|  |
| --- |
| **COVENTRY**  UNIVERSITY |
| Faculty of Engineering and Computing |
| Department of Aerospace, Electronic and Electrical Engineering |
| Computer Hardware and Software Engineering |
| 306AAE Individual Project |
| Noughts and Crosses Advanced AI |
| Author: Alexander Moses |
| SID: 6277932 |
| Supervisor: Chris Bass |
| Submitted in partial fulfilment of the requirements for the Degree of Bachelor of Computer Hardware and Software Engineering |
| Academic Year: 2017/2018 |

Abstract

The project is the creation of an AI for an advanced version of noughts and crosses, with the foundation of the AI being the minimax algorithm to generate possible game states to evaluate a viable one, and using both alpha-beta pruning and rotational symmetry to improve the AI.

The AI was created in stages and therefore there are 4 different AIs for four different types of noughts and crosses (3x3, 9x9, 3x3 Squared Simple, and 3x3 Squared Advanced) and a game system to allow a player to play against any one of these 4 AIs, to play against a second player in one of the 4 types of noughts and crosses or to watch two AIs play against each other in any of the 4 types.

Three sections were researched for the AI; Minimax algorithm in different contexts (Mathematical and multicore processing), Alpha-Beta pruning in different contexts (theoretical knowledge and board game), and Rotational Symmetry. The research used helped create the three sections into the base 3x3 AI. In addition to the research required to make the base AI, the idea of a deep learning AI was research for proof of concept, so that it could in turn replace the other AI (pre-programmed) in the future.

The research was used to create some pseudo code schematics for the algorithms and how they interacted with each other, to be used to create the base 3x3 AI. Once done it was then adapted into the three other versions, one after the other. For the Squared versions, extra algorithms and code sequences was used to improve the AI to best operate the unique nature of the 3x3 Squared board.

The AI was tested regularly throughout its development with two methods, for the AI to play against a copy of itself, and for the AI to play against the developer. Whilst the AI was sufficient in its decision making, it did not live up to the expectation that was outlined in the project brief, this is due to the limitation of the minimax algorithm and the size of the board.

However, the AI is enough to be used as a training AI once the deep learning AI has been attempted. Many other improvements have been considered for the future, such multicore processing, GPU processing, hash maps and the Android interface mentioned in the project brief.

Some sections of the project were completed quicker than expected, but overall the project took longer due to the fact that isolating the problems took a rather long time, but solving the problems once identified was quick.

Table of Contents

[Abstract 1](#_Toc512868387)

[Table of Contents 2](#_Toc512868388)

[1 Introduction 3](#_Toc512868389)

[1.1 Background to the Project 3](#_Toc512868390)

[1.2 Project Objectives 4](#_Toc512868391)

[1.3 Methodology 4](#_Toc512868392)

[2 Literature Review/Theory 5](#_Toc512868393)

[3 Design 11](#_Toc512868394)

[4 Implementation 18](#_Toc512868395)

[5 Testing and Results 23](#_Toc512868396)

[6 Project Management 24](#_Toc512868397)

[6.1 Project Schedule 24](#_Toc512868398)

[6.2 Risk Management 27](#_Toc512868399)

[6.3 Quality Management 27](#_Toc512868400)

[7 Critical Appraisal 28](#_Toc512868401)

[8 Conclusions 29](#_Toc512868402)

[8.1 Achievements 29](#_Toc512868403)

[8.2 Recommendations for Future Work 30](#_Toc512868404)

[9 Student Reflections 31](#_Toc512868405)

[10 References 32](#_Toc512868406)

[Appendix A – Interim Progress Report 1](#_Toc512868407)

[Appendix B – Class Diagram 1](#_Toc512868408)

# Introduction

## Background to the Project

Noughts and Crosses Squared Advanced is an adaptation of the game nought and crosses (tic tac toe) of which its one large 3x3 grid with each square in that grid having a smaller 3x3 grid. (refer to figure 1.1)

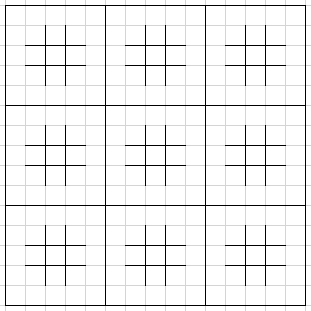


Figure 1.1: Design of Squared Board (Empty)

In this game, the game starts in the middle grid, of which the player chooses a square out of the 9. Whichever square they choose to occupy sends the other player to that corresponding grid (refer to figure 1.2)

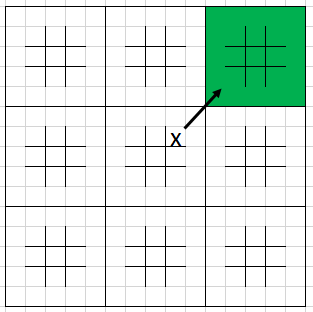


Figure 1.2: Demonstrating Move on Board 1

A player wins a smaller grid by getting three squares in a row (like original noughts and crosses), and a player wins the game by getting three smaller grids in a row. If a player gets sent to a grid that has already been claimed or is full then that player may claim a square in any grid.

The project is to make an AI for a human player to play against with it making its moves within a timely fashion, in addition, a game system will be created handling all the rules and restrictions, and allowing for two people to play against each other instead of an AI. On top of the game system, there are two bonus objectives; the first is creating an Android interface for the game system so that it can be played on a phone or tablet.

The second is a deep learning AI, instead of a pre-programmed AI, this AI would learn from its games and improve over time. However, this AI would be several times more advanced and would also take some time to train it, therefore, this AI is just a proof of concept.

## Project Objectives

The requirements for the final project is that the AI is winnable in a minimum of 9/10 games on an android platform. The project will be broken down into stages to help with its completion, completing each one in succession until the final project is complete. The Primary stages are as below:

* Develop the game and game rules for player vs player
* Develop an AI for 3x3 noughts and crosses
* Develop an AI for 9x9 noughts and crosses winnable by 3 in a row
* Develop an AI for 3x3 inside of 3x3, winnable by claiming one square
* Develop an AI for 3x3 inside of 3x3, winnable by claiming squares in a noughts and crosses fashion

With the secondary stages (non-compulsory), as below:

* Developing it as an Android phone application

With a bonus but non-compulsory stage:

* Create a Deep Learning AI for the 3x3 Squared Advanced

## Methodology

The main framework of the AI is to use the minimax algorithm, this is because it is the prime AI algorithm used in both chess Ai and other noughts and crosses AI because of its ability to generate all (or most) game states for every possible move, thus allowing for an accurate move without a highly complex algorithm.

By the design of the game, the minimax algorithm will have to be improved and many algorithms work in this situation, the top being Alpha-beta Pruning the top reduction algorithm associated with the minimax algorithm.

As the game is based on a symmetrical grid, the concept of rotational symmetry can be used on the Ai to further improve the minimax algorithm.

# Literature Review/Theory

The following Literature Review/Research is from my Interim progress report (Moses, 2018). The interim progress report is included in this document, Appendix A.

One source was used for three sections (minimax, alpha-beta and neural networks) and that was a book called “*Introduction to Artificial Intelligence”* by Wolfgang Ertel (2011). It covers all areas of AI including the three included in this project, it covers the concepts with straightforward explanations, with diagrams, explanations and algorithms in pseudo code. This book (Ertel 2011) was particularly useful for gaining an understanding of all areas of AI useful for starting the research of this project, as a basic knowledge and understanding is required to identify the usefulness of a source.

