solvedMatrix[i][j]):
                    value = randint(1,3)
                    if value == 1:
                        solvedMatrix[i][j] = "g"
                    if value == 2:
                        solvedMatrix[i][j] = "v"
                    if value == 3:
                        solvedMatrix[i][j] = "z"

    if checkSolved(solvedMatrix, vis, numGhosts, numVampires,numZombies):
        return solvedMatrix
    else:
        return "Failed"
```

Figure 5.8: Python code for random brute force solver

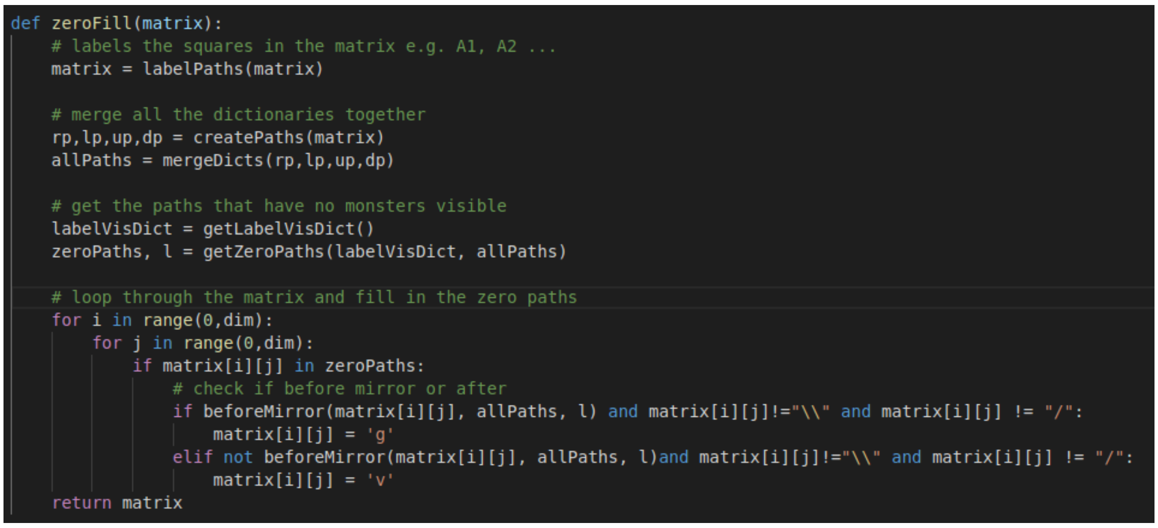
5.3 Advanced Strategies

5.3.1 Fill in Zero Paths

The first advanced strategy applied was to fill in the paths that had no monsters visible as discussed in section 3.2.3. This step is completed before any search begins and it pre-processes the matrix in order to reduce the number of possible solutions for each path so that the final solution can be obtained more quickly.

5.3.2 DFS Square by Square

The first form of DFS implemented was to search the board square by square, recursively trying the different possible values until a possible solution was found. To do this, a list entitled "choices" was created that contained the three possible monsters: choices = [g,v,z] representing a Ghost, Vampire and Zombie respectively. The program would then loop through these choices and fill one in, once it was filled in the matrix was checked to see if any of the constraints mentioned in section 2.5.3 were violated. If a constraint was violated, hence it could not be a possible path, then the next monster would be tried. If no constraint was violated then a recursive call would be made with the new state of the board. If all three choices were tried and each one violated a constraint then the function would return False and move up the recursive tree and move a square back and try the next value for that square. If this occurs all the way back to the first square and all the possibilities for that square are tried, then a solution could



```python
def zeroFill(matrix):
    # labels the squares in the matrix e.g. A1, A2 ...
    matrix = labelPaths(matrix)

    # merge all the dictionaries together
    rp,lp,up,dp = createPaths(matrix)
    allPaths = mergeDicts(rp,lp,up,dp)

    # get the paths that have no monsters visible
    labelVisDict = getLabelVisDict()
    zeroPaths, l = getZeroPaths(labelVisDict, allPaths)

    # loop through the matrix and fill in the zero paths
    for i in range(0,dim):
        for j in range(0,dim):
            if matrix[i][j] in zeroPaths:
                # check if before mirror or after
                if beforeMirror(matrix[i][j], allPaths, l) and matrix[i][j]!="\\" and matrix[i][j] != "/":
                    matrix[i][j] = 'g'
                elif not beforeMirror(matrix[i][j], allPaths, l)and matrix[i][j]!="\\" and matrix[i][j] != "/":
                    matrix[i][j] = 'v'
    return matrix
```

Figure 5.9: Python code for filling in paths where zero monsters are visible

not be found.

This solution worked faster than the brute force for some puzzles, however since the branching factor was so large it failed to find a solution more often than the brute force. This indicated that it was not a viable solution for the problem.

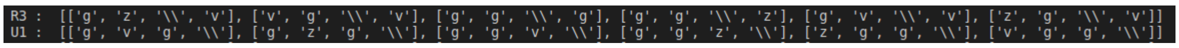
5.3.3 DFS Path by Path

The next stage to obtaining the solution was to try DFS path by path. Since the constraints on the puzzle are actually on each path, rather than each square, a unique solution was found that solved the problem in two steps:

1. Generate all the possible paths
2. Do a DFS on the possible paths

Each path not entirely filled in had every possible value of that path generated, for example a path that was [., /, .] has two blank squares and 3 possible monsters for each square which gives $3^2 = 9$ possible paths. These possible paths are stored as a list of lists. The number of possible paths can be reduced by checking the generated paths against the dictionary that stores the label of the path and the number of monsters visible (section 5.1) and only adding the path to the list if it has the correct number of monsters visible. Once all the possible paths have been generated, this list of lists is stored in a dictionary called "possPathsDict" with the path label as the key.

An example of such a data structure is shown in figure 5.10.



```
R3 :  [['g', 'z', '\\', 'v'], ['v', 'g', '\\', 'v'], ['g', 'g', '\\', 'g'], ['g', 'g', '\\', 'z'], ['g', 'v', '\\', 'v'], ['z', 'g', '\\', 'v']]
U1 :  [['g', 'v', 'g', '\\'], ['g', 'z', 'g', '\\'], ['g', 'g', 'v', '\\'], ['g', 'g', 'z', '\\'], ['z', 'g', 'g', '\\'], ['v', 'g', 'g', '\\']]
```

Figure 5.10: Possible Paths Dictionary

Once all of the possible paths are found, the dictionary can be sorted such that the path with the fewest possibilities is at the start of the dictionary. This means that if R1 has 2 possible paths and L4 has 4 possible paths then R1 will be before L4 in the dictionary. This order was chosen because it means that when the search is performed, the correct values can be found more quickly as when further down the list, where there are more possibilities, more of the board will have been filled in. This will mean that there are actually fewer possibilities to try than previously thought.

Once the "possPathsDict" dictionary is formed, the search begins. At this stage, it is very similar to the strategy discussed in section 5.3.2 and the recursion happens in a similar way. The function will first check if the board is solved and if it is, it will then return the board, otherwise it will take the first value of the possPathsDict and try to fill the first path in the list. If this is a valid path, the function will make a recursive call with the new state of the board and possPathsDict with the first element removed. If the path is not valid then the next path in the list of lists will be tried until there are no other possibilities. If this occurs, it will return False and move up the recursive tree. If the highest level of the recursive tree returns false then no solution could be found.

This method works very quickly and provides solutions to all 4x4 puzzles tried and most 5x5 puzzles. This means that it could be a viable solution, however, there were ways in which it could still be enhanced.

5.3.4 DFS Path by Path with tightened constraints

To further optimise the algorithm, it was appreciated that each path had an exit point and this exit point was the start point for another path. For example in figure 5.11 paths R1 and U3 demonstrate this. This meant that it was only necessary to check one of these paths and the other would be filled in. In doing this, the amount of computation that needed to be done was halved, and therefore the program was much more efficient. The system used for this was to loop through the label path dictionary (figure 5.5) and check where the paths were the same but in reverse and remove one of these from the dictionary that stored all of the possible paths. Since each side of a particular path has a different number of monsters visible, it was possible to take the intersection of the possible paths from each side of the path and eliminate any others, as for a path to be valid it has to be valid from both sides. This greatly reduced the number of paths that needed to be checked and reduced the branching factor tremendously.

With these tightened constraints and optimisations the speed of solving the puzzles was improved further and it was possible to get a valid solution to work for all 4x4 and 5x5 puzzles.

