

ID: 4308281

Supervisor: Venanzio Capretta

Module Code: COMP3003

18/05/2020

Game Development - *Rithmomachia with Artificial Intelligence*

Submitted May 2020, in partial fulfilment of the conditions for the award of
the degree **BSc Hons Computer Science**

Athullya Roy


With supervision from Venanzio Capretta

4308281

psyar7@nottingham.ac.uk

School of Computer Science
University of Nottingham

*I hereby declare that this dissertation is all my own work, except as indicated
in the text:*

Signature: 

Date: 18 / 05 / 2020

*I hereby declare that I have all necessary rights and consents to publicly distribute this
dissertation via the University of Nottingham's e-dissertation archive.*

Abstract

Game play has always been a significant form of entertainment and relaxation. Board games is a branch of games which often requires sufficient intelligence and analytical thinking skills from the player. Through this project, a medieval board game, rithmomachia, will be rediscovered and implemented using Python. The implementation will also follow the trend of most online board games by having an AI opponent for a player to compete against.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Venanzio Capretta, for his guidance and consistent support throughout this project. I would also like to thank Jason Atkin, my tutor, for his confidence in me and for all the kindness he has shown. I am also thankful for the supervision and guidance provided by Armaghan Moemeni.

Finally, I would like to express my appreciation towards my family and friends for their wise counsel and their words of encouragement throughout the year.

Table of Contents

1. Introduction	1
1.1 Project Overview	1
1.2 Brief History of Artificial Intelligence in Board Games	1
1.3 Aims and Objectives	2
2. Motivation	2
3. Description of the work	3
3.1 Rules and Structure	3
3.2 Capturing	4
3.3 Victories	4
3.4 Rules and Structure – Mathematical Perspective Overview	5
3.5 Functional Requirements	5
4. Related Work	5
4.1 StockFish	6
4.2 AlphaZero	6
4.3 Reflections	7
5. Methodology	7
5.1 Game Trees	7
5.2 MiniMax Algorithm	8
5.2.1 Pseudocode	8
5.2.2 Evaluation Function	8
5.2.3 Diagram	9
5.2.4 Drawbacks of MiniMax Algorithm	9
5.3 Alpha Beta Pruning	9
5.3.1 Pseudocode	9
5.4 Monte-Carlo Tree Search	10
5.4.1 Diagram	11
5.4.2 Pseudocode	11
5.4.3 Exploration-Exploitation and UCT	12
5.4.4 Advantages and Disadvantages	13
5.4.5 MCTS and AlphaGo	13
6. Design	13
6.1 Initial Considerations	14
6.2 Early Design Concepts	14
6.3. Move Generation and Position Evaluation	15
7. Implementation	17

7.1 Rithmomachy	17
7.2 MiniMax.....	18
8. Evaluation.....	19
8.1 Evaluation of Game Features.....	19
8.2 Minimax Game Tests	20
8.3 Game Experience Questionnaire – Core Module.....	21
9. Summary and Reflections	22
10. Project Management.....	22
10.1 Initial Plan	22
11. Contributions and Reflections	24
11.1 Self Reflection.....	24
11.2 Improvements on the project	24
12. Bibliography	25
13 Appendix.....	27

1. Introduction

1.1 Project Overview

From the conception of board games in the early dynastic period till today, board games have been played and enjoyed by individuals of all race and nature. From royalty to deprived masonry workers, board games have been significant to both education and recreation throughout history.

Especially, due to the increased use of the internet and a recent surge in the gaming industry, more people are becoming aware and investing their time into playing board games. This is reflected by the fact that the global board games market is predicted to rise over \$12 billion by the year 2023 ^[1]. Observing this recent growth in the industry, it was decided to base the project around the development of such a game – rithmomachia.

Rithmomachia, also known as “The Philosopher’s Game” or “The Battle of Numbers”, is a long forgotten mathematical board game which at its peak was once a strong rival to Chess. The game was once popularised by students and academia for its representation of Boethian mathematical philosophy and its need for quick mental arithmetic. However, its popularity has been fluctuating since its formation in medieval Europe, where it had a life span of over 500 years ^[2]. The game and its rules has been rediscovered by historians by analysing the handful of medieval literature which discussed the game. However, different versions of the descriptions exists as different authors documented it with contradiction. Therefore, the rules and structure stated in this report purely indicates the version which I have developed.

1.2 Brief History of Artificial Intelligence in Board Games

Board games have been closely linked to the development of artificial intelligence, as games are intelligent activities which also provide a simple measure of success or failure. In addition to this, game play does not require large amount of knowledge; as the only necessary knowledge is an understanding of the rules of the game and the conditions for winning and losing. This makes implementation easier and the overall design more feasible.

Artificial Intelligence opponents for games are very common in this era. In this past decade, artificial intelligence has improved significantly to beat world champions in games such as Chess and Go. While implementing Rithmomachia, I will also focus on developing an algorithm which uses artificial intelligence to create an opponent for the player.

The first significant step towards creating an artificial opponent in game play was described in C. E. Shannon’s momentous publication, “*Programming a Computer for Playing Chess*”, where he introduced the theory of the minimax algorithm ^[3]. Shannon begins by proposing an evaluation function which he suggests can be applied to a position P. He used the values to determine whether the game is won, lost or whether it is a draw. Shannon describes a machine which can evaluate the positions and select a move depending on whether that player is trying to maximise the value from the evaluation function or minimise it. Even though this was only a mere idea at the time, the algorithm is still pertinent in the field of artificial intelligence.

A. Samuel’s checkers program, developed in 1952, was the first successful implementation of a self-learning computer program ^[4]. Samuel used the minimax algorithm along with alpha-beta pruning to evaluate the positions of the game tree and chose the best move.

Since then, there has been many developments in the field of artificial intelligence and board games. DeepMind's AlphaGo Zero is an AI developed to play the Chinese game of Go. By the year 2017, AlphaGo became the best Go player in the world by beating the world champion in all three games played ^[5]. AlphaGo learned using reinforcement learning – a neural network learned by playing against itself till it could not be defeated in the game. As it plays, the neural network is tuned and parameters updated so that it can better predict the opponents moves, hence has a higher chance of winning each time.

1.3 Aims and Objectives

The aim of this project is to develop a game of rithmomachia and to produce an AI to play as the user's opponent.

The core objectives which determines the success of the project are:

- a) *Through extensive research, gain an understanding of the various approaches and algorithms used to develop AI for board games.*

Research into min-max algorithm, alpha-beta pruning, Monte-Carlo searching, machine learning etc. This research should provide sufficient information to decide which implementations will be the most feasible and successful.

- b) *Develop a game of rithmomachia which implements the rules and the multiple forms of victories.*

Research into the libraries that can aid the game development and then implement the game so it is authentic to the traditional board game.

- c) *Design and develop a graphics interface for the game.*

Ensure that the representation of the game is easy and simple for the user to understand.

- d) *Enable two human players to compete against each other.*

The game should provide the option for both human vs. human games and, human vs. AI games.

- e) *Develop an AI to play against a human player.*

Develop an AI which wins the majority of the games when playing against a human opponent.

2. Motivation

The key motivation for developing this project is the originality of the task. From research into existing products, it was found that there are no existing versions of the game on the internet; i.e. it has never been implemented before. This could be due to the unpopularity of the game or simply because of the game's complexity. The originality of the project motivated me to conduct further research on the history and origins of the game. It was fascinating to discover how widely popular Rithmomachia once was. However, in today's time, even board game enthusiasts remain unaware of this game. Therefore, this project is used as an opportunity to re-introduce this game to those who play board games yet remain unaware of rithmomachia.

Even though this European mathematical game has comparisons to chess, as both are “intellectuals” board games, the characteristics of rithmomachia are unique and distinguishes it from other board games. The main feature of rithmomachia which differentiates it from other games is that rithmomachia has its structural foundations in mathematics. For example, unlike chess, winning is not as easy as capturing pieces – it involves sufficient intelligence from the player. This is reflected in the games Glorious Victory, where the player has to tactically move a piece to the opponent’s side of the board and then form an arithmetic, geometric or harmonic progression. This need for intelligence in the player, is one of the reasons why it was once popular among students. This influence from the Pythagoreans Greek mathematics and its unique set of game rules has motivated me further to complete this project.

As well as this, there is an ever-growing demand for board games in the online market. With more players engaging with online board games, it was deemed that it would be appropriate to create a computerised version of rithmomachia. As technology is so prevalent in today’s times, it would be ideal to introduce this game as a computer program as more individuals will be introduced to this medieval game.

In addition to this, an incentive behind this project was to add a new perspective to an ancient game by creating an artificial intelligent component to it. From the formation of the term “artificial intelligence”, board games has been at the forefront of its domain of study. Hence, it was interesting to explore the research conducted in the field of artificial intelligence by implementing it to games.