**Minimax Algorithm**

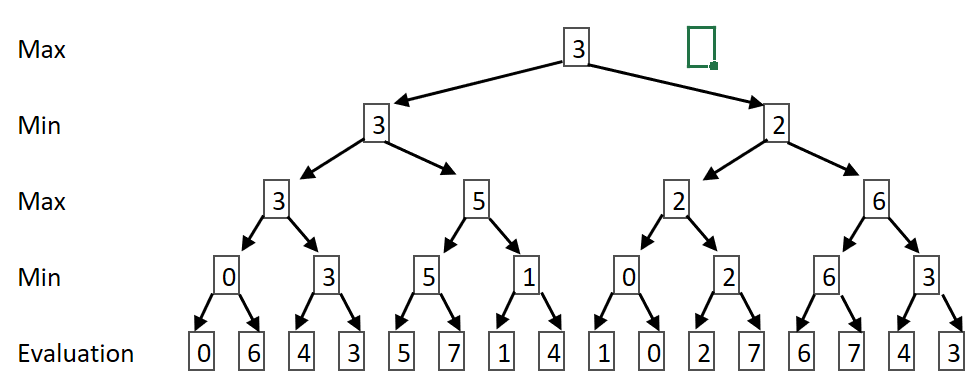
The minimax algorithm is a method for minimizing the possible worst-case scenario and is used in decision theory, game theory, statistics and philosophy. The algorithm in this case if for game theory, to be used in a pre-programmed AI for it to calculate the best move by attempting to minimize the potential ‘score’ that the opposing player can get against the AI. In simple terms, it generates a score for every possible move that the AI can make with every possible move the player can make proceeding the AI makes theirs until every combination of moves has a calculated score, for the AI’s turn it attempts to obtain the maximum score and for the player’s turn it attempts to obtain the minimum score (refer to figure 2.1).

Figure 2.1: Minimax Decision Search Tree 1

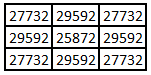
Once implemented the AI evaluates a total amount of 255168 times (the same amount as the number of unique tic tac toe games), the average number of evaluations is 28352 if the player goes first. The table to the left (figure 2.2) is the number of evaluations depending on where the player went first (i.e. top left square is the number of evaluations if the player’s first move is in the top left square)

Figure 2.2: Minimax Search Count 1

The Research required to gain an understanding of this algorithm and how to create one to suit the needs of the project will be two published papers. One being *“Rminimax: An Optimally Randomized Minimax Algorithm”* by Silvia García Díez, Jérôme Laforge, and Marco Saerens (2013), and the other being *“Efficiency of Parallel Minimax Algorithm for Game Tree Search”* by Plamenka Borovska, and Milena Lazarova (2007).

*“An Optimally Randomized Minimax Algorithm”* (García Díez, Laforge, and Saerens 2013) is a paper highlighting the mathematical side of the concept of the algorithm rather than an already created generic algorithm, and so it contains the possible ways that the algorithm can be constructed with full explanations of the mathematical formulas needed to create the algorithm. This is particularly useful as instead of a paper explaining how to create the algorithm, it explains the math behind it so the AI can be personally tailored to the project in comparison to a generic minimax algorithm, allowing for better performance and better results.

The only downside to this is that it is only the mathematical side it does not highlight at all the impact that the algorithm will have on a computer, such as performance or efficiency. That is where the other paper comes in, “*Efficiency of Parallel Minimax Algorithm for Game Tree Search”* (Borovska, and Lazarova 2007) covers the computational side of it. The algorithm, if complex enough, would need to make a very large amount of calculations and that can take time, so to reduce the time constraint, the ‘tree’ that the algorithm is searching can be broken down into sections and said sections are run on different cores of the processor, allowing them to be done in parallel. This reduction of time is limited to the number of cores the device (computer or phone or tablet) has, and the speed of the cores. This paper highlights the ways that the algorithm can be computed in parallel, the advantages and disadvantages to each way, and the efficiency of parallel computing the algorithm. However, this paper alone will not help construct the algorithm, only allow the search to be sped up by utilising the multi-core processor that modern-day devices have, and so both papers together will be needed to make the most optimal algorithm.

**Alpha-Beta Pruning**

The minimax algorithm works by creating a tree of nodes for all combinations of moves, with an increasing level of depth. Alpha-beta pruning is an algorithm to decrease the number of nodes in the tree created in the minimax algorithm. It does this when it encounters a node that it deems that it will not result in a better move it stops evaluating any nodes coming off it, or if a possible move is too good but too far away it believes the player will likely prevent that move and so also removes these nodes.

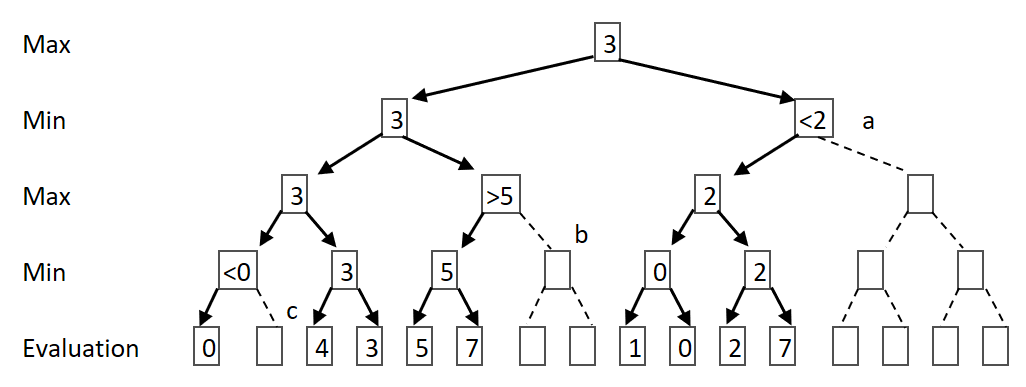
Above (figure 2.3) is the same decision tree used in the minimax explanation, with some changes in respect to how the Alpha-beta pruning would create the tree. Alpha-beta checks to whether searching the rest of a tree node would be beneficial or not.

Figure 2.3: Alpha-Beta Search Tree 1

For example node a, it has already obtained a value for the left-hand side of the tree (3) and at it would take the lowest possible value from its two children branches, one of which returns a 2. Now as node a’s highest possible value can only be 2 which is smaller than the 3 obtained from the other side of the tree. This deduction means that searching the other child branch from a would not result in the path that the Ai would take, and so it is scrapped. This potentially (depending on the order it is searched) results in a heavy reduction in the total amount of searchs.

The same applies to nodes b and c as well. In this example the total amount of evaluations is reduced from 16 to 9, and the amount of minimax checks is reduced by 4, however the total mounts of checks in the first place was small (16) whereas the number of checks needed for the AI will be much larger and so the potential reduction of evaluations is much greater.

By removing redundant nodes, it can heavily reduce the number of calculations and thus heavily reduce the time it takes. For chess, it reduces it from 6,553,600,000,000 possible combinations (worst case scenario) to 5,119,999 (best case scenario), however, that number is not always the case, it depends on the order of which the decision tree is searched. As the minimax algorithm is intended to run over multiple cores, and the depth would be quite high, being able to reduce the number of nodes would be beneficial, the algorithm will have to be made to balance accuracy and how long it would take for it to calculate the best move. If it takes 5 minutes to calculate the best move then the depth would have to be reduced to allow the user to play it in a timely fashion, so being able to reduce the time would also help keep the accuracy.

The research required to gain an understanding of the alpha-beta pruning technique, and how the minimax can be adapted for the project will be through the use of two pieces of research, *“Improved Alpha-Beta Pruning of Heuristic Search in Game-Playing Tree”* by Zhang Congpin and Cui Jinling (2009), and *“An Artificial Intelligence for the Board Game ‘Quarto!’ in Java”* by Jochen Mohrmann, Michael Neumann, and David Suendermann (2013).