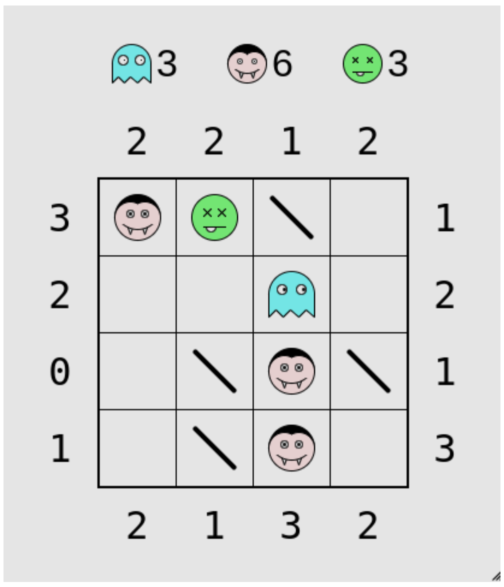
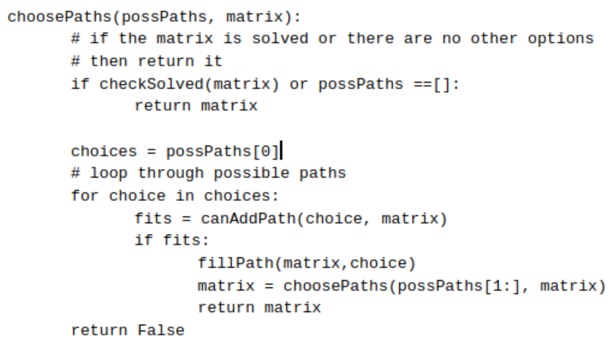


Figure 5.11: Undead Game with one path filled in



```
choosePaths(possPaths, matrix):
        # if the matrix is solved or there are no other options
        # then return it
        if checkSolved(matrix) or possPaths ==[]:
                return matrix

        choices = possPaths[0]
        # loop through possible paths
        for choice in choices:
                fits = canAddPath(choice, matrix)
                if fits:
                        fillPath(matrix,choice)
                        matrix = choosePaths(possPaths[1:], matrix)
                        return matrix
        return False
```

Figure 5.12: Pseudocode for the DFS

Chapter 6

Testing the Solution

6.1 Comparison of all solving methods

This solution to the Undead game is able to solve four different types of board:

1. 4x4 Easy - A board that is 4 squares by 4 squares and of easy difficulty
2. 4x4 - A board that is 4 squares by 4 squares and of regular difficulty
3. 5x5 Easy - A board that is 5 squares by 5 squares and of easy difficulty
4. 5x5 - A board that is 5 squares by 5 squares and of regular difficulty

The method for determining the difficulty of a board is is outlined in section 2.5.4.

For each of these different types of boards, 10 random boards were created to test each different advanced algorithm on (Section 5.3) and the solver was run 10 times for each of these boards. The results were then plotted on a graph. The individual line graphs for each method can be found in Appendix B.

6.1.1 4x4 Easy

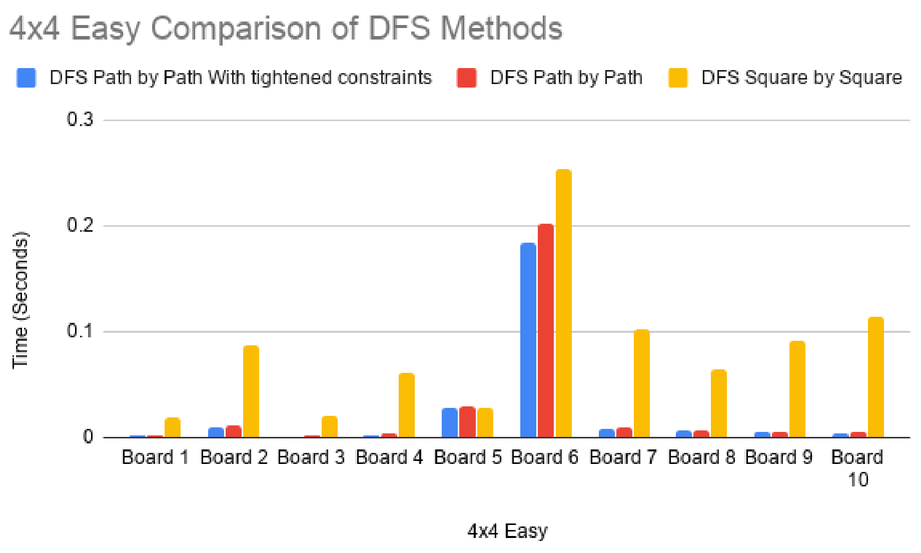


Figure 6.1: 4x4 Easy Comparison of DFS Methods

These results are as expected. Looking at figure 6.1, it can be observed that DFS square by square takes much longer than completing DFS path by path or path by path with tightened constraints. This excludes board 5 (which was an anomalous result) where performing it square by square was slightly faster on average than the other two methods. As for the other two methods, DFS path by path both with and without tightened constraints, for the easy 4x4 puzzles there is very small difference between the times and they seem equally as effective.

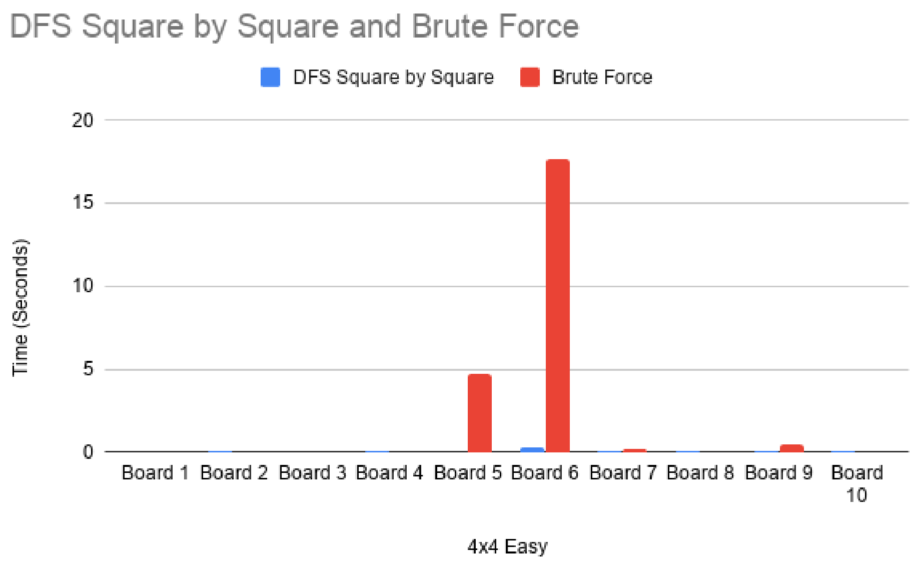


Figure 6.2: 4x4 Easy Comparison of DFS square by square vs Random Brute Force

The bar chart shown in figure 6.2 displays the comparison of DFS square by square with random brute force. This is plotted as a separate graph as the brute force took much longer than all of the other methods, so if it was plotted on the same graph it would be completely impossible to tell how the other methods performed. Therefore, it is plotted on a separate graph with the slowest of the search algorithms, however since boards 5 and 6 are so much slower than any other solves, it makes it difficult to interpret the results.

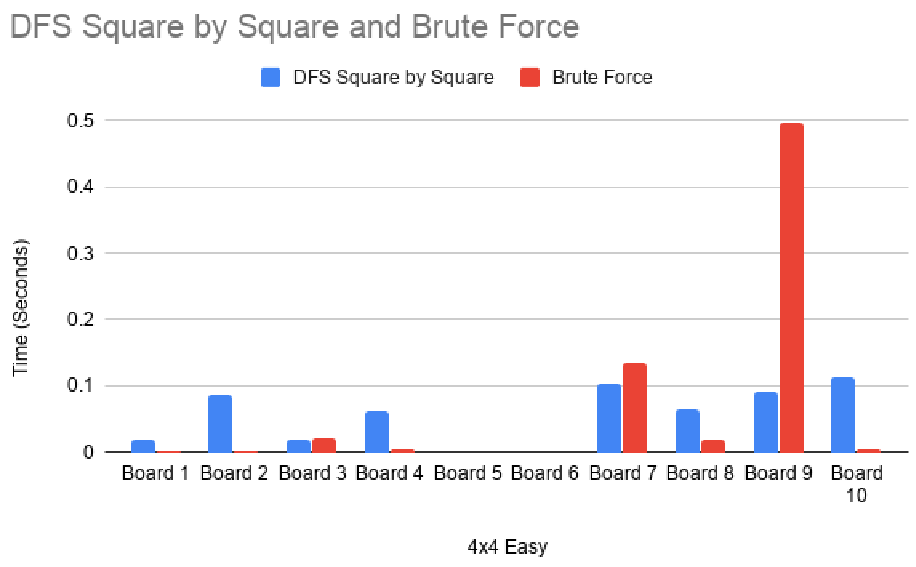


Figure 6.3: 4x4 Easy Comparison of DFS Square by Square vs Random Brute Force without boards 5 and 6

Figure 6.3 is the same graph as figure 6.2 however, boards 5 and 6 are removed. From this graph, it can be noted that in a lot of cases the brute force method works faster that the search method which is surprising. This could be because it is completely random so it could find a working method much faster than by searching through all the possibilities. Another observation that can be made is that the search method has much more consistent timings, whereas the brute force method has some faster and some slower times, which makes it a less reliable algorithm for solving a large number of problems. This means that overall, the average of all of the times across all the boards (now including boards 5 and 6) is faster for the DFS square by square (0.08414053866 seconds) than the brute force (2.308259814 seconds), indicating that overall, DFS square by square is the superior method.