3. Description of the work

3.1 Rules and Structure

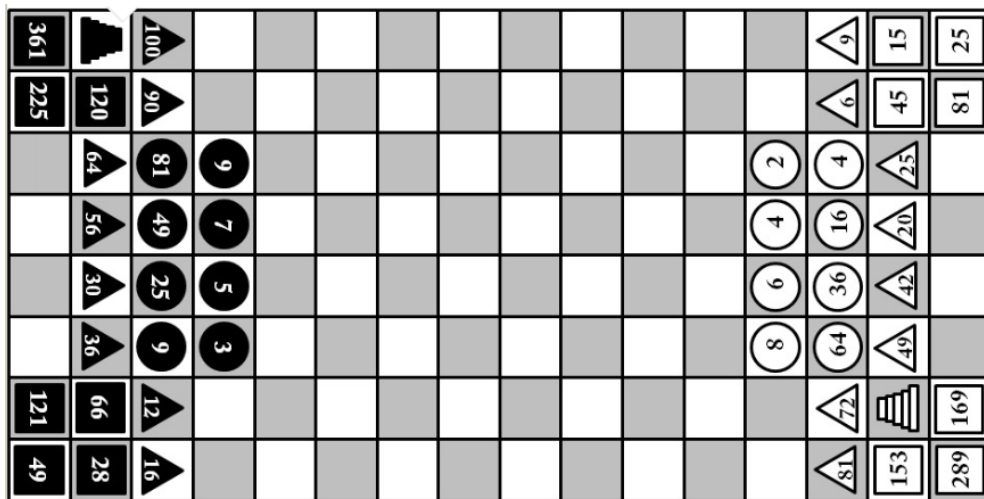


Figure 1: Rithmomachia Board - Initial State ^[6]

A sixteen by eight checkered board is required to play the game and the two players are referred to as black (odd) and white (even). Unlike traditional board games, in rithmomachia, information is asymmetrical. Players are given unique sets of pieces which consist of the shapes - squares, circles and triangles. One square on each side is replaced by a pyramid – which is formed by stacking certain pieces together. All pieces contain inscribed unique numbers. Similar to chess, the different shapes have unique directions they can move in ^[7]. The pieces are not allowed to step over other pieces but they are allowed to move in any direction.

- *Squares*: Allowed to move four steps horizontally, vertically or diagonally
- *Circles*: Allowed to move one step horizontally, vertically or diagonally
- *Triangles*: Allowed to move three spaces horizontally, vertically or diagonally
- *Pyramids*: Follow the rules of the pieces it consists of.

3.2 Capturing

In rithmomachia, the players have to capture the opponent's pieces in one of the four following ways ^[26].

- *By Meeting*: If a piece is one step away from landing on an opponent's piece, then they are allowed to capture it by meeting. However, the player's position does not change.

NOTE: Capture only happens if the current player's piece makes a move. In other words, if the opponent makes a move and the current player can capture it by meeting, then the piece will not be captured.

- *By Assault*: If a piece with a small value and a piece with a larger value are aligned unobstructed on the board, and the smaller value multiplied by the number of empty squares between them is equal to the larger piece, the larger piece is captured.

NOTE: When calculating the distance, the squares which the pieces occupy is also included (i.e. the minimum distance between two pieces is always two)

- *By Ambush*: If a piece is positioned in the middle of 2 opponents pieces, and the sum of the two pieces equal to the piece in the middle, then it is captured.

NOTE: A piece can only be captured by ambush horizontally.

- *By Siege*: If a piece is surrounded by opponent pieces on all four sides, then the piece is captured.

3.3 Victories

The game consists of two forms of victories – common and glorious. There are five types of common victories:

- *Win by Goods/De Bonis*: If a player captures enough pieces to add up to or exceed a certain value, they win the game.
- *Win by Body/De Corpore*: If a player captures certain number of pieces, they win the game.
- *Win by Lawsuit/De Lite*: If a player captures enough pieces to add up to or exceed a certain value and the total number of digits inscribed in the pieces is above a set value, they win the game.
- *Win by Honour/De Honore*: If a player captures enough pieces to add up to or exceed a certain value and they capture a certain number of pieces, they win the game.
- *Win by Honour and Lawsuit/De Honore Liteque*: If a player captures enough pieces to add up to or exceed a certain value and they capture a certain number of pieces and the total number of digits inscribed in the pieces is above a set value, the player wins the game.

On the other hand, glorious victories occur when three of a players pieces reaches the opponents side of the board and they are arranged in either an arithmetic, geometric or harmonic progression. The shapes has to form an unobstructed line (horizontal or vertical) on the board. Here is a brief explanation of the three progressions:

- Arithmetic progressions: A sequence of numbers such that the difference between all two successive terms is a constant.
Ex: 3, 5, 7
- Geometric progressions: A sequence of numbers such that elements after the first is found by multiplying the previous term with a constant
Ex: 1, 4, 16
- Harmonic progressions: A sequence of numbers such that the reciprocal of the terms form an arithmetic progression.
Ex: 3, 5, 15

3.4 Rules and Structure – Mathematical Perspective Overview

The numbers inscribed on the pieces are derived from each other through complex mathematical relations. The first row of both side consists of four pieces with either even or odd single digit numbers, depending on whether the side is even or odd.

The second row of numbers is a result of squaring the numbers of the first row. The second row of triangles are however, derived from the first by summing the value of the shape above (if first row, then the shape beside it) and adding a fraction of the number on the first row. For example, 81 (white triangle) is derived by adding 72 with $\frac{1}{8}$ th of 72. Similarly, 42 (white triangle) is derived using the formula $36 + (\frac{1}{6} * 36)$ ^[8]. The rest of the shapes are derived from triangular numbers, hexagonal numbers and so on. The purpose of having mathematical structure to the numbers is to aid in creating progressions.

The pyramids have a unique formation of numbers – white pyramids include 36, 25, 16, 9, 4 and 1 (sum = 91), whereas the black pyramids consist of 64, 49, 36, 25, 16 (sum = 190). White pyramids are known as a perfect pyramid because 91 is the sum of first 6 squares, whereas back pyramids are imperfect because $190 = 8^2 + 7^2 + 6^2 + 5^2 + 4^2$. Likewise, the numbers on all the shapes and rows follow strict mathematical rules.

3.5 Functional Requirements

Languages which could be used to develop the game were analysed to assess the benefits and disadvantages of them. From this, it was decided to implement the game in Python as it provides extensive support libraries such as Pygame and NumPy.

4. Related Work

Due to the lack of popularity of rithmomachia, the game is yet not available to be played online. However, AI development for other board game such as Chess and Go have been very successful in the past. Therefore, I have researched into existing chess engines; understanding the algorithms followed can then influence how I approach building the AI for rithmomachia.

4.1 StockFish

StockFish is one of the strongest chess engines currently available. It begins by evaluating the current board position by developing a legal-move tree, hence generating an evaluation function. This evaluates how good the current positions of the pieces on the board are. When evaluating the position, StockFish considers several human-programmed rules such as checking if the game is close to the end. Endgame weights differently to midgame since deeper searches would not need to be conducted, as winning/loosing is more close to hand. It also analyses positions to see if there are any trapped pieces that will be captured if they move or if any pieces are not protected – this should have a higher priority in the next move.

To find the best possible position, StockFish follows a search algorithm which is summarised here. Firstly, it analyses the current position and a set of legal moves which it could take - this produces the possible positions. It then analyses the possible positions, with the past positions and weakness in opponent's position to identify the strength of the position. This results in the best possible solution being generated.

StockFish generates a transposition table which is a hash table that stores the previously performed searches. This helps reduce the search space of the tree, hence making it more efficient. It uses a modified version of minimax for searching, and then uses a variation of alpha beta pruning to make the searches more efficient. While creating rithmomachia, this will influence the project as I will attempt to implement a transposition table so that the same path isn't searched multiple times ^[9].

In addition to this, StockFish also uses different heuristics depending on whether the game is in the beginning or if it is midway. At the beginning of the game it is programmed so that it develops minor pieces such as knights and bishops as well as trying to spread out and control the middle of the board. However, as the game progresses, StockFish focuses more on the opponent and tries to take initiative by attacking the white and setting "traps" which encourages the opponents to make bad moves.

Alongside this, StockFish stores the chessboard in bitboards where each square is represented by 1 or 0, indicating whether a piece is present or not. It stores each move in 16 bits where bits 0 to 5 stores the destination square (from 0 to 63), bits 6 to 11 stores the origin square (from 0 to 63), bits 12 to 13 stores the promotion piece type, and the final bits store special move indicators (i.e. promotion etc) ^[10]. However, due to the more complicated shape structures in rithmomachia, where more information has to be held regarding the pieces, and as multiple pieces can be build up to form other shapes, I opted out of using bitboards. Instead to store the board and the move information, I have created matrixes.

4.2 AlphaZero

AlphaZero originates from DeepBlue's AlphaGo, and it has been declared as the worldwide strongest AI-performance, when it beat StockFish in a 100-game match. In the match AlphaGo did not lose once, instead it won the game 28 times and drew the rest of the times. The main differences between both relies on the algorithms adapted by both; for example, where StockFish uses an alpha-beta search engine, AlphaZero opts to use a Monte-Carlo Tree Search (MCTS) ^[11].

One of the main advantages of AlphaZero over StockFish is that it replaces the handcrafted knowledge-based minimax search in StockFish with a deep neural network and a general-purpose tree search algorithm. StockFish relies on a handcrafted minimax algorithm, which was designed with 1000's of heuristics implemented by chess champions, whereas AlphaZero learns using a deep neural network.