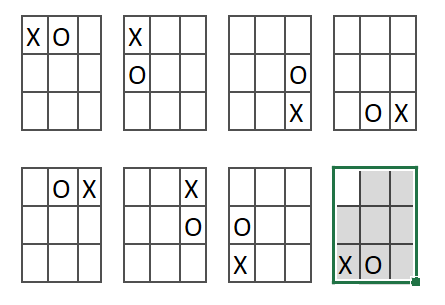
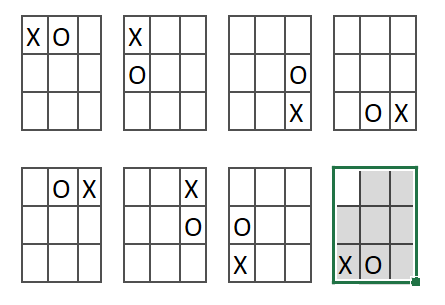
The first, *“Improved Alpha-Beta Pruning of Heuristic Search in Game-Playing Tree”* (Congpin and Jinling 2009), goes through applying a simple version of the technique to a minimax specifically for tic-tac-toe, improving it, and the effects it will have on node reduction and speed. This will be particularly useful for this project, it is designed on the same basic rules of as the projects, plus the section on improving it will allow for a significant reduction of nodes especially with a number of calculations the AI will have to make. For example, for a depth of 4, the minimax algorithm has 462 nodes, with the alpha-beta having 121 however once improved the alpha-beta has 50 nodes, with the depth being higher the reduction would be greater.

However, as the algorithm is designed for the basic tic-tac-toe it cannot be directly used for the project, as this version is several times more complex, and so it will have to be heavily adapted same as the minimax algorithm within this project. This is where the second paper comes in, *“An Artificial Intelligence for the Board Game ‘Quarto!’ in Java”* (Mohrmann, Neumann, and Suendermann 2013), it uses the same technique but for a different turn-based game. This will be useful to grasp an understanding of Alpha-Beta Pruning in different scenarios so it can, in turn, adapt it to this project specifically, as this project is a unique version of tic-tac-toe and so copying one already designed for the original game would not be sufficient, and so it would have to be modified.

**Rotational Symmetry**

The 8 grids to the left (figure 2.4) are all the same in the sense that it results in the same outcome, running the minimax algorithm would output the same results (refer to the table of calculations in the minimax explanation section) because each of those grids are either a symmetrical mirror or a rotation of the others.

Figure 2.4: Rotational Symmetry Example 1



Therefore, if the player’s move is either the side or corner (such as the picture) then the maximum amount of possible moves the AI can make is 5 rather than the 8 squares that are possible, in retrospect the number of possible starting moves is only 3 rather than the 9 squares that are available. This means that the number of branches can be reduced from the very top of the tree, thus heavily reducing the number of evaluations.

For the second picture (figure 2.5), with the X being in the middle, each corner search (blue) would result in the same decision tree as each other and each side search (red) would also result in the same decision tree as each other, thus reducing the number of search branches from 8 to 2, heavily reducing the number of evaluations.

Figure 2.5: Rotation Symmetry Diagram

A useful source for gaining an understanding of how to implement it in the project is “Symmetry Detection in General Game Playing” by Stephan Schiffel (2010). It explains the usefulness of symmetry detection, with an algorithm on how to implement it, plus the results of symmetry detection in a variety of games (such as connect 4 and tic tac toe).

These reductions mean that the Ai can afford to make the decision tree have a much lower depth in the same time frame as before, resulting in a better AI, making fewer mistakes.

**Deep Learning AI**

Both the minimax algorithm and Alpha-Beta pruning, are techniques used to create what is called a pre-programmed AI, the alternative to this type of AI is a deep learning AI. A deep learning AI is not programmed to be able to beat a person at a game but programmed to learn from its mistakes and successes. This method of AI is far more sophisticated and harder to program but allows for a much better AI. The main aspect to deep learning AIs is the neural network that is constructed to allow it to learn, much like how a human learns through neurons, and like a human the neural network is very large contains lots of layers of neurons before getting to the final conclusion.

A neural network essentially breaks down the input data, each neuron interpreting the data differently and sending forth calculations to another layer of neurons, and each neuron interprets the data differently, until it reaches the end and the output takes the information from all the neurons from the final layer and calculates whatever the AI is intended for (in this case the best move). The way the neurons interpret the data is through the use of formulas, weighing the input data and the weights for each input data is determined during the training phase. Once the neural network is created the AI is trained, it is given data and it attempts to predict the answer, its then compared to the real answer and then goes back through the network and changed the weight and keeps doing so until the values match, this is called backpropagation. Once the AIs Output matches the ‘correct answer’ it then receives a set of new data and it does the same before, each time refining it until its complete.

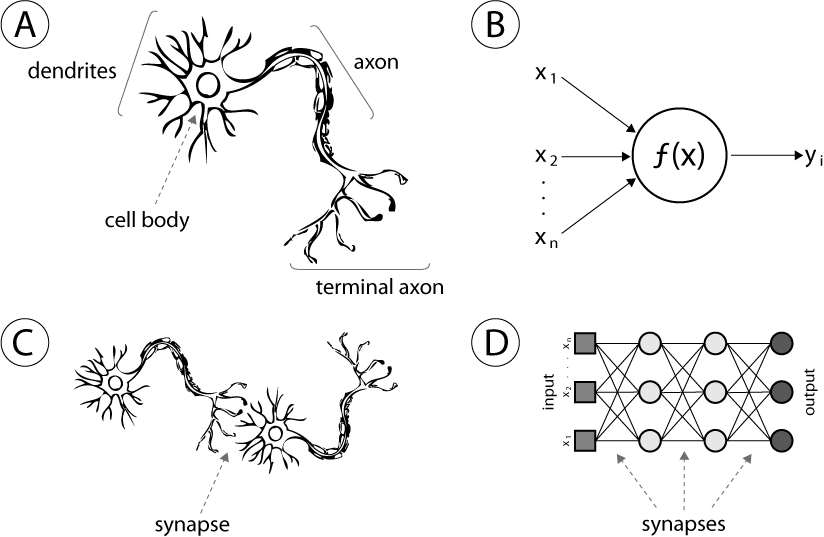


Figure 2.6: Neural Network (Intech 2013)

Above (figure 2.6) is a picture showing the neural network both within a brain and a computer. Section A is a neuron, which receives data, manipulates it and passes it to the next neuron, such as section B is a neuron within a computer working in the same fashion. Section C demonstrates how the neurons are connected within the brain, same as section D but for the computer.

An online book called *“Neural Networks and Deep learning”* by Michael Nielson (2015) covers all aspects of creating a deep learning AI from the theory of neural networks, to creating a backpropagation algorithm, to creating the finished AI which is this case can read and interpret handwritten digits with a 98% accuracy. This online book is particularly useful as it includes all the necessary information such as mathematical formulas but its broken down in a way that anyone new to deep learning can understand it, with useful diagrams and graphs, and with good explanations for all the concepts introduced with deep learning.

Another useful tool in deep learning AI is the Bellman Equation, which is used more in scenarios like mazes or computer games whereas the book by Michael Neilson is more for recognition (but useful nonetheless), created by Richard Bellman in the early 1950s, the algorithm essentially calculates a reward for each possible move the AI can make, similar to the minimax algorithm, however the reward changes depends on the previous times the AI made that move, which is why it differs from the minimax because it learns and this equation is integrated into the neural network. The Equation has been changed and revamped many times over the years allowing for such concepts as ‘Living Penalty’ which simply is a penalty for each move forcing the AI to complete its task in the least amount of moves possible.

The equation differs from project to project depending on its input data, its possible moves and its output data and so it will need to be tailored to this project specifically. The perfect starting point for learning how to create one would be a paper called *“The Theory of Dynamic Programming”* by Richard Bellman (1954), and it is a mathematically heavy paper with lots of formulas and is fairly complicated, but it successfully explains how to create such an equation.