6.1.2 4x4

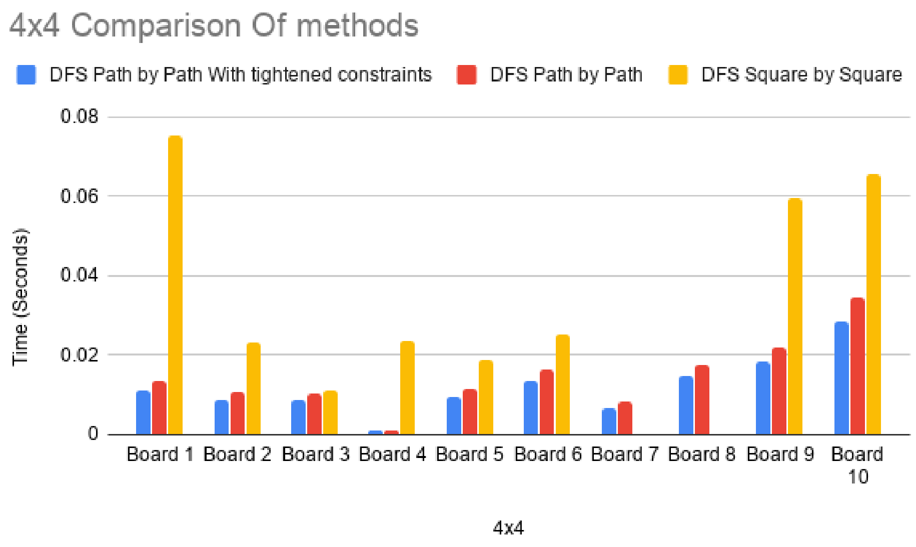


Figure 6.4: 4x4 Comparison of DFS Methods

Since the 4x4 boards are harder than the 4x4 easy boards, they take longer to solve. This also means that there are more apparent differences between the different solving methods which makes the effectiveness of them easier to analyse. For this style of board, it is clear that the DFS square by square is much slower than the other two methods, so much so that for boards 7 and 8 it failed to solve them. This shows that this method is not an effective method for solving the problem as it failed at the second easiest type of board. The other two methods were close in timings for solving all of the boards but with the exception of board 4, the DFS path by path with tightened constraints was slightly faster which implies that this could be a better method overall.

For the same reasons as mentioned in the previous section, DFS square by square and brute force were compared separately (figure 6.5). Clearly, these findings are much the same as in the previous section. It varies across the boards whether the DFS square by square or the brute force is faster, however when the DFS is faster it is normally by a significant amount. For boards 7 and 8 the random brute force was able to solve them whereas the DFS square by square was not. This would indicate that even though on average the DFS square by square was faster, overall, the brute force is a better method for this set of boards.

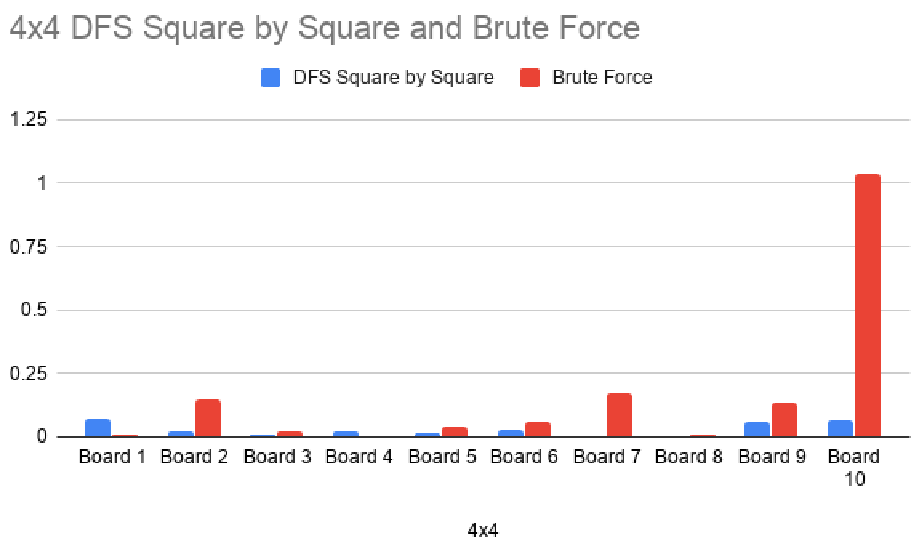


Figure 6.5: 4x4 Comparison of DFS Square by Square vs Random Brute Force

6.1.3 5x5 Easy

For the 5x5 Easy boards, the brute force failed on all of the boards, hence it is not included in any of the charts. This shows that brute force is the least effective method investigated as it is unable to solve all types of board and consequently cannot offer a complete solution. This is due to the fact that, as calculated in section 3.1, a 4x4 board has 3^{12} possibilities whereas a 5x5 board has 3^{18} possibilities. This is an increase of a factor of 3^{6} times, which means that there are significantly more possibilities to try and since the brute force algorithm is on average O(n^3) time complexity this type of board is much too computationally expensive for this solving method.

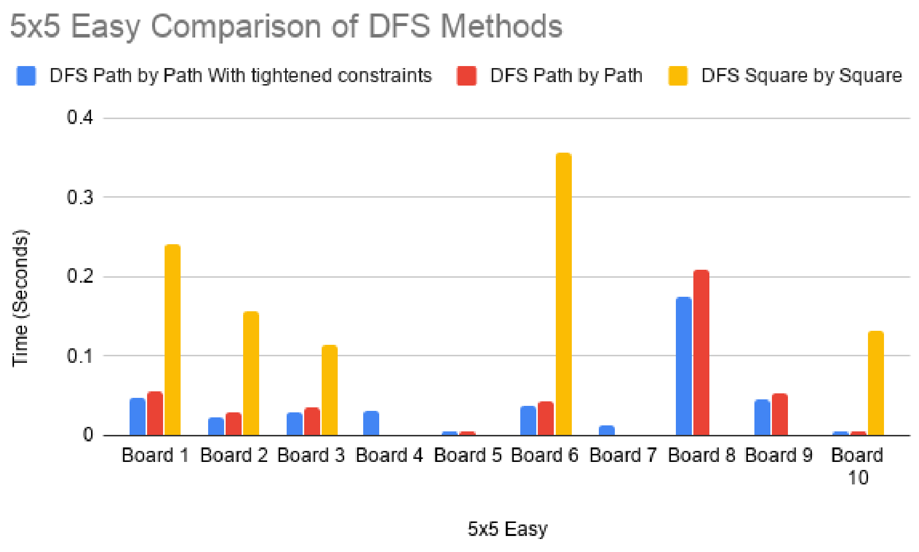


Figure 6.6: 5x5 Easy Comparison of DFS Methods

In figure 6.6, the comparisons of the different DFS methods on these boards can be examined. Across all 10 boards, the same pattern persists: the DFS square by square is significantly slower than the other two methods and even fails on boards 4, 5, 7, 8 and 9. The next slowest method is DFS path by path, however this is only slightly slower than DFS path by path with tightened constraints apart from boards 4 and 7 in which this method failed. The reason that this method failed was that there were too many paths to check which increased the branching factor significantly; this meant that the algorithm was too computationally expensive and failed as it took too long to solve. The DFS path by path with tightened constraints worked for every board and worked faster than the other two methods as it significantly reduced the number of possible paths requiring checking (as described in section 2.5.5) which meant that it was able to try all possibilities more quickly and therefore find the solution.

6.1.4 5x5

The brute force method was not compared for this type of board as it was unable to solve any, for the same reasons described in section 6.1.3.