From this neural network, AlphaZero has taught itself by playing many games against itself the rules of Chess, Go and Shogi ^[12]. For chess, the neural networks takes the board position as an input and produces a vector of move probabilities for each action as an output. AlphaZero learns these move probabilities and from self-play evaluates the usefulness of the move probabilities. It then uses this knowledge to improve the moves in future games.

4.3 Reflections

As rithmomachia is not available on the online platform and an artificial intelligent engine has not been designed for it, my project will significantly vary from both these existing chess engines. However, since I am designing an artificial intelligence game opponent, I have drawn influences from these works. For example, I will initially try and implement a minimax algorithm as done by StockFish and when that successfully works, try and implement a MCTS as implemented by AlphaZero. However, despite AlphaZero, I will not implement the Machine Learning component by creating a deep convolutional residual neural network. This is because of the limited timespan and this might not be feasible due to the requirements needed to implement this.

5. Methodology

There are many different approaches and algorithms that could be used to develop the AI component. From researching different algorithms, here is an explanation of some data structures and methodologies which seems feasible and applicable to this problem.

5.1 Game Trees

Game trees are a nested data structure which represents a directional graph. The root node in a game tree stores the current game state. The child nodes of the game tree represents the states that can be reached from the root node by the current player within the range of a single move. The edges represent the move which is taken to reach from a node to its child node. The leaf nodes are those without any child nodes, i.e. they cannot be expanded further. These represent the terminal states where either the game has been won or there is a draw.

The advantage of game trees is that they map out all the possible moves from a player and all reactions from the opponent ^[13]. This enables the tree to be back-propagated and choose the move which provides them with a higher chance of winning. As well as that, game trees are useful when analysing possible moves, as they store the necessary information required at each stage. For example, each node can store the game state at that point, the evaluation score and whether the node has been fully explored. Therefore, the data structure is ideal when exploring all the possible moves as it stores the information which is needed.

On the other hand, the disadvantage of game trees is that they can increase in size rapidly. For example, at the beginning of rithmomachia, the root node can be expanded to have 22 children nodes. Each one of these children can then have 22 child nodes each and so on. Due to the large

size of the game tree, playing the game can be very slow as the algorithm will search the entire tree. A solution to this is searching only up to a given depth as to reduce the time and computational power required. The more ideal solution is alpha-beta pruning, which is discussed later.

5.2 MiniMax Algorithm

Minimax is a backtracking algorithm which recursively searches through a game tree to find the optimal move in a perfect information game ^[14]. It is widely used in developing AI for 2-player turn-based games, such as Chess and Go. The main concept behind minimax is to minimise the maximum possible loss ^[15].

Minimax algorithm identifies two players – MAX and MIN. When the minimax algorithm is called, a game tree is created starting at the root node. The children of the root node is explored one by one. The tree is expanded till a leaf node is reached, then a static evaluation is performed on the game states to calculate how beneficial the move is for the player. From this, a heuristic value is assigned to the leaf nodes. Larger, positive values from the evaluation will favour the maximising player, and smaller, negative values will favour the minimising player. Depending on which players turn it was to move in the previous stage, either the minimum or the maximum value is chosen from the child nodes. This process is repeated till the root node chooses a value depending on whether it is the maximising or minimising player.

5.2.1 Pseudocode

Here is a pseudocode of the minimax algorithm ^[16]:

```
1. Function Minimax(root, depth, isMaximsing)
2.   if depth == 0 or terminal node reached then
3.     return utility score
4.   if isMaximisng then
5.     maxScore = - Infinity
6.     for each child from root
7.       evalScore = Minimax(child, depth - 1, false)
8.       maxScore = max(evalScore, maxScore)
9.     return maxScore
10.  else
11.    minScore = + Infinity
12.    for each child from root
13.      evalScore = Minimax(child, depth - 1, true)
14.      minScore = min(evalScore, minScore)
15.    return minScore
16. end function
```

5.2.2 Evaluation Function

There are different methods of calculating the evaluation function for games. In simple games, such as tic-tac-toe, it is as easy as assigning -1 if MIN wins a game, + 1 if MAX wins a game or 0 if the game is a draw. For games such as chess, different methods are often adapted. A common one is adding up the total number of pieces left for black and white, and then finding the difference between them. An improvement to this method is to assign different values to the pieces. For example, white pawns have a value of 10 and black pawns -10, and white knight

has a score of 30 and black knights -30. This method is useful as the value of all pieces are not the same. The players have eight pawns each, but only two knights each. Therefore, it makes sense for those to have a higher score so that saving them and capturing the opponents is prioritised. The queen's value is assigned much higher than all other pieces at 900 as this is the most important piece as capturing it means the game has been won or lost ^[17].

5.2.3 Diagram

The diagram on the right illustrates the minimax algorithm. The root node symbolises the current state of the board which the player has already made. The entire game tree is explored before calculating the evaluation function. When a leaf node is reached, the heuristic is calculated. From here, the values are backtracked onto the parent nodes where they choose the maximum or minimum score depending on whose turn it is. Level 1 indicates the opponent's possible moves; it has been reduced to two nodes for simplicity.

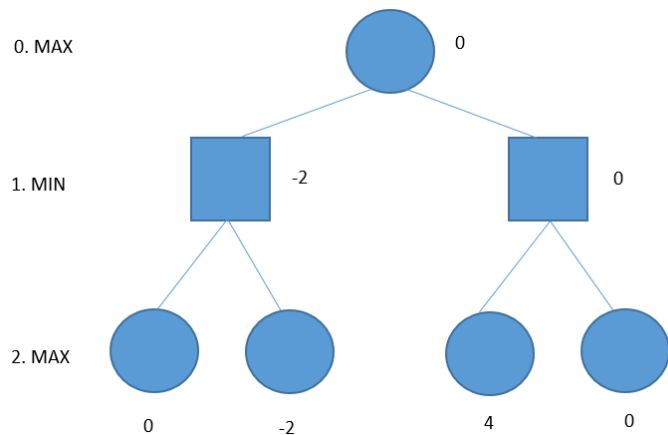


Figure 2: Minimax Algorithm

The next level is the current player's turn where they will either win the game (evaluation = 4) or draw the game (evaluation = 0). Minimax performs a depth-first search where it aims at searching down the levels before searching horizontally.

5.2.4 Drawbacks of MiniMax Algorithm

The main drawback of minimax is that it is really slow as it needs to expand the complete game tree for every single move. This requires large amounts of computational power, space and time. For games such as chess and rithmomachia, the branching factor is huge, therefore, the deeper the tree searches, the slower it becomes.

5.3 Alpha Beta Pruning

Alpha beta pruning is used in conjunction with the minimax algorithm as an optimisation technique. Alpha beta aims to find the optimal minimax solution while avoiding searching subtrees that won't be selected.

It uses two parameters, alpha and beta. Alpha is the best value that the maximizing player currently can guarantee at that level or above. Beta is the minimum lower bound of the possible solutions at that level or above.

5.3.1 Pseudocode

1. Function Minimax(root, depth, isMaximsing, alpha, beta)
2. if depth == 0 or terminal node reached then
3. return utility score
4. if isMaximisng then
5. maxScore = - Infinity

```

6.      for each child from root
7.          evalScore = Minimax(child, depth - 1, false, alpha, beta)
8.          maxScore = max(evalScore, maxScore)
9.          alpha = max(alpha, evalScore)
10.         if beta <= alpha then
11.             break
12.     return maxScore
13. else
14.     minScore = + Infinity
15.     for each child from root
16.         evalScore = Minimax(child, depth - 1, true, alpha, beta)
17.         minScore = min(evalScore, minScore)
18.         beta = max(beta, evalScore)
19.         if beta <= alpha then
20.             break
21.     return minScore
22. end function

```

5.4 Monte-Carlo Tree Search

MCTS is a heuristic search algorithm which uses reinforcement learning to choose an optimal game move. MCTS improves on the tree search algorithm as it aims to combine the benefits of tree search, which is precision, along with the generality of random sampling ^[18]. It consists of 4 stages – selection, expansion, simulation and backpropagation.

MCTS is a relatively new algorithm but it has quickly gained in popularity as it beat the world champion in the game of Go. Quite often minimax along with alpha beta pruning is considered to be sufficient for games such as Chess and Rithmomachia, however, this research explores the different algorithms which are available.

The algorithm builds an asymmetric tree in an incremental manner, performing the four stages at each step. Below is an explanation of the different stages of the algorithm ^[25]:

- *Selection*: This is the first stage of the algorithm and it uses an evaluation function to choose the child node which has the highest probability of winning. It does this by traversing along the current game tree using the Upper Confidence Bound formula. The child node which returns the highest value for this calculation will be selected and then will be passed onto the next stage, which is expansion. Note: the child node which is selected for expansion will always be a leaf node.
- *Expansion*: In this stage, new child nodes are created from the node which was selected. The new nodes represent the state of the board with each possible move from the selected node. This holds true unless a winning state is reached. A node is randomly chosen from the newly created child nodes and passed onto the next stage, which is simulation.
- *Simulation*: Simulation performs a random playout from the child node selected in the expansion stage until a terminal state is reached. A reward value is produced from this simulation depending on whether the game has been won or if there is a draw. These

random playouts are carried out so that different areas of the search space is searched. Also, this method is useful as it does not require domain knowledge.