However, like said before, the equation has been improved and revamped many times over the years and so another piece of literature would be required to create the equation, such as *“Markov Decision Processes: Concepts and Algorithms”* by Martijn van Otterlo (2009). This paper is more recent and includes concepts such as ‘Living Penalty’ and a thing called ‘Q-Learning’ which is one of the most important factors of a deep learning AI. Q-Learning works similarly to what is explained in the original Bellman equation but with a slight difference, while the original Bellman equations focus on the location the AI can go to (maze scenario), Q-learning focuses on the actions rather than location (so the journey rather than the destination), this is particularly useful in situations if there is any aspect of probability in play. The paper covers such things and more in a somewhat simple manner without removing any of the complicated mathematical equations required to make it.

The only downside to this paper is that it only includes aspects such as the improved Bellman Equation and so the online book by Michael Nielson (2015) would still be needed to help create the neural network and so both pieces are needed. The paper by Richard Bellman will be useful to grasp an understanding of the Bellman Equation and to allow the equation laid out in Martijn van Otterlo’s paper (2009) to be changed to meet the needs of the project.

# Design

**Minimax Algorithm**

Max (Board, Score)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached(Board)

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

Make move for AI

Score = Maximum (Score, Min(Board, Score))

Undo move for AI

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Score

Min (Board, Score)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached(Board)

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

Make move for Opponent

Score = Minimum (Score, Max(Board, Score))

Undo move for Opponent

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Score

**Minimax with Alpha-beta pruning**

AlphaBetaMax (Alpha, Beta, Board)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached(Board)

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

Make move for AI

Alpha = Maximum (Alpha, AlphaBetaMin(Board, Alpha, Beta))

Undo move for AI

If Alpha >= Beta

Return Beta

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Alpha

AlphaBetaMin (Alpha, Beta, Board)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached(Board)

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

Make move for Opponent

Beta = Minimum (Beta, AlphaBetaMax(Board, Alpha, Beta))

Undo move for Opponent

If Beta <= Alpha

Return Alpha

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Beta

**Minimax with AB and Rotational Symmetry**

AlphaBetaMax (Alpha, Beta, Board)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

If Symmetry(Board) != 1000

Return Score(Board)

else

Make move for AI

Alpha = Maximum (Alpha, AlphaBetaMin(Board, Alpha, Beta))

Undo move for AI

Add Board to List for Symmetry Check

If Alpha >= Beta

Return Beta

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Alpha

AlphaBetaMin (Alpha, Beta, Board)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached(Board)

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

If Symmetry (Board) != 1000

Return Score(Board)

Else

Make move for Opponent

Beta = Minimum (Beta, AlphaBetaMax(Board, Alpha, Beta))

Undo move for Opponent

Add Board to List for Symmetry Check

If Beta <= Alpha

Return Alpha

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Beta

Symmetry (Board)

Loop Count from 0 to 3

Rotate Board Right and Add to Temp List

Flip Columns on Board and Add to Temp List

If Count less than 3

Loop Back

Loop through Temp List (Loop 1)

Loop through Symmetry List (Loop 2)

If Current Temp Board is the same as current Symmetry Board

Return Score of Symmetry Board

Else

Loop back to Loop “

Loop Back to Loop 1

Return 1000 (meaning no Symmetry found)

**MM with AB, RS and Reduction Algorithm (for squared versions)**

AlphaBetaMax (Alpha, Beta, Board)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

If Reduce(Board) = true

Return zero

else

Make move for AI

Alpha = Maximum (Alpha, AlphaBetaMin(Board, Alpha, Beta))

Undo move for AI

If Alpha >= Beta

Return Beta

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Alpha

AlphaBetaMin (Alpha, Beta, Board)

If EitherPlayersWin

Return Score(Board)

If DepthLimitReached(Board)

Return Score(Board)

If BoardIsFull

Return Zero

Loop Count from 0 to 8 (or 80 for the 9x9 AI)

If Reduce(Board) = true

Return zero

Else

Make move for Opponent

Beta = Minimum (Beta, AlphaBetaMax(Board, Alpha, Beta))

Undo move for Opponent

If Beta <= Alpha

Return Alpha

If Count less than 8 (or 80 for the 9x9 AI)

Loop back

Return Beta

Reduction (Board)

If grid isn’t empty

Return false

If the next grid is empty

Return true

Else

Return false

**GUI**

Figure 3.1 is a Mockup design for the Android interface. The top Left is the Menu, top right is 3x3, bottom left is 9x9, and bottom right is for the two squared versions.

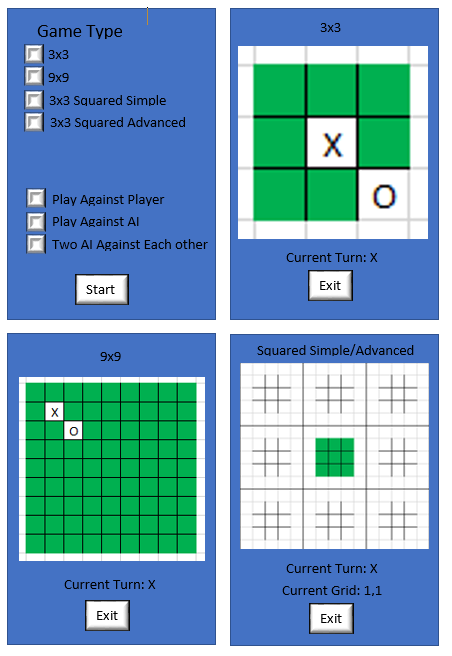


Figure 3.1: GUI Design

# Implementation

The first stage was to create the minimax algorithm for the 3x3 AI. This was to be the foundation of the AI for all versions. The nature of the algorithm is to calculate a score for every possible move by calculating every single possible game state, thus creating a very large decision tree, for the 3x3 it has a branch number of 255168. Creating the algorithm for the 3x3 was simple enough by creating a recursive function simulate every move for both the AI and the opponent one after the other until it reaches the end of a branch, and then creates the score, and continues creating the decision tree.

Now that the base algorithm was created the biggest obstacle was to reduce the size of the decision tree and therefore reduce the time it takes. Now whilst the time it takes to create the tree is low (125ms), for the larger boards the time taken is the most important aspect as for the larger boards the amount of possible game states so far too large to calculate within a human’s lifetime and so the depth of the decision tree must be limited, thus reducing the time it takes allows me to have a deeper depth for the tree.

The most effective way of reducing the size of the tree is the Alpha-beta pruning algorithm. Pruning away the branches of the tree that would not yield a useful move (refer to the research section on how it works). Creating this aspect was simple enough, is the score generated at the end of one of the branches deems that the rest of the connecting branches are redundant, it exits the recursive function thus heavily reducing the number of branches (89.33%) and more importantly the time it takes (86.4%).

The next aspect to add to reduce the number of branches is rotational symmetry, checking if the possible game state has already been checked in a variance of it, this is done by rotating the board 4 times (saving a copy of it each time) and flipping it every time it rotates it (and also saving a copy), once its generated the 8 symmetries then it checks those against the games states already evaluated. This reduced the number of branches heavily (99.95%) however because it has to do so many checks during this algorithm it actually takes longer than the Alpha-beta pruning.

When both the Alpha-beta pruning and rotational symmetry it can both reduce the number of branches and reducing the time it takes. However as seen below, the combined version takes 1ms longer than the alpha-beta alone and so to create the optimal AI the rotational symmetry was capped at a depth of 4, any evaluations at depth of 5 or lower didn’t run the rotational symmetry as each depth lower it goes the higher the cost whilst the lower the return.

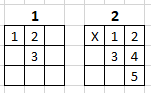
Refer to figure 4.1. The first column shows the possible moves the AI can make at the start, which is 3, and the symmetry algorithm is only run 9 times (for the 9 initial starting moves). Move down a layer (second column in the picture) and the player has 5 possible moves however the algorithm is run 24 times, for the 8 initial moves times 3 for the 3 moves from the previous layer. Therefore, the lower the depth of the tree the more times the symmetry algorithm is run whilst the effectiveness of the algorithm is reduced. Therefore, it's more efficient to cap the depth of the rotational symmetry to improve the time it takes (from 18ms to 16ms). Now whilst that time reduction is small, it will make a big impact on the larger boards.