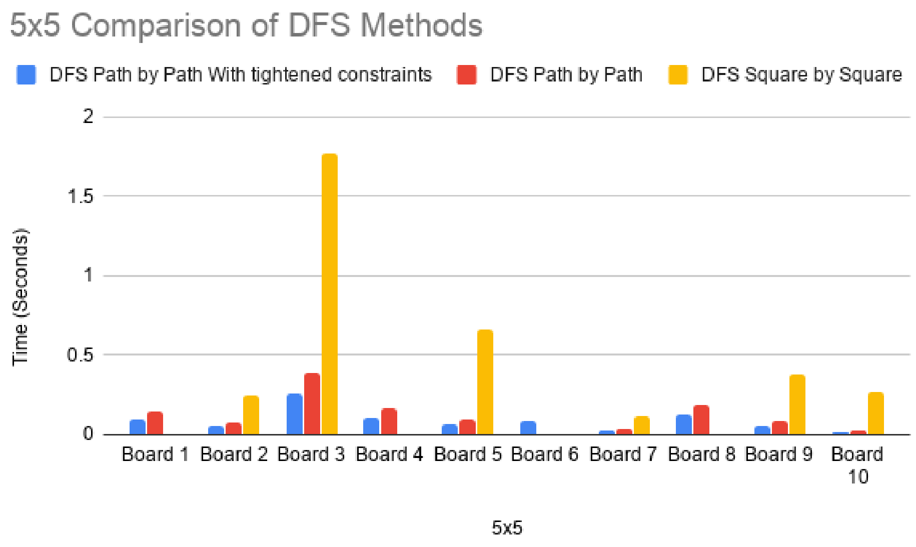


Figure 6.7: 5x5 Comparison of DFS Methods

The solving methods performed as expected on this style of board. Overall, they took more time to solve the puzzles than on the 5x5 Easy boards, however, they still show the same trends in the differences in times of each method. The DFS square by square was still the slowest out of the three methods and it also failed on boards 1, 4, 6 and 8. The DFS path by path was the next slowest, however the differences in time taken to solve between this method and the DFS path by path with tightened constraints were more apparent and it additionally failed on board 6. On the whole, the DFS path by path with tightened constraints method was the fastest and most consistent for this style of board as it was able to solve every board, moreover in a faster time than the other methods.

6.1.5 Summary

For every style of board, the same trend persisted: the slowest method tended to be the brute force, followed by the DFS square by square and DFS path by path. The fastest and most consistent method was the DFS path by path with tightened constraints.

Overall, the three-step approach implemented was highly effective in solving the Undead game:

1. Generating all possible paths
2. Tightening the constraints on these paths
3. Doing a depth first search using the paths

This method was much more effective than the traditional DFS square by square approach, which is applied to many different puzzles and games, as the method created was individually tailored to this game and capitalised on how the constraints are enforced and how the board must be filled in. This method was significantly faster than all other methods and was able to solve every board presented to it with no difficulty, which means that it is a viable and effective solution to the problem presented in this paper.

6.2 Comparison of filling in zero paths

As described in section 3.2.3, before any solving algorithm is run, the paths with zero monsters visible are filled in. This section shows the difference in performance if these are filled in vs if they are not. For simplicity, the method of filling in zero paths will be referred to as "*zero-fill*" from now on. In the previous section, it was established that DFS path by path with tightened constraints was the best method, and thus this method will be used to evaluate the impact that *zero-fill* has on the times to solve.

6.2.1 DFS Path by Path with Tightened Constraints with and without zero-fill

4x4 Easy

In general, for the 4x4 easy boards, implementing *zero-fill* produces a slightly faster solve. This is with the exception of board 6 in which using it is slightly slower, however this is an anomalous result and can be put down to random variation. Predominantly, using *zero-fill* is faster by a small margin which shows that this method is slightly better, however most solves are still below 0.08 seconds for both with and without it. This is most likely due to the fact that this type of board is the easiest of all the boards so it is easy enough to solve that implementing *zero-fill* only provides a small optimisation.

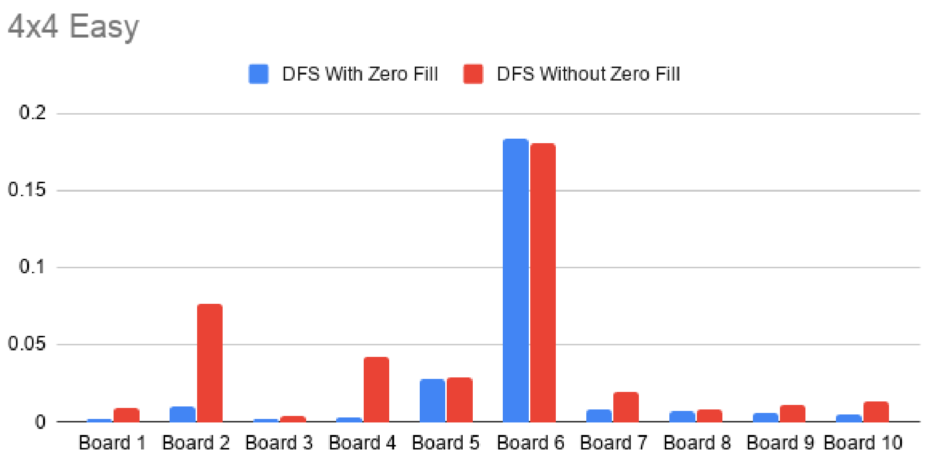


Figure 6.8: 4x4 Easy Zero Fill Comparison

4x4

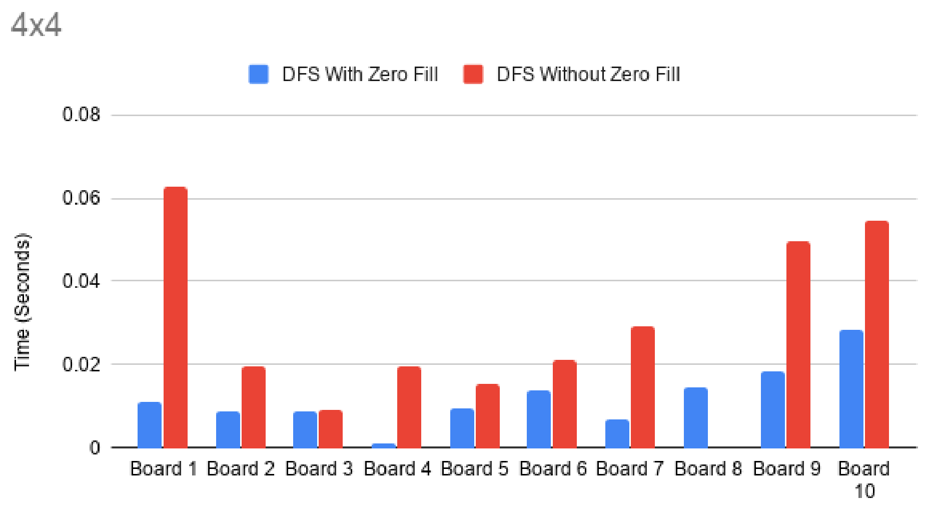


Figure 6.9: 4x4 Zero Fill Comparison

Considering a comparison with a regular 4x4 board, it is clear that implementing the *zero-fill* makes a much larger difference. This is especially highlighted in board 8, where the algorithm failed to solve the problem without using *zero-fill*. Since this board is harder than the previous to solve, using *zero-fill* reduces the time to solve significantly and therefore is a necessary and effective addition.

5x5 Easy

For the 5x5 Easy boards the *zero-fill* is even more effective. This is clearly shown as boards 4, 5, 7, 8 and 9 are all unable to be solved without using the *zero-fill* and all other boards take significantly longer without it. This is a vast difference from the 4x4 boards as there are many more possibilities for the 5x5 board, so by partially filling in the board beforehand, this reduces the number of paths to check significantly, leading to the solution being found more quickly.

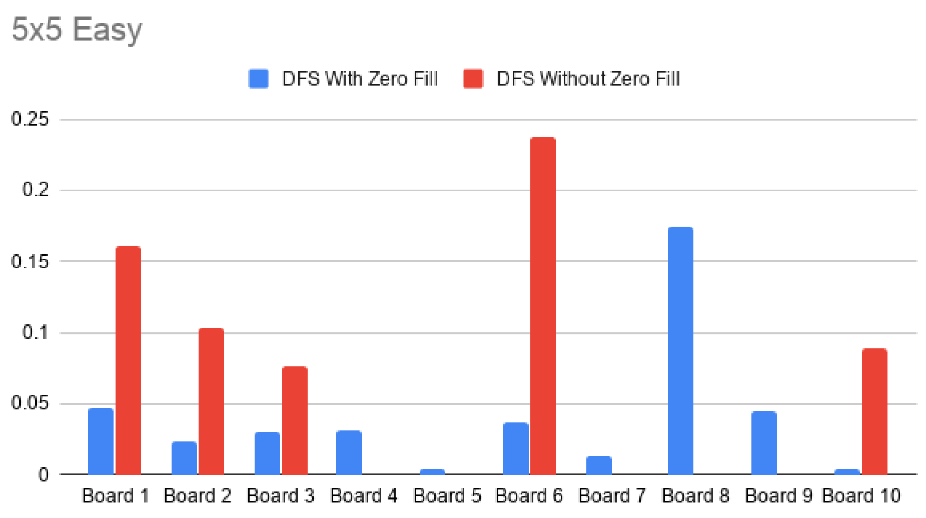


Figure 6.10: 5x5 Easy Zero Fill Comparison

5x5

Finally, for the 5x5 board, the necessity for use of *zero-fill* is just as apparent as in the previous section. Without using this, the algorithm fails for boards 1, 4 and 6 and takes significantly longer with all of the other boards. This due to the same reasons as discussed for the 5x5 Easy boards and shows that the *zero-fill* unequivocally optimises the solve.