- *Backpropagation*: The final stage of the process is backpropagation. The terminal nodes, once the simulation is finished, will have either negative or positive scores. Backpropagation traverse up the tree from the child to the parent nodes to update the score of the game played. The variable which stores the total number of games played will also be updated.

On the other hand, the drawback with these random simulation is that they are not realistic as actual players will not make random moves; instead they would make thought out, intelligent moves. There are different methods to enhance these simulations, such as rule-based simulation policy. This enables the programmer to program a domain specific policy so that moves made are rational over random ^[19].

5.4.1 Diagram

The diagram below shows the four stages of the Monte Carlo Tree Search describe above.

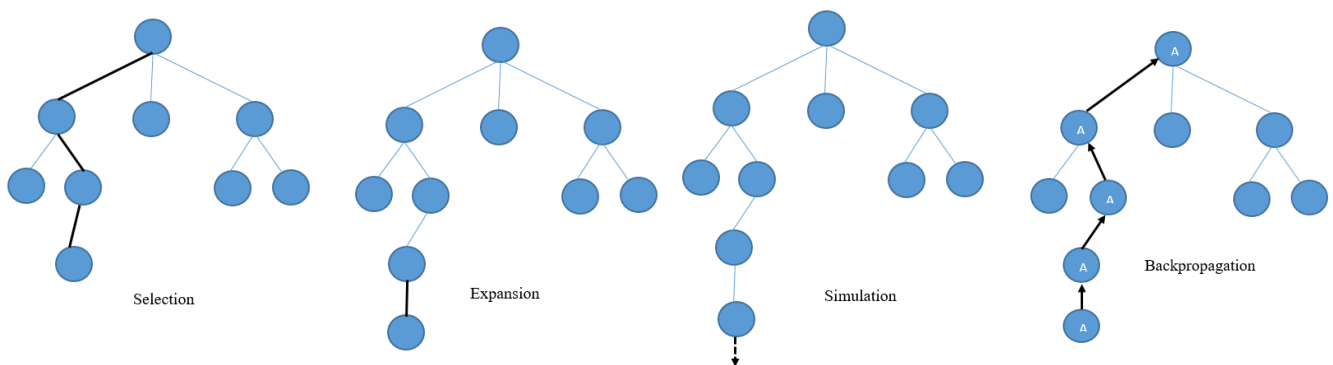


Figure 3: Monte-Carlo Tree Search

5.4.2 Pseudocode

```

1. function mcts(root)
2.   while time < computational budget then
3.     run = run + 1
4.     selection = select_node(root)
5.     expansion = expand_node(selection)
6.     result = simulate(expansion)
7.     back_propagation(expansion, result)
8.
9.
10. function select_node(root)
11.   if root has child then
12.     for each child node
13.       ucb = update_ucb()
14.     return the node with the highest ucb
15.   return root
16.
17.

```

```

18. function expand_node(parent)
19.     possible_moves = getPossibleMoves(parent)
20.     if child unexplored then
21.         expand the child node
22.     return random child
23.
24. function simulate(expansion)
25.     while terminal state not reached
26.         choose random move from node
27.         possible_moves = getPossibleMoves(parent)
28.         create child nodes for possible_moves from parent node
29.         if game won by human then
30.             result = -1
31.         else if game won by MCTS then
32.             result = 1
33.         else if game is a draw then
34.             result = 0
35.     return result
36.
37. function back_propagation(expansion, result)
38.     while root not reached
39.         update value of parent
40.         node_visit + 1

```

5.4.3 Exploration-Exploitation and UCT

When selecting new nodes, it is important to create a balance between exploration and exploitation. Exploration consists of searching a large section of the search space with the aim of finding the most optimal path. On the other hand, exploitation relies on searching only a limited area of the search space

In other words, in tree search, it cannot be guaranteed that the path currently being looked at is the most optimal as the whole search space has not been evaluated. Therefore, the algorithm relies on the exploration - exploitation trade off, which enables it to continue to evaluate other paths while exploiting the current optimal path which has the highest heuristic value. Exploitation is a greedy approach and it focuses on searching the depth of the tree, whereas exploration aims to search the breadth of the tree.

A solution to this is to calculate the Upper Confidence Bound Applied to Trees (UCB), which balances the exploration – exploitation trade off. The formula for UCB is given below.

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln N_i}{n_i}}$$

Figure 4: source [24]

- w_i : Total number of wins for the node after the i^{th} move.

- n_i : Total number of games which have been played after the i^{th} move. The fraction finds the mean of the wins of a node. i.e. (wins/total games)
- c : A constant, the exploration parameter.
- N_i : Total number of games played by the parent of node i .

The first fraction of the formula is the exploitation component and the second corresponds to exploration. UCB relies on the idea that if there is uncertainty in a game move, then it should be assumed that the move is correct, hence it should be exploited further ^[20]. The move with the highest UCB is selected to be expanded and then simulations are run on the node.

5.4.4 Advantages and Disadvantages

One of the main disadvantages of the Monte Carlo Tree Search is as it uses reinforcement learning, the algorithm needs large number of iterations to effectively decide on the most reliable path. This takes computational time and power, which algorithms such as Minimax could avoid. In addition to this, the tree growth becomes rapid after a few iterations. This means that it would need large amounts of memory.

On the other hand, an advantage is that no domain knowledge is required as paths are picked randomly. However, the algorithm can be improved, as mentioned above, by programming in conditions while picking nodes. As well as this, MCTS doesn't expand every single node, which means that less computational time and space is wasted on nodes that are more likely to win to a loss. It supports an asymmetric expansion and have a balance between the exploration and exploitation trade off.

5.4.5 MCTS and AlphaGo

DeepMind's AlphaZero uses MCTS to evaluate the possible moves and choose the optimal one. The MCTS stores four statistics in each edge of the tree, N , W , Q and P , where N is the number of times an action (edge) has been taken from the state (node), W is the total value of the next state, Q is the mean value of the next state (i.e W/N), and P is the prior probability of selecting an action derived from the neural network ^[21].

AlphaZero then carries out the four stages of the Monte Carlo Tree Search. It selects by choosing the action that maximises $Q+U$, where U is a function of P and N , which increases when the branch hasn't been greatly explored. AlphaZero relies on performing more exploration towards the beginning of the game and more exploitation as the game progresses. In the next step, the tree is expanded until a leaf node is reached. The game state of the leaf is then passed to the neural network, which make predictions about the move probabilities and the value of the state for the current player. After this, AlphaZero backs up the previous edges that were traversed to get to the leaf. Finally, it selects a move, either deterministically, for competitive play, or stochastically, for training. The selected move then becomes the new root node and unwanted nodes from the tree are discarded. This process is repeated by AlphaZero till the game is completed ^[22].

6. Design

After conducting extensive research, it was concluded that the programming component of this project can be broken down to three significant parts – developing a text-based game,

developing the graphics for the game and developing the AI for the game. The first part focuses on implementing the rules regarding the movement of the pieces, methods for capturing and ways of winning the game. Developing the graphics involves designing a graphical user interface which provides good human-computer interaction. In other words, the application must be easy for the user to understand and use. Thirdly, developing the AI opponent includes implementing the minimax algorithm, alpha beta pruning and the Monte Carlo Tree Search.

6.1 Initial Considerations

Before beginning the implementation of rithmomachia, it was crucial to understand the rules of the game. This required researching through multiple journals, books and websites and combining the information provided by them. This is due to the lack of information available regarding rithmomachia as it is not a popular game. As well as that, the references often did not support each other as multiple authors documented the rules differently in the medieval ages when the game was prevalent. Therefore, significant time was spent before programming the game, researching into the different rules and the variations between them. The game which was developed was mainly focused on the paper published by *The American Mathematical Monthly* ^[26]. Other sources were used to support the instructions given here and fill in rules which is not stated here.

As well as that, when planning the project, significant amounts of time were dedicated to researching into different artificial intelligence algorithms. The main reason behind this was my inexperience in this field. Having never programmed an AI, it was understood at the beginning that significant research must be conducted as to how to approach the programming, rather than beginning to program straight away. Therefore, when planning the project, time for research into AI was assigned at the beginning of the project and at the beginning of when AI was ready to be implemented.

6.2 Early Design Concepts

The problem which the project addresses is to develop a game of Rithmomachy and create an artificial intelligent opponent for the game, which has a high chance of winning any given game. To address the issue, the solution which was concluded on after conducting research was to implement the game using Python and develop the graphics using Pygame. After this, the AI feature will initially be developed using minimax. Due to the lack of efficiency of the algorithm, this would not guarantee on winning most of the matches played. The algorithm will then be improved by implementing alpha-beta pruning; this increases the efficiency of the AI developed. Finally, it was planned to implement the Monte-Carlo Tree Search, which is a strong AI algorithm, hence increases the probability of the games played being won.