Figure 4.1: Rotational Symmetry Demo 1

The table below shows the tree size and time taken if the Ai goes first, for the minimax, alpha-beta, rotational symmetry. The Most optimal depth for the rotational symmetry was 4.

|  |  |  |  |
| --- | --- | --- | --- |
| **3x3** | | | |
| **Algorithm** | **Total Branches** | **Time Taken** | **Reduction of ms** |
| Minimax | 255168 | 125ms | - |
| Alpha-beta | 27237 | 17ms | 86.4% |
| MM + Rotational Symmetry | 138 | 27ms | 74.2% |
| MM + AB + RS | 434 | 18ms | 85.6% |
| Optimal (RS of 4) | 1439 | 16ms | 87.2% |

Table 1: 3x3 AI Statistics

Now that the AI was completed for the 3x3 it was time to expand it to the 9x9, simply by expanding its decision tree width from 9 to 81 by increasing the size of the board and the size of all the loops within the AI. Doing so wasn’t difficult, the problem with is was the size of the decision tree, as the 9x9 has 81 starting moves it results in factorial 81 possible game states (5.797126e+120) which would take 3.16 e106 years (at 5.8 million checks a second).

The table below shows the tree size and time taken if the Ai goes first, for the minimax, alpha-beta, rotational symmetry and all combined with the depth capped at 4. As for the Rotational symmetry it had had to be capped at 2 to get an estimate as increasing it to 3 increases the time taken by an incredibly large number. The Most optimal depth for the rotational symmetry was 1 (top layer only), reducing the time taken enough to increase the depth from 4 to 5. similarities

|  |  |  |  |
| --- | --- | --- | --- |
| **9x9 (Max Depth of 4)** | | | |
| **Algorithm** | **Total Branches** | **Time Taken** | **Reduction of ms** |
| Minimax | 3074591520 | 507626ms | - |
| MM + Alpha-beta | 34148633 | 5532ms | 98.9% |
| MM + Rotation Symmetry (1) | 569368800 | 85627ms | 83.1% |
| MM + AB + RS (RS of 3) | 1482534 | 8679ms | 98.3% |
| Optimal (RS of 2) | 1625157 | 428ms | 99.9% |
| Optimal (Depth of 5) | 20615447 | 3507ms | - |

Table 2: 9x9 AI Statistics

The idea of multi-core processing didn’t yield any useful reductions. For testing purposes, it was implemented onto a small function that goes through all the squares checking if any are empty. This function has 9 checks (81 for the 9x9) and so the multi-core processing was implemented to run 3 checks at the same time. However, it resulted in taking longer (refer to figure 3).

|  |  |  |
| --- | --- | --- |
| **Version** | **Time (beforehand)** | **Time (Afterwards)** |
| 3x3 | 18ms | 60ms |
| 9x9 | 428ms | 13380ms |

Table 3: Multicore Before and After

Changing the checks to 5 improves the timings (3x3: 19ms, 9x9: 7135ms) but unfortunately it still results in it taking longer than before. The likely cause for this result is that creating the streams required to run it on multiple cores takes more time than running the checks one after another, if this is the case then using the multicore processing for such a small task is meaningless but might still give a good outcome if used for the minimax algorithm itself. However even if multicore processing did manage to reduce the amount of time, as the nature of Alpha-beta pruning and rotational symmetry rely on previous branches, if all of this was run parallelly off multi-cores then it would heavily reduce the effectiveness of both Alpha-beta pruning and rotational symmetry, and so might negate the bonuses from multicore processing. For now, the multi-core processing was scrapped until it could be investigated further.

The Ai for the Squared versions is a mixture between the 3x3 AI and the 9x9 AI, depending on the current state of the board (checking 9 moves in current square, or 81 moves if the player can go anywhere). A few adjustments we needed to allow for it to generate the score for a winning or losing move, but ultimately it was roughly the same, with the exception of rotational symmetry.

For rotational symmetry to work, instead of just checking the smaller grid of which the AI is in, it also checks the larger grid and then all remaining smaller grids (refer to figure 4.2). It runs the symmetry algorithm within the grid the AI is. If similarities are found, then it checks the values of the larger grid if they match the values of the game state it is comparing to. If similarities are found there, then it checks the values for all the other smaller grids.

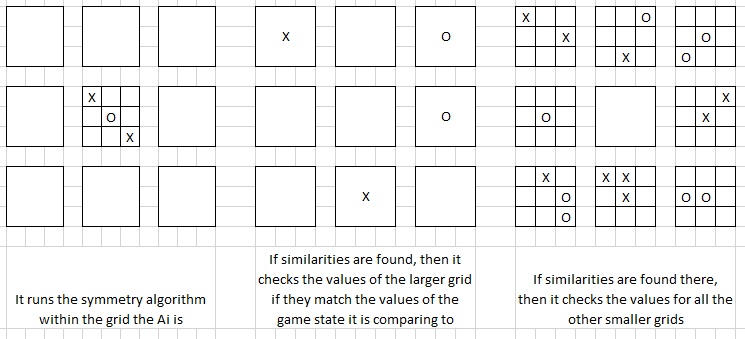


Figure 4.2: Rotational Symmetry Demo 2

However, running such an algorithm would be very taxing on the system and it wouldn’t yield much reduction and so creating the concept was scrapped as it wouldn’t have been worth it. So, in its stead I’ve replaced it with a different algorithm, a simple one but effective (90.65% branch reduction). As observed from the AI, as the depth is capped a lot of the time the score generated by the AI is 0 (no end in sight). If the AI is currently in an empty square (for example at the start) and the possible move would result in the other player being in an empty square then the algorithm will give it a score of 0 instead of running the minimax algorithm for that branch, as it would have generated a score of 0 regardless.

Whilst the Reduction algorithm and Alpha-beta pruning reduced the time taken by reducing the decision tree, lots of extra optimizations were added. Most were small, mainly by exiting any loop before the end of its cycle if possible, when this is done for any loops within the recursive function then it can reduce the time by quite a bit if the depth is deep enough.

One of the biggest difference is the evaluation during the minimax algorithm. During the algorithm each branch and layer must check if the latest move results in an endgame state, as this had to be done several thousand times, reducing the impact can be impactful, this was done by checking the current state of the grid at the time. If a player only has 2 squares within a grid then there is no point running that evaluation as there is no way for there to be an endgame state, refer to the table below for the time reduction. Now whilst it’s a small reduction, attempting to add as many techniques and tricks like this reduces the time taken and thus helps improve the AI.

|  |  |  |
| --- | --- | --- |
| **Squared (Depth of 8)** | **Simple** | **Advanced** |
| **Algorithm** | **Time Reduction** | **Time Reduction** |
| Minimax | 1063ms | 1792ms |
| MM + Alpha-beta | 27ms | 35ms |
| MM + Reduction | 116ms | 163ms |
| MM + AB + Reduction | 32ms | 33ms |
| Combined (Depth of 14) | 225ms | 352ms |

Table 4: Evaluation Improvement Stats

Combined with alpha-beta and extra reduction strategies, the branch size and time taken is reduced enough to allow for a high depth (14). Now that the reduction is enough the final thing was to improve the output of the AI, as it can’t see the end point (as the depth is capped) then the move decided by the AI is flawed. Now whilst reducing the decision tree size and time it takes is one of the most effective way of improving it, it can only be done so much and so another algorithm had to be created to improve the score generation processes if the depth is capped.