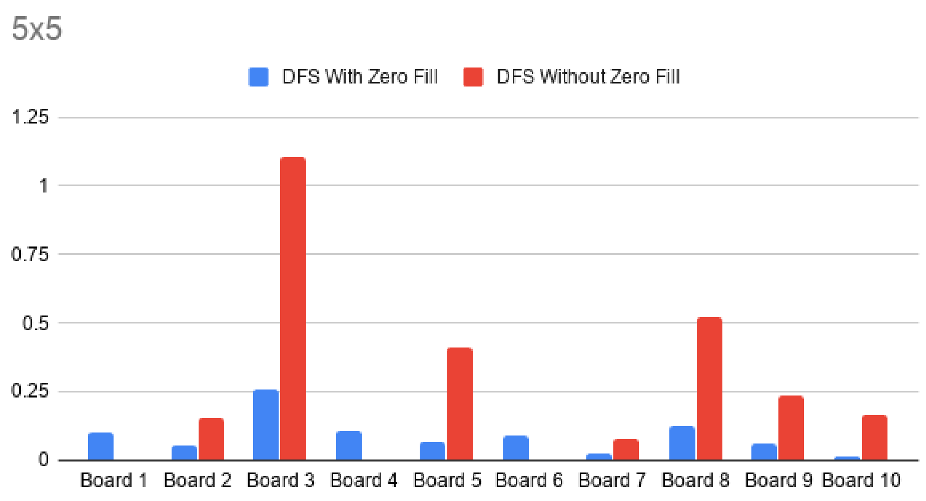


Figure 6.11: 5x5 Zero Fill Comparison

Summary

In conclusion, by pre-processing the board and filling in the zero paths with their implicit values, the number of possible paths are greatly reduced which considerably improves the performance of the solving algorithm for every type of board. For this reason, it is a necessary optimization for a good solution to the overall problem.

Chapter 7

Conclusion

7.1 Aims and objectives

This section will critically review the objectives that were set out in section 1.3.

1. **Pursue and document my background research into the implementation of AI in games and puzzles**

   This objective was achieved in Chapter 2 where research was conducted into a number of different games and puzzles and how AI is used in order to solve them. A number of the techniques discovered in this section were used later in the project in order to create my own unique solution.

2. **Read in a string of characters from a text file and process these into a particular configuration of the Undead game**

3. **Form a path by iterating through adjacent squares and changing direction if a mirror is hit**

4. **Represent the board in a form that it can be solved**

   Objectives 2, 3 and 4 were all achieved and demonstrated in sections 3.2.1 and 5.1. Section 3.2.1 describes how the board should be read from a text file, and what data structures need to be formed to work with the rest of the program. In section 5.1, how this is undertaken is precisely described and it shows how the board is represented within the text file for it to be read. The trace path function is shown, and its functionality is described in terms of how it traverses through the path, and how the relevant data structures were created.

5. **Develop an AI algorithm to solve both 4x4 and 5x5 configurations of the Undead game**

   Sections 5.2 and 5.3 show the 4 different AI strategies, both basic and advanced, that were created and implemented in order to solve the different puzzles. They present how these strategies differ from one another and how they were implemented into the final solution.

6. **Evaluate the performance of this AI on a number of different 4x4 and 5x5 boards**

   The performance of the different algorithms were compared in chapter 6. This chapter provides graphs and descriptions evaluating the performance of the different algorithms as well as analysing their strengths and weaknesses.

Overall, I consider the aims of the project, outlined in section 1.2, were achieved to a high standard. The AI has shown that it can solve the puzzle efficiently and accurately. Moreover, it was able to be scaled up to a 5x5 board. Although there is not an existing solver to compare my solution against, I can confidently say that the solution is written to the best of my ability bearing in mind the time constraints: it achieves the objectives in a sufficiently short time and has found a solution to all the puzzles presented to it.

7.2 Future work

On the website for Undead [18] games, a 7x7 board can also be generated. Given more time, I would extend my solution to work for these boards as, currently, the solution takes too long to solve this type of board and creates too many recursive calls. Solving this issue would be my initial step if I were to re-visit this problem.

I would also create a GUI that would display the solved board to the user so that they would have a clearer view of the game. Furthermore, another feature that could be added would be to animate the solving algorithms so that the user would be able to see how they actually worked and observe the DFS trying different possibilities and then backtracking to try other ones. This would be beneficial for demonstration purposes to show how the algorithms work and could help to visualise why certain algorithms are superior to others.

7.3 Legal, Social, Professional and Ethical issues

The British Computer Society's (BCS) code of conduct [4] and the Association for Computing Machinery's (ACM) code of ethics were consulted in order to determine the legal, social, professional and ethical issues that would be encountered during this project.

7.3.1 Legal

One of the four key principles of BCS's code of conduct [4] "have due regard for the legitimate rights of third parties". This principle relates to the potential issue of the violation of data protection and intellectual property laws. However, this project has cited all resources that have been used to create the solution, in particular the original Undead game created by Simon Tantham [18]. This project also does not use any third-party data or service and, therefore, there are no legal issues arising in this respect.

7.3.2 Social

As this project is an automated solver, it has very little human interaction. I am the only person that has worked on the project and the only person who has tested the program, therefore, no social issues have arisen or could arise.

7.3.3 Professional

Section 2 of the BCS code of conduct expresses the standards of professional competence and integrity [4]. It describes the professional issues that could arise in a project, these can be summarised as the developer misrepresenting their skills and resources when taking on a task, thus overreaching their scope of practice. ACM's code of ethics also reiterates this [8]. This project was fully completed and met all the objectives set, which clearly indicated that it was within my level of competence and capabilities. Furthermore, any issues that had arisen in the iterations leading to the final solution were discussed in Chapter 6, during the testing of the different solutions. Therefore, there are no professional issues that will arise for this project.

7.3.4 Ethical

Both the ACM's code of ethics [8] and the BCS's code of conduct [4] discuss the ethical principles that any developer should adhere too. The principles can be summarised as follows:

- Ensure that the public good is the central concern
- Respect the privacy of others
- Do not discriminate against others.

This project has not used any external data which eliminates the privacy issue. This project has not discriminated against anyone and has limited user interaction therefore there are no ethical issues.

Due to the fact that AI techniques are central to this paper, considering the "public good" as an ethical concern can be expanded to AI as a whole [14]. AI is becoming increasingly prevalent in today's society and is now being used in most industries. The primary reason for the increasing use of AI, as shown in this paper, is that AI can solve problems much faster and more efficiently than humans. This has led to AI algorithms and techniques replacing humans in certain jobs. This can be considered an ethical issue as the loss of jobs could be viewed as detrimental to the "public good". Furthermore, AI is being used increasingly in medical applications which raises the debate: if a patient is harmed as a result of an AI's decision, who takes responsibility? Is it the doctor who decided to use the AI? Is it the developer who created the AI? These are just some of the ethical issues that could arise. As AI is still a relatively new field, and we are still progressively implementing it into society many ethical issues still have not been fully explored, and therefore the ethics of AI are still a heavily debated subject among professionals in computing.

Overall, there are no particular ethical issues that have arisen in this project, however the general ethical issues of the increasing development and use of AI are applicable.

7.4 Personal Reflection

Completing this project has been extremely challenging but equally, rewarding. It has been enjoyable to work on a project and subject area that I am passionate about. The skills I have developed and experience that I have gained will be invaluable in future projects.

Through this project I have learnt about implementing and adapting various search algorithms and how AI can be used to solve different problems. I have also developed project management skills and learnt about the importance of iterating through different designs in order to find a solution that works to meet the intended specification. Despite this, there are still areas for improvement, my management of the project could be improved as I incorrectly predicted how long certain elements of the coding would take which lead to me finishing the final solution much later than intended. However, as this is my first individual project, I believe this need to adapt my project plan has had a positive impact as I have learnt to ensure that I account for delays in future project planning. Ultimately, I am passionate about AI and find the subject area of my project stimulating, which was key in maintaining my focus on the project and ensuring that I finished each stage relatively on schedule.

The biggest challenge in the project presented itself when I had to implement the DFS path by path with tightened constraints. This was a unique solution that I crafted specifically for this project and, although the base structure of it was a DFS, there were many more different elements that made this solution distinct from a classic DFS. As a result there was not any existing code or algorithms that I could take inspiration from in order to create this unique algorithm. Furthermore, as this was the last solution that was created, I had to write this code concurrently with a large number of my other university assignments, which only made the problem more demanding. Although I planned for this with my risk mitigation strategies in section 4.4, I did not accurately assess how difficult this problem would be to solve. In the future, I will be more careful and analytical when creating my timeline, anticipating all reasonable possibilities.