The game will be implemented in Python 3.7 using the IDE PyCharm. Various approaches could be taken to develop the graphics for the game. The most basic would be to use Pygame, which is an open-source Python programming library build on top of SDL and is used for graphics development. Another approach would be to use a game engine, such as OpenGL. If this is the case, I will have to install PyOpenGL, which is a Python OpenGL binding. The main benefits of using Pygame is that it is easy to learn and there is lots of support and tutorials available online. As well as that, Pygame should be sufficient for a 2d game development. Therefore, I have decided to use Pygame over game engines as it will be easier and more convenient for this scenario.

Libraries such as NumPy and math was also planned to be used to calculate formulas for functions such as the evaluation function. A matrix will be implemented to store the board. As well as this, a transposition table will be implemented to store results of previous searches in the form of a hash table. All code will be uploaded to a Gitlab repository.

The figure below shows a simple design of the user interface of the game; this was included in the interim report and the final product is a modified version of this. In the plan, the pieces captured are visible to the user so they are aware of the progress of the game. The green squares indicates the possible moves the piece selected can make; since the game has a complicated set of rules, this should aid the player in considering all their choices hence making the game easier for them. The bar at the bottom indicates how strong the moves made are. This was inspired from *chess.com* ^[23], which is a famous online chess platform. A help button is also provided which will display the instructions to the user.

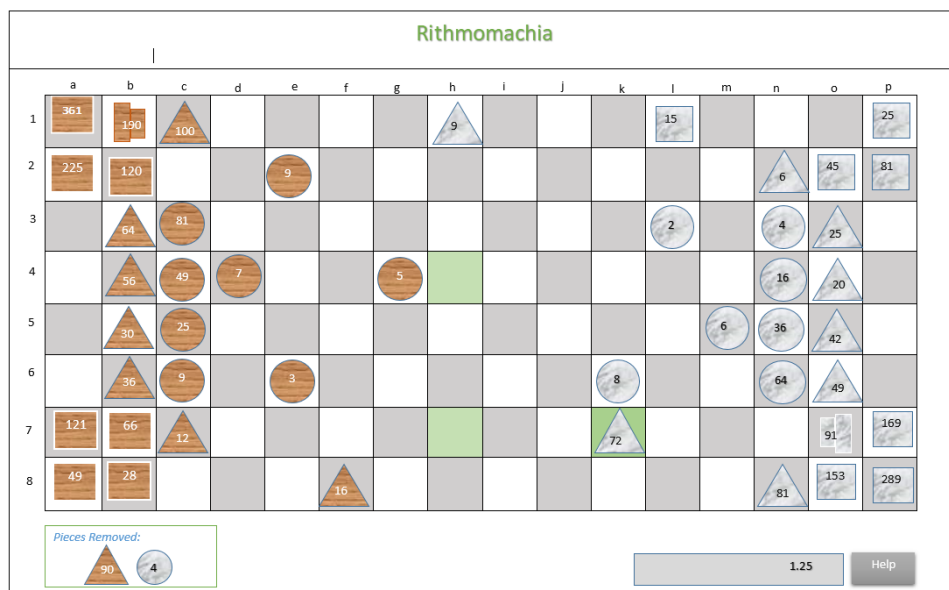


Figure 5: Initial design plan of the user interface

6.3. Move Generation and Position Evaluation

Another significant part of designing the AI was planning the move generation and the position evaluation. In game play, move generation is the method of choosing a move out of all possible moves. This is easier in games such as tic-tac-toe and connect-four where choosing a move is as simple as checking which positions in the board are empty. In tic-tac-toe, a move could be assigned -1 if it causes the opponent to win, +1 if it causes the player to win and 0 if the game is a draw. This is easy to implement. However, with games such as chess and rithmomachia, all the different pieces on the board must be evaluated, along with all the different possible moves every single piece could make. This is partially why game trees for games like chess grow exponentially the more spaced out the pieces become.

In the design phase, the position evaluation was determined to be based on the following factors:

1. If a terminal game state is reached, assign the move + infinity or – infinity depending on the player making the move

Reaching a terminal node determines the end of the game, therefore, it is highly preferred. If it is the opponent winning, a $-\infty$ score is given so that the path can be avoided when choosing between child nodes with different evaluation scores.

2. If there are no preferential moves, then let the first row move forward to form arithmetic progression in the opposite side of the board.

If the evaluation score for all the different moves are the same, then allow the pieces on the first row to move forward. This is because the pieces on the first row (3, 5, 7, and 9 for the white player) form an arithmetic progression. A victory is formed once this progression is made on the opposite side of the board. It was also planned that once the white piece reaches the x coordinate of 8, a point should be deducted from it so that the piece above it (R, 5, 0) is chosen to move forward.

3. Moves where multiple captures are possible are preferred over moves with single or less captures.

Capturing pieces is a technique of winning the game, therefore, when evaluating possible moves, positions and moves where the opponent's pieces can be captured should be given additional points. The more captures possible by a single move, the more points should be assigned to that move/position.

7. Implementation

7.1 Rithmomachy

Flow chart showing the implementation of the game logic.

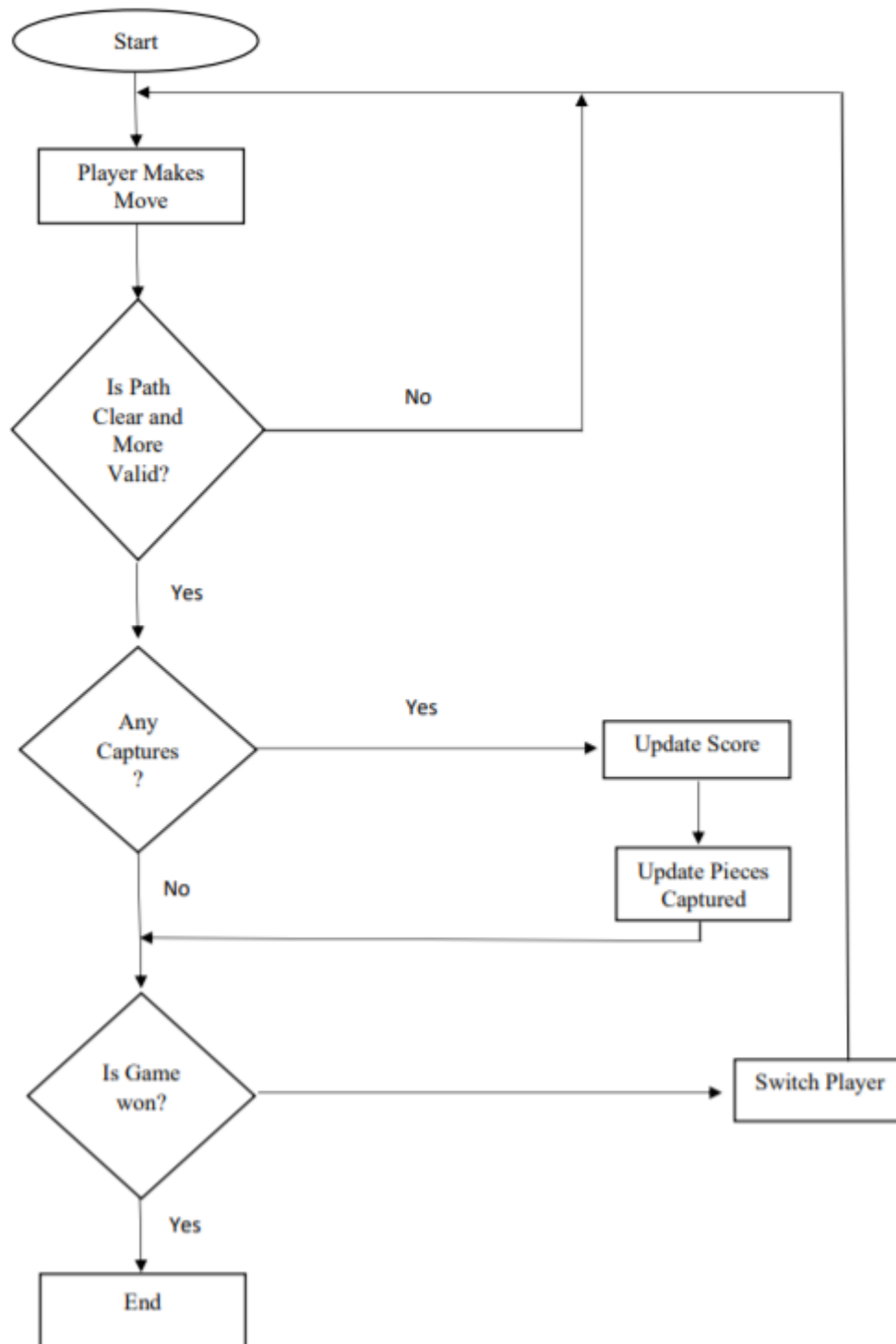
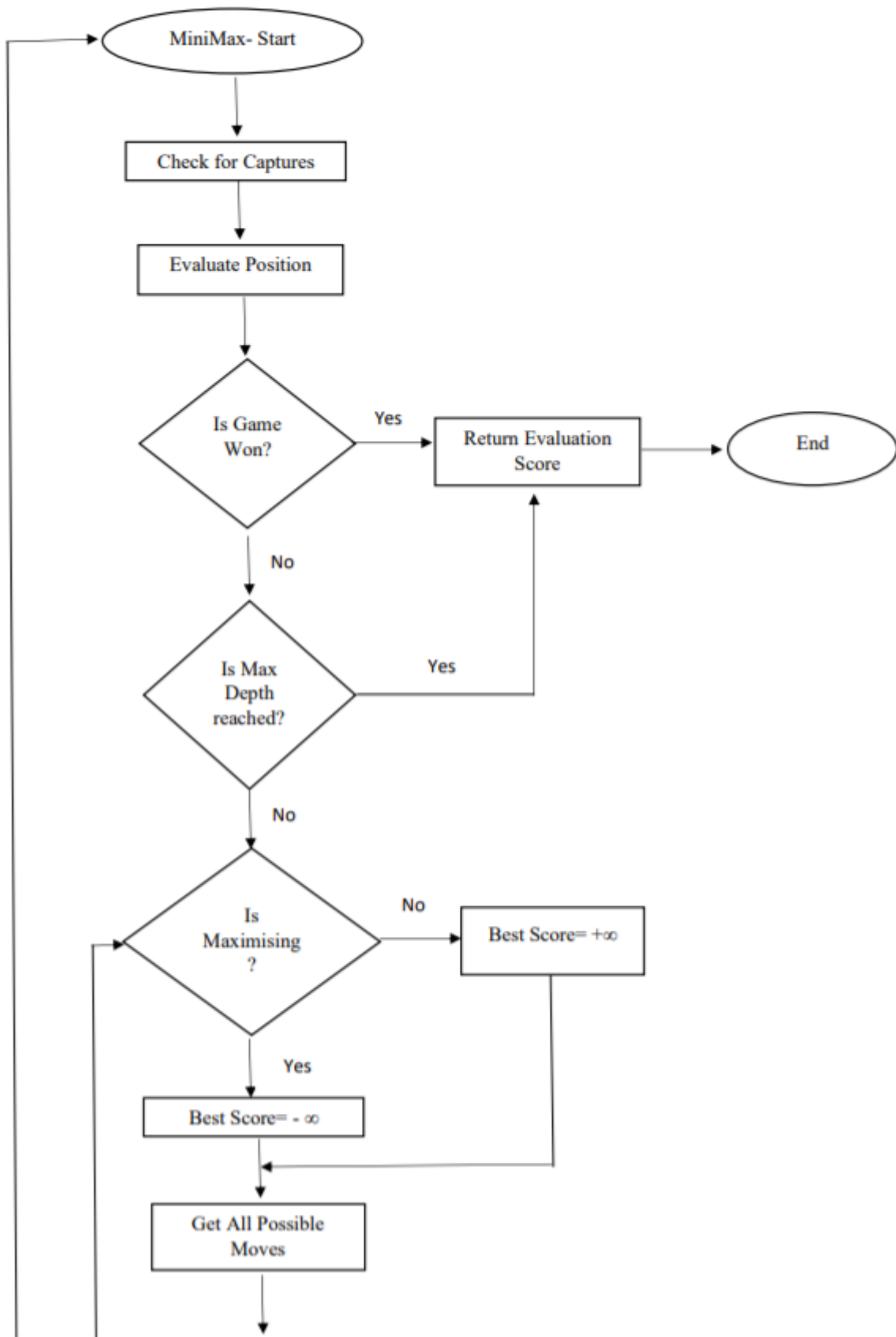