Originally it returned a score of 0, a simple addition was adapting that zero based on blocking. If at any time during that branch (game state) the AI blocks the players attempt to win then its score increases slightly, and vice versa if the player blocks the AI, thus making it that the AI actively blocks the player instead of just never sending the player back to that square.

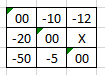
However, whilst this made sure the AI actively blocked and attempted to stop the player from blocking it, the AI still made questionable decisions. If sent to the same square twice it wouldn’t necessarily use those moves to win that square, and so a simple algorithm was devised. If after all its evaluations were completed and more than one square generated the highest value (more likely when all the squares return a score of zero) then it runs the 3x3 AI onto the moves within that grid that generated the highest score (refer to figure 3)

Figure 4.3: Logic Improvement Demo 1

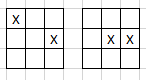
3 moves generated the highest score of zero (for example) and so the 3x3 Ai in run on those 3 squares, so instead of following the rules of the squared game (position of square in smaller grid sends the other player to a different smaller grid) it stays within that grid for all the checks so for those 3 moves it can figure out which one is best to choose. Refer to figure 4, as the top left square would have been the first evaluated it would choose that one, (the first picture) whereas the second picture would be a better move. This addition whilst improving the AI’s moves at the start of the game doesn’t impact the AI much as it only increases the number of branches by 434.

Figure 4.4: Logic Improvement Demo 2

The tables below shows the tree size and time taken if the Ai goes first, for the minimax, alpha-beta, rotational symmetry. A depth of 8 was chosen for the tests as the minimax took 13 seconds and increasing the depth would increase the time taken for the tests, the reduction in ms would remain roughly the same. For the simple version, the depth can be increased

To 14 with it making its move within a timely fashion, with the advanced version having a max depth of 13.

|  |  |  |  |
| --- | --- | --- | --- |
| **Squared Simple (Depth of 8)** | | | |
| **Algorithm** | **Total Branches** | **Time Taken** | **Reduction of ms** |
| Minimax | 278386272 | 13028ms | - |
| MM + Alpha-beta | 216222 | 56ms | 99.57% |
| MM + Reduction | 26023632 | 1178ms | 90.96% |
| MM + AB + Reduction | 22007 | 27ms | 99.79% |
| Combined (Depth of 14) | 25408721 | 1517ms | - |

Table 5: Squared Simple AI Stats

|  |  |  |  |
| --- | --- | --- | --- |
| **Squared Advanced (Depth of 8)** | | | |
| **Algorithm** | **Total Branches** | **Time Taken** | **Reduction of ms** |
| Minimax | 281067408 | 12699ms | - |
| MM + Alpha-beta | 327518 | 63ms | 99.50% |
| MM + Reduction | 26198592 | 1293ms | 89.81% |
| MM + AB + Reduction | 27745 | 30ms | 99.76% |
| Combined (Depth of 13) | 38746014 | 2061ms | - |

Table 6: Squared Advanced AI Stats

Once the AI was created for all 4 versions the next task was improving the code by combining the AIs together and formatting the entire project into an object-oriented design to reduce the amount of duplicated code (from 2200 lines including the test version to 1400 lines) and allow for changes easily without conflicts. Refer to Appendix C for the class diagram for the final product.

The intention was to then add an Android interface to the project so that it can be used on a phone, however it isn’t quite as simple as putting an interface over the existing project. The project would have been created from scratch in an android environment or developed the project into a Kotlin project, therefore the Android interface was put on hold.

# Testing and Results

Testing this AI isn’t the easiest of tasks as it must do several thousand calculations in such a short time, testing to see whether or not the results of those calculations are correct is difficult, and so there are two main ways to test the end result of the AIs.

The first is to make the AI play against itself, this helps to identify any problems within the AI both logically and systematically. However, as the AI is playing against itself this test only helps identify problems rather than the ability of the AI.

So, the second form of testing is required. After identifying problems, the AI is in a stable enough state for it to be tested against a Player (in this case the developer or volunteers). However, the problem with this form of testing is the human themselves in several ways. The first being the human’s decision making, if the tester is proficient in games like this then the AI will likely lose or if the tester is weak at games like this then the AI will likely win, thus resulting in this form being an inconsistent form of testing.

The second being the tester’s ability to learn, as the AI’s moves always follow the same pattern, if the tester loses against the AI there’s a higher chance they will win the next game as there a chance they have learnt from the game, thus again resulting in an inconsistent form of testing. Refer to table 7, for the developer the one win was after the two draws, after playing two games it’s easier to win against it after seeing how it plays and more importantly where the player went wrong. That said, the AI is massively predictable as some of the testers still didn’t win any games, watching the AI play only helps so far, as the AI can still see a dozen moves ahead.

|  |  |  |  |
| --- | --- | --- | --- |
| **Opponent** | **Wins** | **Loses** | **Draws** |
| Developer | 1 | 0 | 2 |
| Tester #1 | 0 | 0 | 3 |
| Tester #2 | 0 | 1 | 2 |
| Tester #3 | 1 | 0 | 2 |
| Tester #4 | 0 | 3 | 0 |

Table 7: Results from Play Tests

On top of that, for both tests, if the AI makes a bad move against a player, it’s hard to differentiate between a flaw in the system or the limitation of minimax.

# Project Management

## Project Schedule

* Created game system for 3x3
* Created game system for 9x9
* Created game system for 3x3 squared simple
* Created game system for 3x3 squared advanced
* Combined game systems
* Developed Minimax for 3x3
* Adapted Minimax for 9x9
* Developed Alpha-Beta Pruning for 3x3
* Developed Rotational Symmetry for 3x3
* Adapted Alpha-Beta Pruning to 9x9
* Adapted Rotational Symmetry to 9x9
* Adapted Minimax to 3x3 Squared Simple
* Adapted Alpha-Beta Pruning to 3x3 Squared Simple
* Adapted Minimax to 3x3 Squared Advanced
* Adapted Alpha-Beta Pruning to 3x3 Squared Advanced
* Developed Reduction Algorithm for 3x3 squared simple
* Adapted Reduction Algorithm to 3x3 Squared Advanced
* Improve score generation for both squared versions
* Convert code to suit an object-oriented design

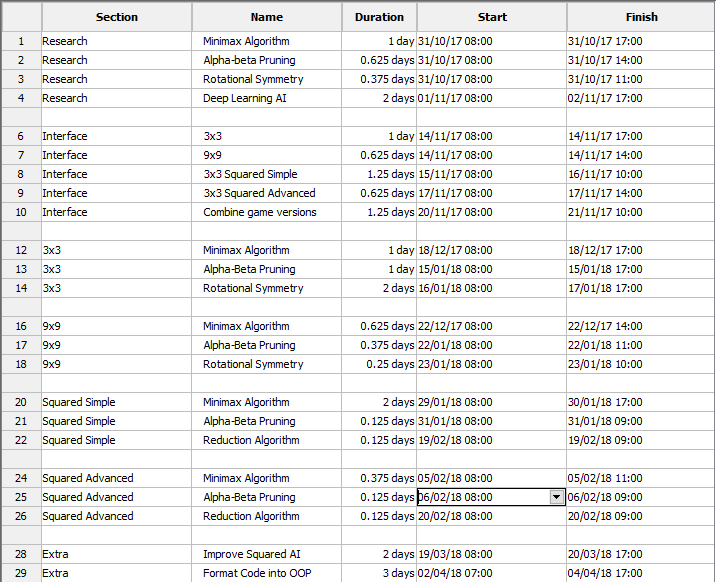


Figure 6.1: Time Chart

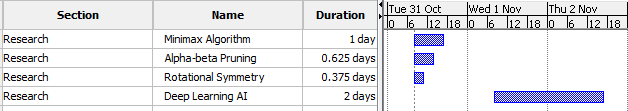


Figure 6.2: Gantt Chart Part 1

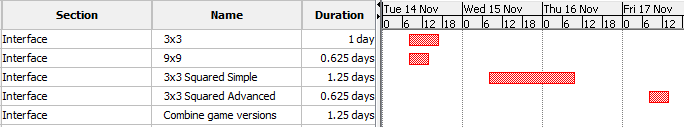


Figure 6.3: Gantt Chart Part 2



Figure 6.4: Gantt Chart Part 3

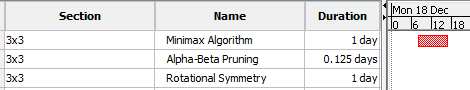


Figure 6.5: Gantt Chart Part 4

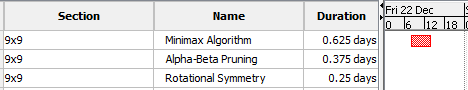


Figure 6.6: Gantt Chart Part 5

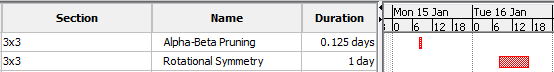


Figure 6.7: Gantt Chart Part 6

The alpha-beta pruning did not take as long as planned as the amount of code and adjustments required to make it work was small.