In conclusion I consider that I have produced a good,effective solution to create an artificial intelligence algorithm that is competent enough to solve the game of Undead, in a reasonable time. I have been challenged and have enjoyed gaining experience about the subject area and implementing the different stages of the project.

References

[1] A. Alliance. Agile 101. https://www.agilealliance.org/agile101/. Accessed: 2019 - 12 - 16.

[2] W. Ball. Mathematical recreations and essays. fourth edition. pages 88–103, 10 2008.

[3] Y. Bassil. A simulation model for the waterfall software development life cycle. *CoRR*, abs/1205.6904, 2012.

[4] BCS. Bcs code of conduct. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/. Accessed: 2020 - 4 - 21.

[5] K. Binmore. Playing for real. a text on game theory. *Journal of Economics*, 93:216–217, 04 2008.

[6] G. Bonanno. Game theory (open access textbook with 165 solved exercises). abs/1512.06808, 2015.

[7] B. Coppin. Artificial intelligence illuminated. pages 75–80, 2004.

[8] A. for Computing Machinery. Acm code of ethics. https://www.acm.org/code-of-ethics. Accessed: 2020 - 4 - 21.

[9] E. Hoffman, J. Loessi, and R. Moore. Constructions for the solution of the m queens problem. *Mathematics Magazine*, 42:66–72, 03 1969.

[10] J. Levine. Constraint satisfaction: Introduction. https://www.youtube.com/watch?v=_e64FiDWvqs. Accessed: 2020 - 4 - 15.