Figure 6: Flowchart of the game logic

7.2 MiniMax



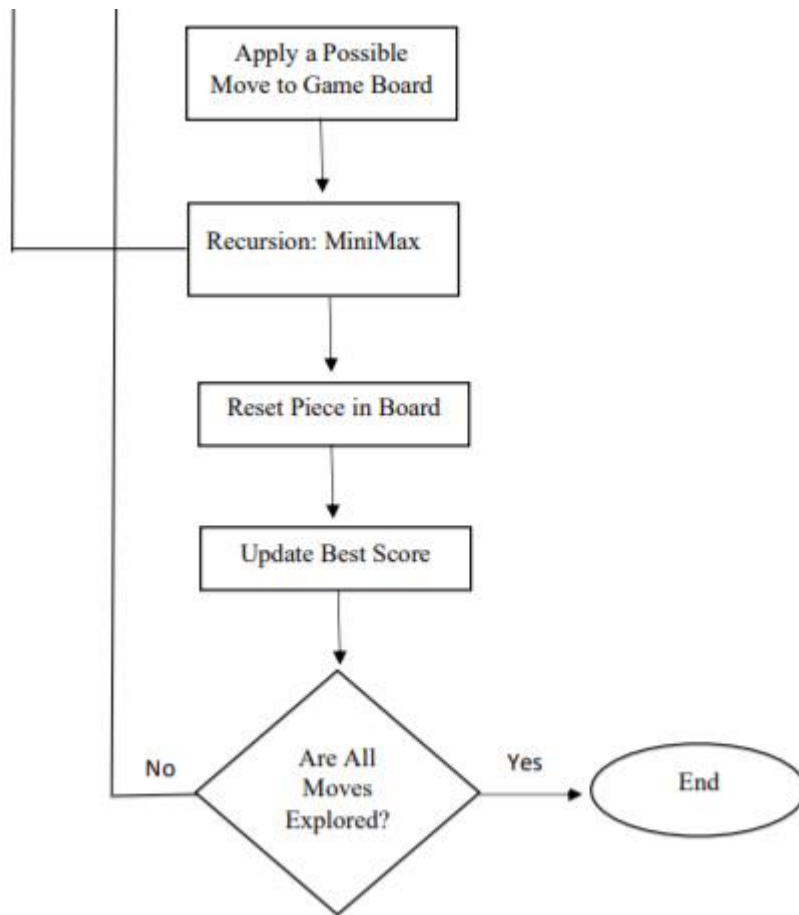


Figure 7: Flowchart of the minimax algorithm

The flow chart above illustrates the process of calling the minimax function. It is a visual representation of the AI program provided for ease of understanding and for simplicity.

8. Evaluation

8.1 Evaluation of Game Features

The table below shows all the different features of rithmomachia which needed to be developed. It shows that all the features of the game which was agreed on with the supervisor has been successfully completed.

Category	Feature	Implemented	Tested	Note
General	Store game board	✓	✓	Game board is stored in a Matrix.
	Create shapes	✓	✓	
	Pyramids	✓	✓	Pyramid consists of other shapes, hence, was more

			challenging to program
	Move Validation	✓	✓
	Validation for accessing Matrix	✓	✓
<i>Captures</i>	By Meeting	✓	✓
	By Assault	✓	✓
	By Ambush	✓	✓
	By Seige	✓	✓
<i>Common Victories</i>	By Goods	✓	✓
	By Body	✓	✓
	By Lawsuit	✓	✓
	By Honor	✓	
<i>Glorious Victories</i>	By Honor and Lawsuit	✓	✓
	Victory by Progressions	✓	✓

8.2 Minimax Game Tests

The table below shows four different tests which has been conducted on the artificial intelligent component designed. The AI being tested is the minimax algorithm with alpha beta pruning.

<i>Test ID</i>	<i>Purpose</i>	<i>Winner</i>	<i>White Pieces Captured</i>	<i>Black Pieces Captured</i>	<i>Notes</i>	<i>Test Pass (Y/N)</i>
1	Test if the AI is responsive to user interaction	AI	(T, 12, 0)	(T, 72, 1), (R, 8, 1), (T, 25, 1), (T, 9, 1), (R, 2, 1)	(T, 72, 1) was captured by Assault, rest by meeting. Game won By Body	Y
2	Test how many steps ahead the AI would "plan"	Human	(R, 9, 0), (R, 7, 0), (R, 5, 0), (R, 3, 0), (R, 9, 0)	(R, 8, 1), (R, 6, 1)	All pieces were captured by Meeting. Game won By Body	N
3	Check if AI could beat a player making random moves	AI	(T, 12, 0)	(T, 81, 1), (S, 153, 1), (T, 72, 1), (R, 64, 1)	Random moves were picked for the human player. Game won By Goods	Y
4	Testing if the AI will prevent a	AI	(R, 3, 0), (R, 5, 0), (T, 16, 0)	(R, 6, 1), (R, 4, 1), (T, 72, 0)	Attempt by player to make an arithmetic	Y

progression from forming			1), (T, 25, 1), (R, 2, 1)	progression on the opposite side of the board. Game win By Body	
--------------------------------	--	--	------------------------------	---	--

From testing the AI multiple times, the weakness in the AI was discovered. As shown in the diagram below, (R, 3, 0) still moved to (7, f) from (6, f) even though it is the black player's turn next and (R, 8, 1) at (9, f) could move to (8, f) and capture the white piece. This could be explained as the depth of the search was set to two, which means that deeper levels or moves