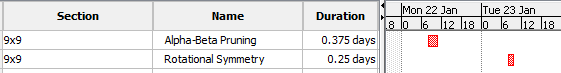


Figure 6.8: Gantt Chart Part 7

Adapting the AI for the 3x3 to 9x9 was simple (such as for the minimax) as all it included was increasing the number of squares it checks; therefore, it took less time as planned.

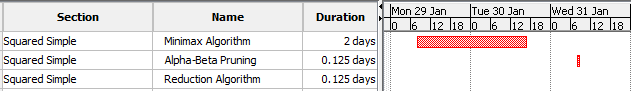


Figure 6.9: Gantt Chart Part 8



Figure 6.10: Gantt Chart Part 9

Adapting the minimax to Squared simple took longer than planned as I had to include extra checks and validations, in addition to extra ways to generate the score if the depth limit was met. However, adding the alpha-beta pruning was a simple task and so took less time than planned.

After evaluating the effectiveness rotational symmetry for the squared versions, I replaced it with the reduction algorithm which took less time to create then it would have to adapted the rotational symmetry.

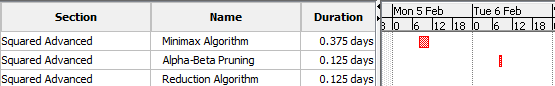


Figure 6.11: Gantt Chart Part 10



Figure 6.12: Gantt Chart Part 11

The extra code and adjustments needed to make the squared advanced work were less than anticipated and so took less time than anticipated.

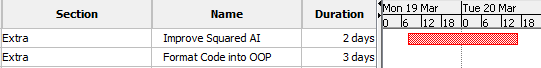


Figure 6.13: Gantt Chart Part 12



Figure 6.14: Gantt Chart Part 13

These sections weren’t originally planned and so it made the total time of the project longer.

The creation of the android interface was originally planned, even with it being a non-compulsory task, and was on the time plan proposed during the interim progress report. However, it was never implemented.

## Risk Management

By the nature of the project, being purely software, there were no risks.

## Quality Management

To help evaluate the move the AI makes, the system outputs a series of data related to the AI and the algorithm run: max depth, number of branches, the time is taken and the score generated for all moves considered. With this information, it was easy to check for errors, or gaps in logic for the AI plus it made sure that any improvements to the code actually reduce the decision tree size or time taken. Whilst this does help improve the quality of the end result, it was mostly useful during the development stage to speed up any problems encountered.

Another technique was the option to watch two AI play against each other (mentioned in the testing section). This allows for an unbiased viewing of the AI, and for a quick assessment of the AI as it makes moves quicker than a human. Combined with the data technique, it helps improve the quality and speeds up development.

# Critical Appraisal

Creation of the game systems for the 4 different versions was successfully. Each version runs without hinderance, and allows two players to play against each other on any of the 4 versions.

Creation of the base AI with the minimax algorithm was successful without any major issues (and the small issues were resolved). The AI generated all 255,168 game states which means that it worked correctly. Adaptation of the Minimax algorithm to incorporate Alpha-Beta Pruning was successful without any major issues. The Alpha-Beta pruning worked without problem, it heavily reduced the size of the decision tree whilst not impeding the decision making of the AI.

Development of Rotational Symmetry was a success, and worked as intended, also helped reduce the size of the decision tree for the 3x3. However, it was too taxing and resulted in taking longer than without it. To limit the cost of Rotational Symmetry

Adaptation of 3x3 AI to 9x9 was a success. Depth of AI had to be capped much higher up than anticipated (at 5). However, the rotational symmetry was even more taxing with the board being larger and it resulted in reducing the decision tree less than it did for the 3x3, the depth of the rotational symmetry algorithm was capped at 2.

Adapting the Minimax Algorithm and Alpha-Beta Pruning to 3x3 Squared Simple was a success. By the nature of this version of the game, the maximum possible game states is lower than the 9x9 and therefore the depth can be deeper then compared to the 9x9, with a Depth of 8 taking a fraction of the time that the depth of 5 for the 9x9 despite not having rotational symmetry and any other way of reducing the decision tree size. After realising that rotational symmetry was very taxing, and that for it to work for the Squared AIs, it would have to be three times as taxing (three times as many calculations) and that rotational symmetry would not reduce the size of the decision tree as effectively as it did for the 3x3, rotational symmetry was never developed for the Squared AI, and was instead replaced with a simple Algorithm that reduced the decision tree size for the first move, by checking if the current square is empty and any following squares are empty.

Adapting the Squared Simple AI to Squared Advanced was a success, with only a few chances were required. The AI works at an acceptable level, with it operating roughly an extra 1/3 calculations compared to its simple counterpart.

Creating an adaptation of the 3x3 AI to use for the Squared versions of the AI allowed for better decision making, with the taxation on the system being minimally small.

Adapting the code so into an OOP format was a success, and reduced the amount of duplicated code.

Whilst the creation of the AI was a success, and worked as intended, the AI is easier to beat then planned. This is because of the depth being capped on the algorithm, as the minimax works best by seeing all game states and analysing them, with the depth capped it can’t evaluate all game states and therefore cannot work fully.

The project took longer than anticipated, due to a delay in isolating problems. Due to the nature if the minimax Algorithm calling a recursive function, isolating potential logic bugs in it took a long time as it required the use of the debugger and sifting through each long of code one by one to identify the problem, however once identified resolving it took a short amount of time.

# Conclusions

Now whilst the creation of the AI was a success and the algorithms worked better than anticipated, the end result of the AI was not as anticipated. As the nature of the minimax algorithm is based on creating every possible game state and selecting the best move, having a limited depth and not being able to see the end game state had a bigger impact on the AI’s decision making and thus is not as difficult to play against. That said, the AI can still beat a human player or at least give it a considerable challenge, it just won’t be able to win 9/10 games as hoped.

This is where the deep learning AI comes into it (refer to research). As the learning AI works in a very different fashion, making its Decision based on past experiences (like a human being) rather than attempting to create the entire decision tree. This AI would be less taxing as fewer calculations would need to be done however the AI would be several times more complex and would require training. Now whilst the pre-programmed AI might not be able to win as much as hoped, it would be very effective as a tool to train the learning AI until its trained enough to give a human a challenge.