[11] A. Newell and H. Simon. Computer science as empirical inquiry: Symbols and search. pages 113–126, 1976.

[12] P. Norvig. Sudoku solver. https://norvig.com/sudoku.html. Accessed: 2019 - 12 - 16.

[13] J. Pearl. Heuristics: Intelligent search strategies for computer problem solving. page 165, 1988.

[14] W. Ramsey and K. Frankish. *The Cambridge Handbook of Artificial Intelligence.* Cambridge University Press, 2014.

[15] D. Ross. Game theory. https://plato.stanford.edu/archives/win2019/entries/game-theory/. Accessed: 2020-04-01.

[16] sgt puzzles. Undead manual ubuntu. http://manpages.ubuntu.com/manpages/bionic/man6/sgt-undead.6.html. Accessed: 2019 - 12 - 16.

[17] P. N. Stuart Russell. *Artificial Intelligence A Modern Apporach Third Edition.* Prentice Hall, third edition, 2009.

[18] S. Tatham. Undead. https://www.chiark.greenend.org.uk/ sg-tatham/puzzles/js/undead.html. Accessed: 2020 - 4 - 4.

[19] Vivek. In the undead game, what makes one board more difficult than another? https://puzzling.stackexchange.com/questions/17995/in-the-undead-game-what-makes-one-board-more-difficult-than-another. Accessed: 2020-04-01.

[20] T. YATO and T. SETA. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A, 05 2003.

Appendices

Appendix A

External Material

- Gitlab was used to version control the source code of my project. This is available at https://gitlab.com/sc17kdp/final-year-project
- VSCode was used to develop the game and AI
- Overleaf was used to write the report