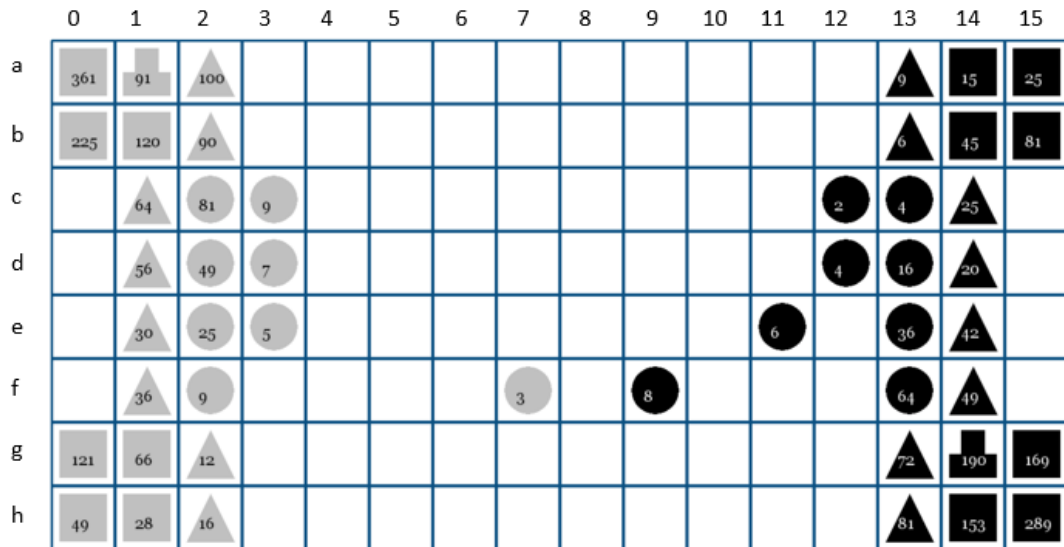


Figure 8: Weakness in the algorithm

were not searched. As shown in test 2, this weakness is more evident if the player only moves round shapes. This could be because as round shapes can only move one square each time, the area it searches is much smaller, compared to shapes such as triangles which can search up to three squares in all directions.

8.3 Game Experience Questionnaire – Core Module

Due to the pandemic I had to complete the user testing by myself, contrary to the initial plan.

From the questionnaire completed (appendix A), it can be analysed that the game challenged my thinking as it required planning ahead of the opponent and making rational decisions. This is shown by the statement “I felt challenged”, where a score of four was given. As this is the maximum score, it can be understood that the game was extremely challenging.

In addition to this, it also required high levels of concentration hence it was easy to get involved in the game. Though there was no time limit for each opponent to make the move, I felt pressured to act and think fast. This was the fun aspect of it. Furthermore, I felt a sense of appreciation towards the game due to the complexity of the rules and regulations involved. However, this also made me feel tired quickly as once the game has started it was easy to be lost in it.

9. Summary and Reflections

Looking back at the requirements specified in the interim report, I believe that the game development has been completed quite well. On the other hand, graphics has scope for a lot of improvement. Regarding artificial intelligence, the minimax and alpha beta pruning algorithms implemented are sufficient as they win the majority of the games played. However, taking into consideration that MCTS was initially planned to be programmed, it would have been interesting to implement it and then compare the performance of the different algorithms.

10. Project Management

10.1 Initial Plan

The project was designed with the waterfall software methodology in mind. The programming component of this project can be broken down to three significant parts – (A) developing a text-based game that works, (B) developing the graphics for the game and (C) developing the AI for the game. These are colour coded and shown in the table and the Gantt chart below.

Component	ID	Task	Completed
Planning	1	Propose a project plan to supervisor	100%
Documentation	2	Complete the ethics checklist	100%
C	3	Research into existing AI for games	100%
A	4	Develop a text-based board, players and pieces	100%
A	5	Program the enforcement of game rules	100%
Documentation	6	Write the Interim Report for dissertation	100%
A	7	Implement the winning conditions – common and glorious victories	100%
B	8	Develop the graphics for the game	80%
C	9	Further research into AI and algorithms	100%
C	10	Develop the MiniMax algorithm	100%
C	11	Develop Alpha-Beta Pruning	100%
C	12	Develop Monte-Carlo Searching	-
C	13	Improve the AI further	-
Testing	14	Test the program	50%
Documentation	15	Write final dissertation	100%

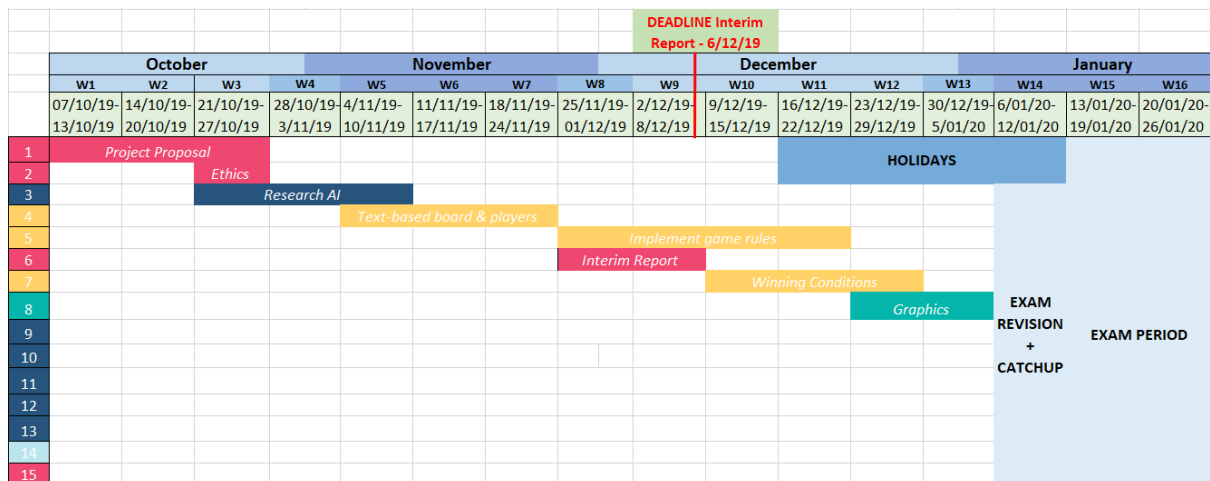


Figure 9: Project Plan (Modified version from Interim Report) - Part 1/1

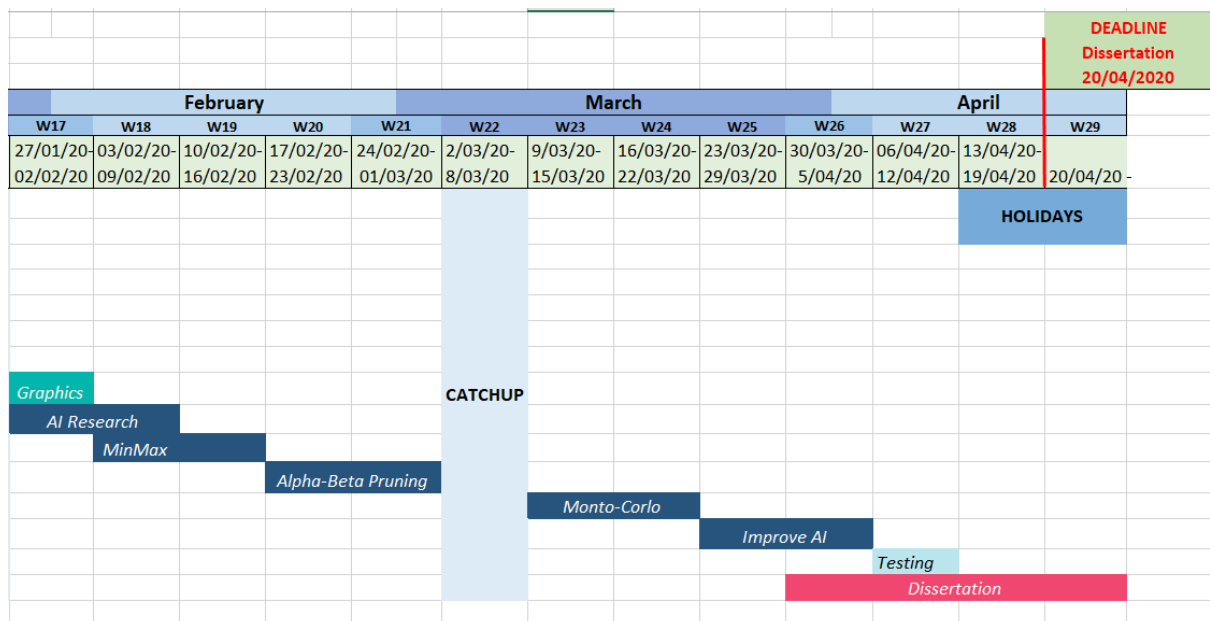


Figure 10: Project Plan (Modified version from Interim Report) - Part 2/2

From the initial stages of the project, planning and time management were identified as crucial skills to complete the project within the deadline. Therefore, as soon as the project requirements were agreed on with the supervisor, a plan was created outlining internal deadlines for all the sub-tasks. Therefore, I have been successful in conducting sufficient research to aid in the development of the game and for choosing which algorithms to implement.

In the table, the progress for testing is shown as 50% because it was initially planned to be conducted by a group of individuals; half of whom are keen board game enthusiasts and the other half relatively new to playing board games. However, due to pandemic and the lockdown, I was not able to do this. Instead, the testing of the software was conducted by myself.