- Google Sheets was used to create all the graphs and charts seen in the report

Appendix B

Additional Graphs

B.1 4x4 Easy

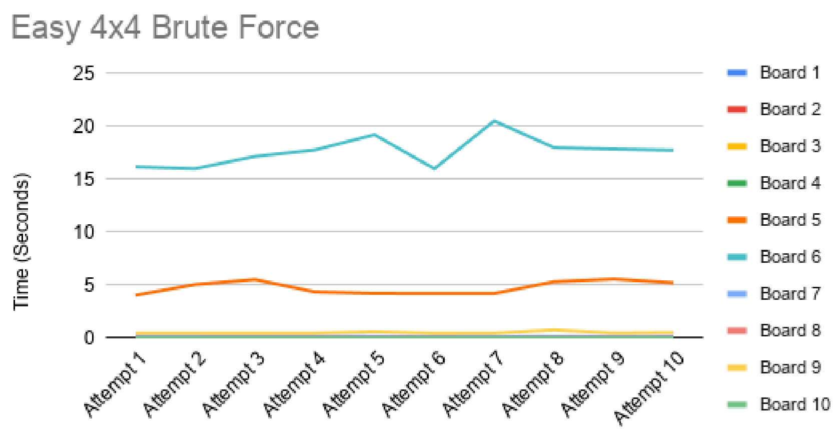


Figure B.1: 4x4 Easy Brute Force Comparison

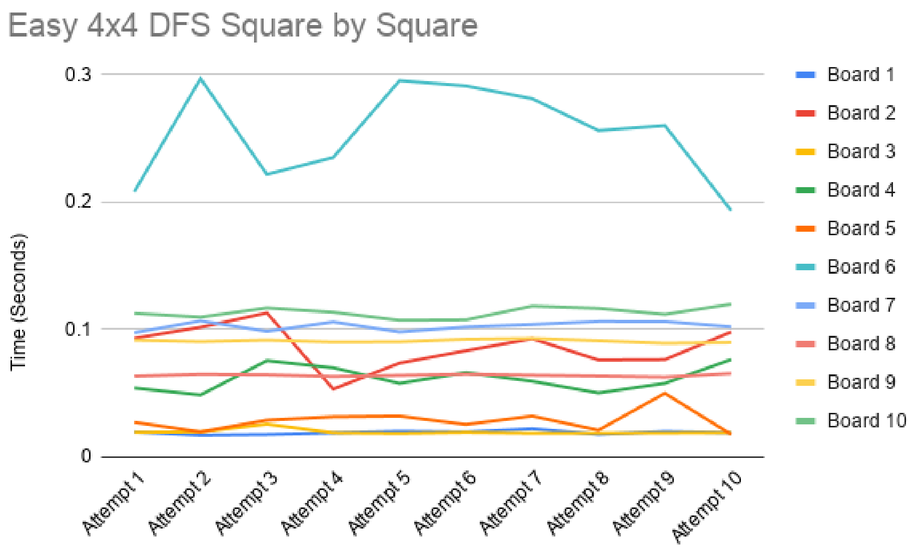


Figure B.2: 4x4 Easy DFS Square by Square Comparison

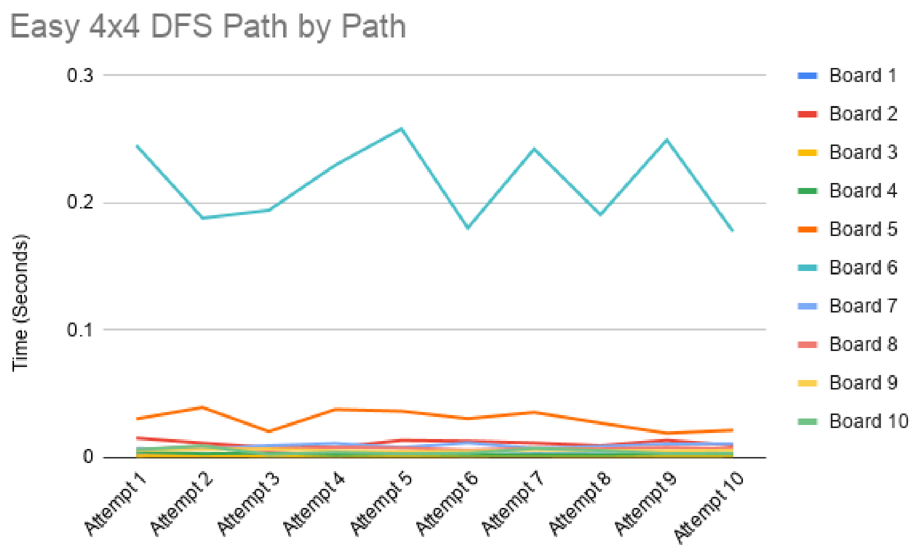


Figure B.3: 4x4 Easy DFS Path by Path Comparison

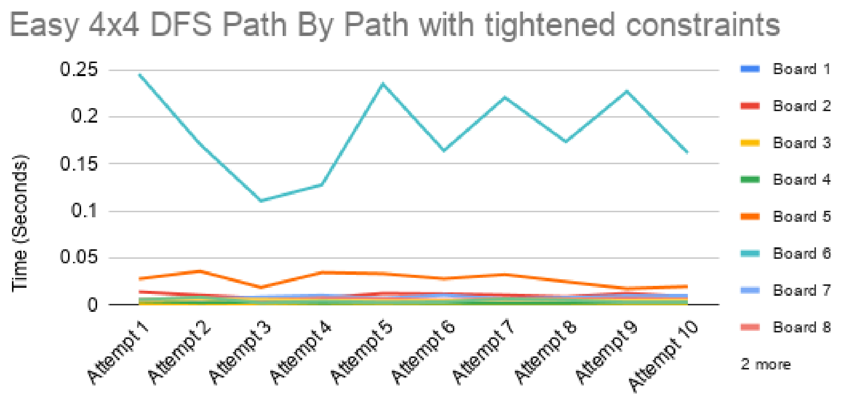


Figure B.4: 4x4 Easy Path by Path with tightened constraints Comparison

B.2 4x4

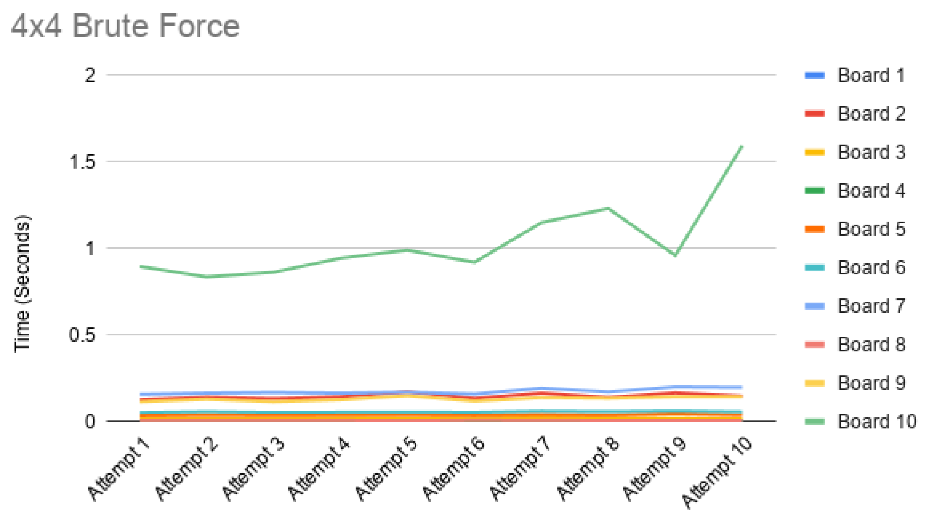


Figure B.5: 4x4 Brute Force Comparison

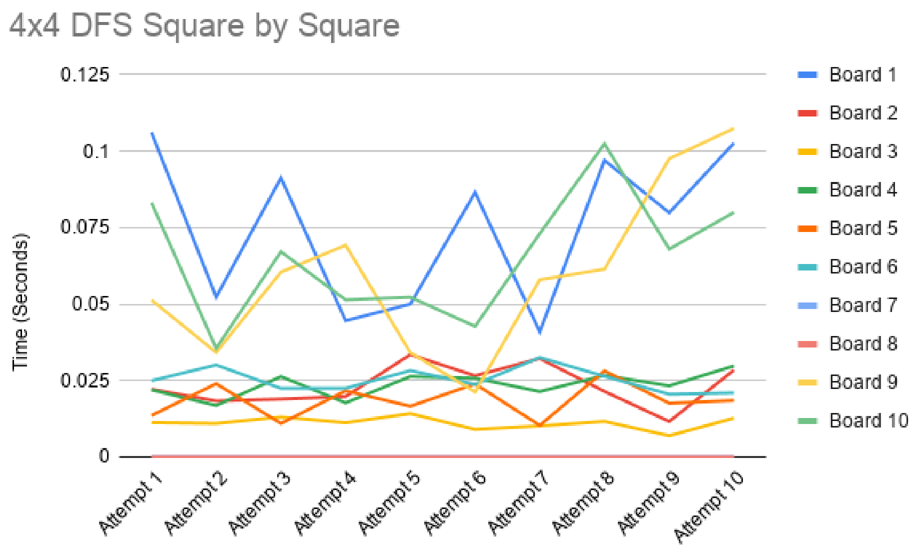


Figure B.6: 4x4 DFS Square by Square Comparison

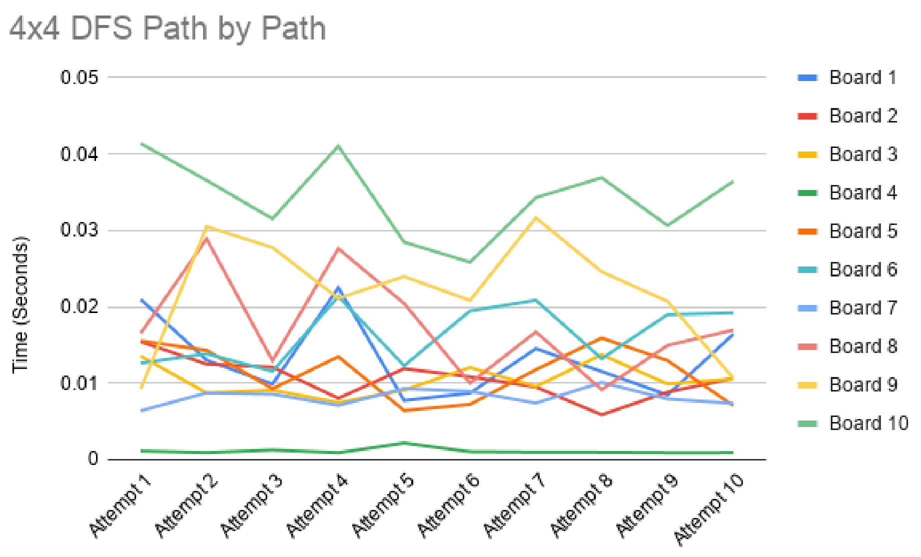


Figure B.7: 4x4 DFS Path by Path Comparison

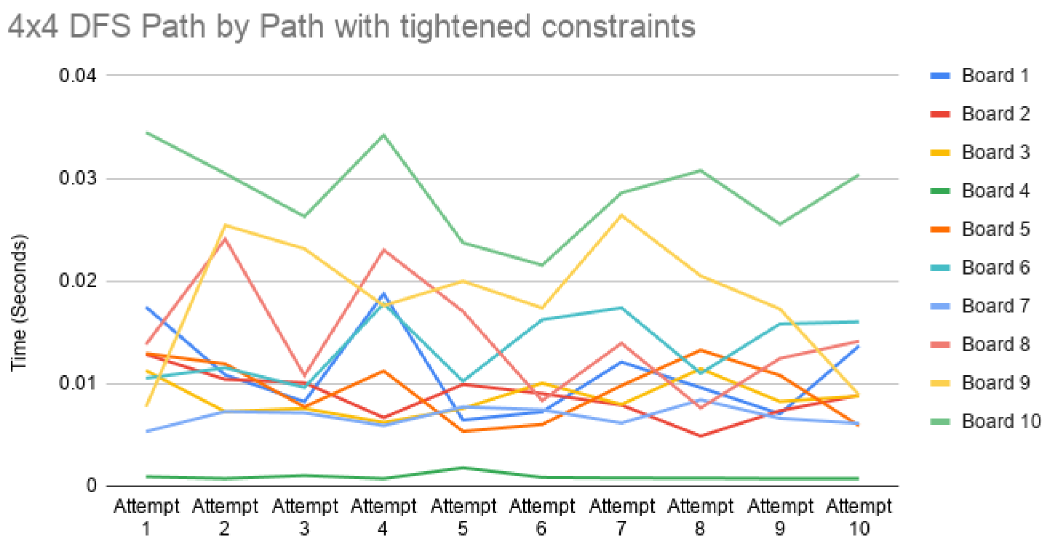


Figure B.8: 4x4 Path by Path with tightened constraints Comparison

B.3 5x5 Easy

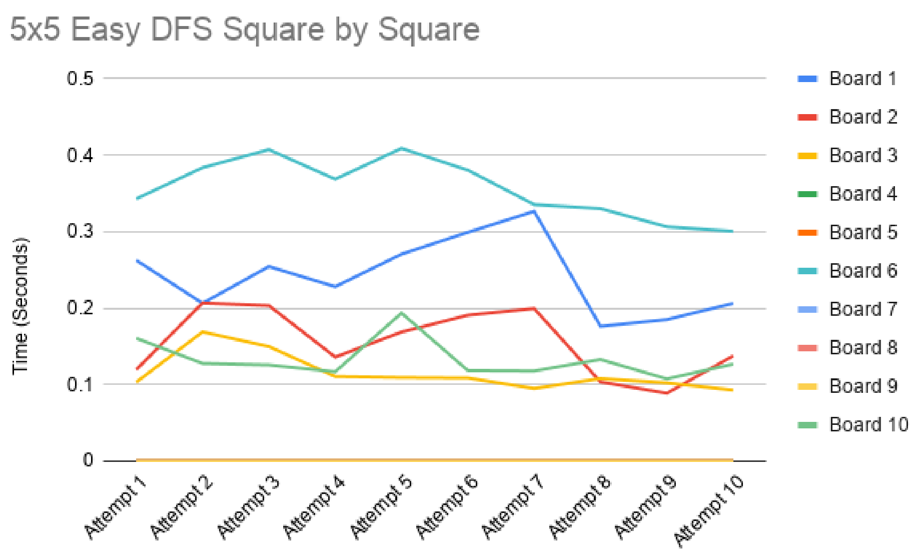


Figure B.9: 5x5 Easy DFS Square by Square Comparison

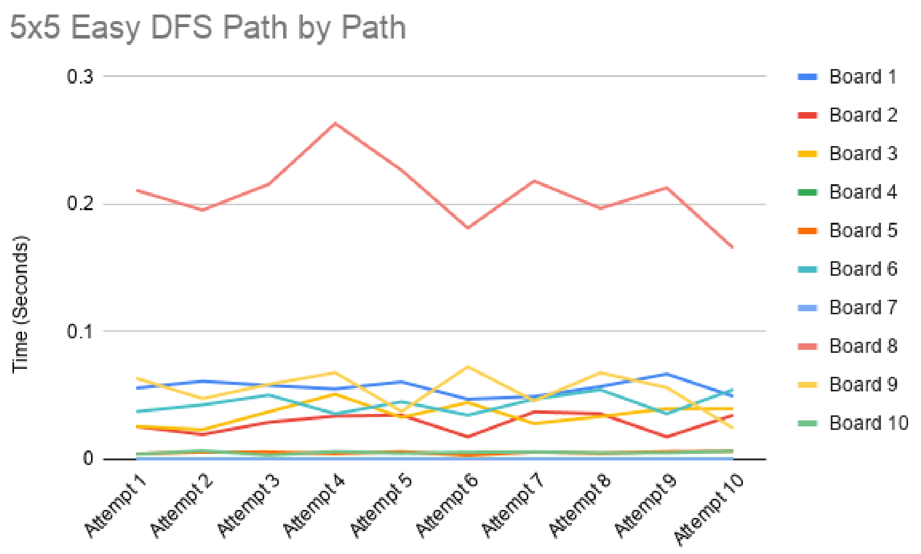


Figure B.10: 5x5 Easy DFS Path by Path Comparison

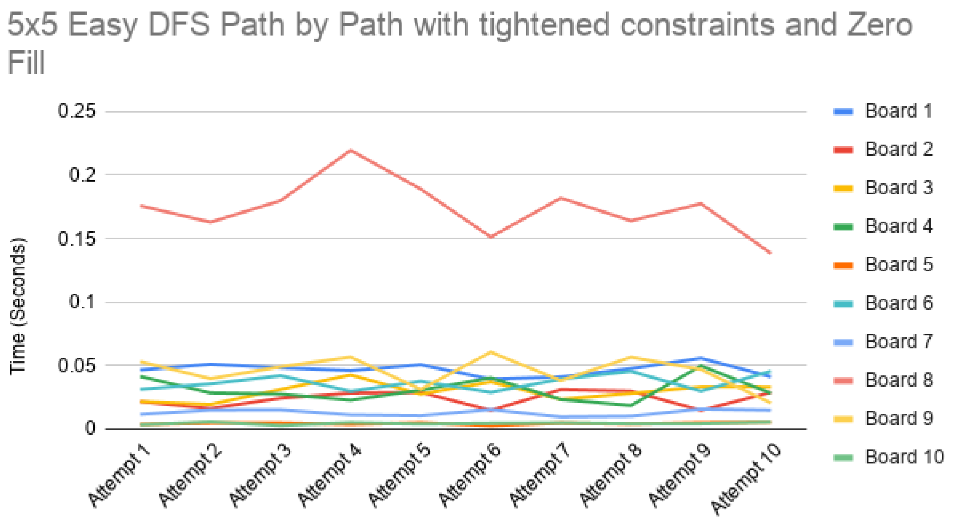


Figure B.11: 5x5 Easy Path by Path with tightened constraints Comparison

B.4 5x5

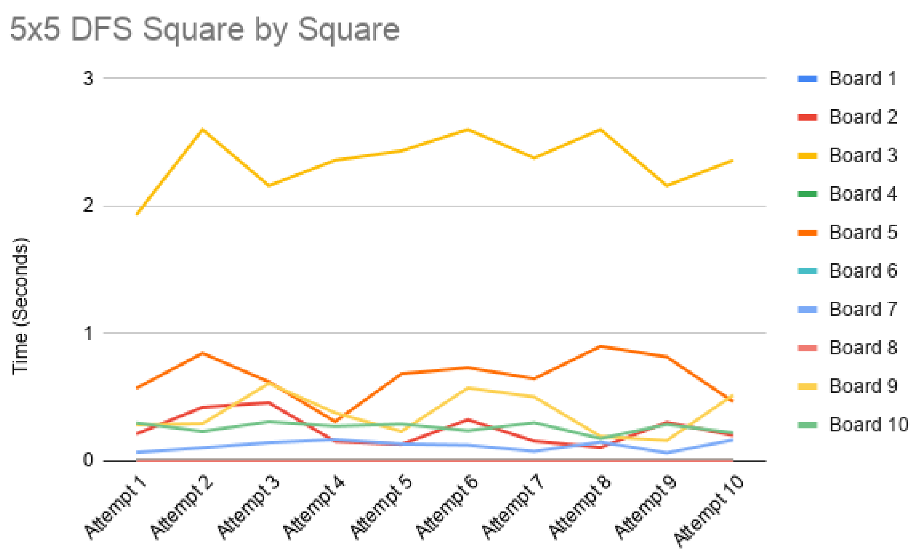


Figure B.12: 5x5

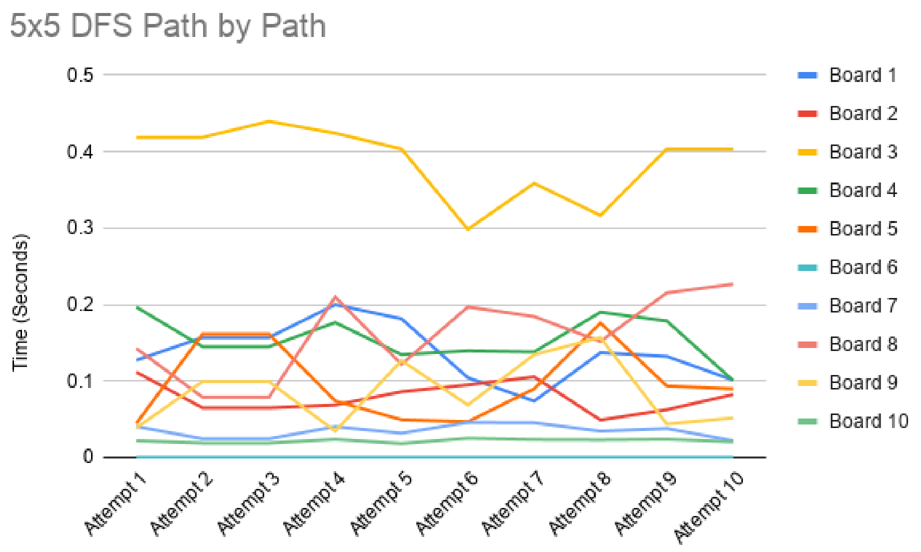


Figure B.13: 5x5 DFS Path by Path Comparison

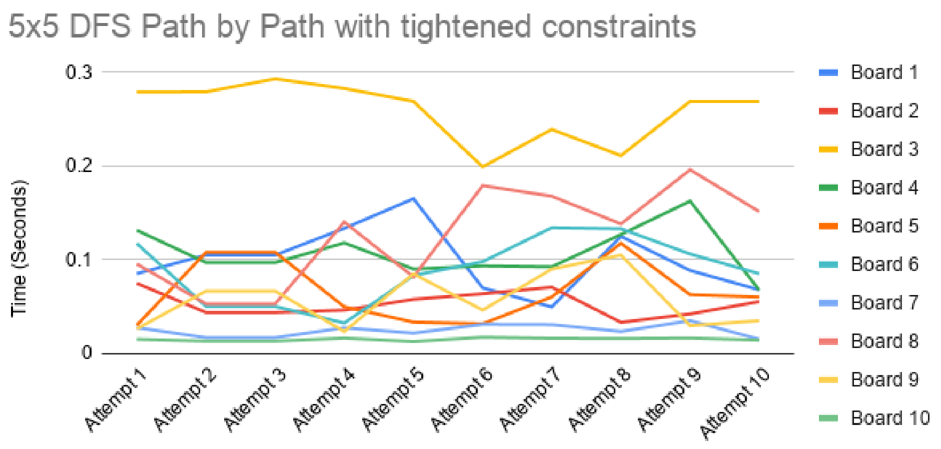


Figure B.14: 5x5 Path by Path with tightened constraints Comparison