11. Contributions and Reflections

11.1 Self Reflection

Working on this project for the past year has shaped my ability to take initiative and be proactive. I have successfully arranged meetings with my supervisor throughout the year and ensured that he was updated about the progress that was being made. I also enjoyed researching into the game as it was exciting to rediscover something which not even many board-game enthusiasts are aware of. It was also enjoyable to combine the different variations of the rules as most were not written in great detail.

The thing that challenged me the most is time management. Though I had set myself internal deadlines with a goal to complete certain aspects of my final project, I was unable to always achieve this because I underestimated the amount of time required for each internal deadline. This meant that I had several incomplete internal deadlines and so I had to spend more time than presumed on each piece of work. For example, the development of the game required more time to complete than what I initially presumed. This is because I had several errors in my coding that I had to identify and fix. Therefore, for my future submissions, I aim to set more realistic goals that are more achievable.

In addition to this, towards the end of the project, I was unable to complete tasks such as developing the MCTS algorithm and performing user testing. A reason that I was unable to implement MCTS is poor time management as I spend a long time on game development, which should have been put aside while progress was made on creating this. In addition to this, I was unable to conduct user testing because of the current global pandemic. Instead I attempt to evaluate the software by myself and this is provided above.

11.2 Improvements on the project

I am satisfied with the implementation of the game as it matches the functional requirements which were made during the project proposal. However, areas of the project which could still be improved are graphics and the AI.

To further improve the graphics, a tutorial page could be provided for the user where they can experiment on a scaled-down game board by moving the four different shapes one after the other. This interaction will help them to adapt to the software and to understand the rules of the game. As well as that, it would be useful to have a “home” button on the game pages so they can easily navigate between the game and the main page. Alongside this, an issue with the current design is that once a position is clicked, the user is not shown any indication that a position has been clicked. This is poor human-computer interaction as the user should always be made aware of the results of their actions.

To improve the AI, a better utility function could be implemented as discussed above. In addition to this, implementing MCTS would have been the more optimal solution as it uses reinforcement learning to search the game tree.

12. Bibliography

- 1) 1. Wood, L. (2018) 'Global Board Games Market Outlook and Forecast 2018-2023: Major Players are Asmodee Editions, Hasbro, Mattel & Ravensburger', PR Newswire, 07 August.
Available at: <https://www.prnewswire.com/news-releases/global-board-games-market-outlook-and-forecast2018-2023-major-players-are-asmodee-editions-hasbro-mattel--ravensburger-300693174.html>
- 2) Moyer, A. E. (2001) 'The Philosophers' Game, Rithmomachia in Medieval and Renaissance Europe', United States of America: University of Michigan.
- 3) Shannon, C.E. (1950) 'Programming a Computer for Playing Chess', *Philosophical Magazine*, Ser.7, Vol 41, No. 314, Taylor & Francis, Ltd
- 4) Samuel, A.L (1959) 'Some Studies in Machine Learning Using the Game of Checkers', *IBM Journal of Research and Development*, Vol 3, no. 3, IBM.
- 5) Silver, D. and Hassabis. D. (2017) 'AlphaGo Zero: Starting from scratch', DeepMind, 18 October. Available at: <https://deepmind.com/blog/article/alphago-zero-starting-scratch>
- 6) *Diagram From:* Tomas, H.M and Borquez. D, (2015) 'Breve Historia de los Juegos de Mesa', Ludoteca de Pampala Press. Available at: <https://ludotecapampala.wordpress.com/2016/06/01/rithmomachia-bhjm-11/> [Accessed 5 May 2020].
- 7) Newton, D. P. (1984) 'Rithmomachia', *Mathematics in School*, Vol 13, no. 2, The Mathematical Association.
- 8) Smith, D.E and Eaton, C.C, (1911) 'Rithmomachia, the Great Medieval Number Game', *The American Mathematical Monthly*, Vol 18, No. 4, Taylor & Francis, Ltd
- 9) Ray, C. (2019) *How Stockfish Works: An Evaluation of the Databases Behind the Top Open-Source Chess Engine – good fibrations*. [online] Rin.io. Available at: <http://rin.io/chess-engine/> [Accessed 2 May. 2020].
- 10) Romstad, T. et al (2018) Stockfish [online] github.com. Available at: <https://github.com/official-stockfish> [Accessed 3 May. 2020]
- 11) Anton, R. (2018) 'Revaluation of AI engine alpha zero, a self-learning algorithm, reveals lack of proof of best engine, and an advancement of artificial intelligence via multiple roots', *Open Access Journal of Mathematical and Theoretical Physics*, Vol 1, no 2, MedCrave.
- 12) Silver, D., Hubert, T., Schrittwieser, J. and Hassabis, D. (2019). *AlphaZero: Shedding new light on the grand games of chess, shogi and Go*. [online] Deepmind. Available at: <https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go> [Accessed 6 April. 2020].
- 13) Kwanghui, n.d. Segment 5: Strategic Thinking. [online] Kwanghui.com. Available at: http://kwanghui.com/mecon/value/Segment%205_5.htm [Accessed 28 April 2020].
- 14) Eppes, M., (2019). How A Computerized Chess Opponent "Thinks"—The Minimax Algorithm. [online] Towardsdatascience. Available at: <https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1> [Accessed 8 May 2020].
- 15) Kosciuk, K. (2009). Is Minimax really an optimal strategy in games? [online], Poland: Bialystok University of Technology
- 16) Javatpoint, n.d. Artificial Intelligence | Mini-Max Algorithm - Javatpoint. [online] www.javatpoint.com.

- Available at: <https://www.javatpoint.com/mini-max-algorithm-in-ai> [Accessed 13 May 2020].
- 17) Hartikka, L., 2017. A Step-By-Step Guide To Building A Simple Chess AI. [online] freeCodeCamp.org.
Available at: <https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/> [Accessed 6 May 2020].
 - 18) Browne, C., Powley, E., Whitehouse, D., Lucas, S. and Cowling, P., 2012. A Survey Of Monte Carlo Tree Search Methods - IEEE Journals & Magazine. [online] Ieeexplore.ieee.org.
Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6145622> [Accessed 13 May 2020].
 - 19) Roy, R., 2018. ML | Monte Carlo Tree Search (MCTS) - Geeksforgeeks. [online] GeeksforGeeks.
Available at: <https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/> [Accessed 14 May 2020].
 - 20) Samishwal, n.d. Upper Confidence Bound Algorithm In Reinforcement Learning - Geeksforgeeks. [online] GeeksforGeeks.
Available at: <https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/> [Accessed 17 May 2020].
 - 21) Yannakakis, G.N and Togelius, J. (2018) 'Artificial Intelligence and Games', Springer
 - 22) Silver, D. et al (2018) 'A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play', DeepMind, 06 December.
Available at <https://deepmind.com/research/publications/general-reinforcement-learning-algorithm-masters-chess-shogi-and-go-through-self-play>
 - 23) Chess.com. (2019). *Chess.com - Play Chess Online - Free Games*. [online] Available at: <https://www.chess.com/> [Accessed 6 Dec. 2019].
 - 24) *Diagram From*: Wikipedia, n.d. Monte Carlo Tree Search. [online] En.wikipedia.org. Available at: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search#Exploration_and_exploitation [Accessed 15 May 2020].
 - 25) Sharma, S. (2018) 'Monte Carlo Tree Search', *Towards Data Science*, 01 August. Available at: <https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa>
 - 26) Smith, D.E and Eaton, C.C, (1911) 'Rithmomachia, the Great Medieval Number Game', *The American Mathematical Monthly*, Vol 18, No. 4, Taylor & Francis, Ltd

13 Appendix A

Game Experience Questionnaire – Core Module

Please indicate how you felt while playing the game for each of the items, on the following scale:

Not at all	Slightly	Moderately	Fairly	Extremely
0	1	2	3	4
< >	< >	< >	< >	< >
1. I felt content				[3]
2. I felt skilful				[4]
3. I was interested in the game's story				[3]
4. I thought it was fun				[4]
5. I was fully occupied with the game				[4]
6. I felt happy				[2]
7. It gave me a bad mood				[1]
8. I thought about other things				[3]
9. I found it tiresome				[2]
10. I felt competent				[3]
11. I thought it was hard				[2]
12. It was aesthetically pleasing				[2]
13. I forgot everything around me				[4]
14. It felt good				[3]
15. I was good at it				[3]
16. I felt bored				[1]
17. I felt successful				[3]
18. I felt imaginative				[4]
19. I felt that I could explore things				[3]
20. I enjoyed it				[3]
21. I was fast at reaching the game's targets				[3]
22. I felt annoyed				[1]
23. I felt pressured				[3]
24. I felt irritable				[2]
25. I lost track of time				[3]

26. I felt challenged	[4]
27. I found it impressive	[3]
28. I was deeply concentrated in the game	[4]
29. I felt frustrated	[2]
30. It felt like a rich experience	[2]
31. I lost connection with the outside world	[2]
32. I felt time pressure	[3]
33. I had to put a lot of effort into it	[4]