## Achievements

* Develop the game and game rules for player vs player
  + Created game system for 3x3
  + Created game system for 9x9
  + Created game system for 3x3 Squared Simple
  + Created game system for 3x3 Squared Advanced
  + Combined the games
* Develop an AI for 3x3 noughts and crosses
  + Developed Minimax
  + Developed Alpha-Beta Pruning
  + Developed Rotational Symmetry
* Develop an AI for 9x9 noughts and crosses winnable by 3 in a row
  + Adapted Minimax to 9x9
  + Adapted Alpha-Beta Pruning to 9x9
  + Adapted Rotational Symmetry to 9x9
* Develop an AI for 3x3 inside of 3x3, winnable by claiming one square
  + Adapted Minimax to 3x3 Squared Simple
  + Adapted Alpha-Beta Pruning to 3x3 Squared Simple
  + Developed Reduction Algorithm
* Develop an AI for 3x3 inside of 3x3, winnable by claiming squares in a noughts and crosses fashion
  + Adapted Minimax to 3x3 Squared Advanced
  + Adapted Alpha-Beta Pruning to 3x3 Squared Advanced
  + Adapted Reduction Algorithm to 3x3 Squared Advanced
* Convert code to suit an object-oriented design

## Recommendations for Future Work

Implementing multicore processing on the entire minimax algorithm, whilst it proved ineffective for the small function that it was implemented on, it is highly possible that applying it to the minimax algorithm itself could yield high improvements. Once that is done another potential improvement is augmenting it to run the calculations off the computers GPU instead of CPU, with the GPU having a lot of cores (thousands compared to half a dozen), as all the calculations are small but in great number, therefore, the GPU could potentially outperform the CPU, however this would heavily depend on the GPU itself therefore the results would vary from computer to computer.

During the research, the idea of using hash maps to save the combinations was encountered. Whilst for the 3x3 AI it would simply save the combinations, for the other versions it would have to take the saved combinations and expand them (as depth was limited). This feature has the potential to allow for a higher depth for the AI if it isn’t too taxing and the idea will definitely be explored in the future.

As the Android GUI was not developed for the project, this feature would be more potential future work, making sure that the project is lightweight so it can be run on a phone (when compared to a computer), however because of the difference in performance a phone has compared to a computer, the depth of the AI would have to be lower as it would take longer to do the same amount of calculations.

As the deep learning AI wasn’t attempted for the project, the learning AI would be definite future work, using the pre-programmed AI as a training tool. If the multi-core processing and GPU processing is successfully implemented and if it yields great results then it would greatly improve the pre-programmed AI, and as it would be used as a training tool for the learning AI, it would in turn also improve it.

# Student Reflections

My personal performance was adequate, with the creation of the AI being completed within the time frame. Whilst the final product doesn’t achieve the 9/10 wins as hoped, this is because of the limitation of Minimax and not with my abilities as a programmer, with that said the final AI created was above expectation.

I started this project before being taught Java so this project allowed me to use my already existing knowledge of programming to learn a new programming language and to implement that knowledge. Whilst I did run into some issues (Java related), with a short amount of time I was able to overcome them and learn from those problems. On top of that, the project ran in parallel with my Java-based Object Orientated Programming module which also allowed me to implement knowledge learnt from that module, such as formatting the project to suit an OOP design to reduce duplicated code.

During the initial creation of the minimax algorithm for the 3x3, the AI created a decision tree with over 700,000 branches despite noughts and crosses only having 255,168 unique game states. The problem was that the AI was calculating the game states more than once, and was, therefore, calculating a lot more than necessary. Whilst at the time it wasn’t a major issue as it could calculate roughly 750,000 branches in one second, it would have been a problem when trying to implement the algorithm for the other boards as the AI would not be able to generate all the possible game states even when it isn’t over calculating. The problem was resolved with the inclusion of Depth, originally the AI did not keep track of the depth as it doesn’t need it to generate the game states, however without it, it couldn’t keep track of which branches had already been evaluated.

I adapted the minimax algorithm to the 9x9 Board before creating the Alpha-beta pruning or rotational symmetry algorithm because adapting it was as simple as increasing the size of the loops to cover all 81 squares of the Board. However, without any form of reduction, the AI had to calculate so much that the AI’s depth had to be capped at 3, which in turn made for a very dull AI. Whilst there wasn’t a fault in the AI and the minimax had to be adapted to 9x9 at some point, I should have waited till Alpha-Beta and rotational Symmetry was done so that the 9x9 AI could have a max depth that allowed it to be functional.

That said, the entire inclusion of the 9x9 AI was, in hindsight, a waste of time. Initially the 9x9 AI was just a stepping stone from the 3x3 AI to the Squared Simple because the Squared board is 81 Squares, however by the nature of the 9x9 board, whoever goes first wins the game no matter how advanced the AI and so creating the AI was pointless, plus the Squared AIs were more like the 3x3 AI rather than the 9x9 Ai as the grids were split up into 3x3.

Whilst adapting the AI to work on the Squared Simple version was rather straightforward, the AI itself was lacking at first. For the 9x9, when the max depth was reached it returned a score of zero which was a problem, however by the nature of the Squared Simple game, returning a zero for when the depth was capped causes problems in the AI’s decision making. Therefore, several additions were to be added so that AI can still make a decent move without having to see the very end of the game.

One of the most crucial but simplest additions was giving and taking points for blocking. Simply, if the AI blocks the player’s attempts to win it gains some points, with the same being in reverse, if the player (when simulated in the minimax algorithm) blocks the AIs attempts the win then it loses points. Another feature was the inclusion of the 3x3 AI to decide which move is best when more than one move has the highest score (refer to implementation).

# 10 References

Bellman, R. (1954). *The Theory of Dynamic Programming.*

Borovska, P., & Lazarova, M. (2007). Efficiency of Parallel Minimax Algorithm for Game Tree Search. *International Conference on Computer Systems and Technologies - CompSysTech'07*.

Congpin, Z., & Jinling, C. (2009). Improved Alpha-Beta Pruning of Heuristic Seach in Game-Playing Tree. *2009 WRI World Congress on Computer Science and Information Engineering*, 672-674.

Ertel, W. (2011). *Introduction to Artificial Intelligence.* London: Springer-Verlag.

García Díez, S., Laforge, J., & Saerens, M. (2013). Rminimax: An Optiamally Randomized MINIMAX Algorithm. *IEEE Transactions on Cybernetics*, 385-393.

Intech. (2013). *Applications of Artificial Neural Networks in Chemical Problems*. Retrieved 01 06, 2017, from https://www.intechopen.com/books/artificial-neural-networks-architectures-and-applications/applications-of-artificial-neural-networks-in-chemical-problems

Mohrmann, J., Neumann, M., & Suendermann, D. (2013). An Artificial Intelligence for the Board Game 'Quarto!' in Java. *PPPJ '13 Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 141-146.

Moses, A. (2018). *305AAE Preparation Progress Report.*

Nielson, M. (2015). *Neural Networks and Deep Learning.* Determination Press. Retrieved from neuralnetworksanddeeplearning: neuralnetworksanddeeplearning.com

Schiffel, S. (2010). Symmetry Detection in General Game Playing. *AAAI Conference on Artificial Intelligence*.

van Otterlo, M. (2009). Markov Decision Processes: Concepts and Algorithms. *SIKS course on ”Learning and Reasoning”*.

Appendix A – Interim Progress Report

The following is my Interim Preparation Progress Report submitted on 8th January 2018 (Moses, 2018)

**Individual Assignment Task**

**Developing an AI for advanced noughts and crosses - a turn-based board game**

Alexander Moses, 6277932

Supervisor: Chris Bass

**Introduction**

The project is to create an AI for an advanced version of noughts and crosses (3x3 grid within each square of a 3x3 grid, with the game to be won in the same fashion of normal noughts and crosses). The final product will be playable on an Android device, with multiple versions of noughts and crosses (refer to aims), with the option to play each version versus a player or AI.

The purpose of the project is for the designer to use this AI to demonstrate the creators programming ability and understanding of AI to potential employers. The intended purpose can be extended to allowing users of any age group to challenge themselves against an AI in a strategic turn-based game.

The motivation for the project is for the designer to gain an understanding of AI, and to use this project to help obtain a career in the field of AI.

**Aims and Objectives**

The requirements for the final project is that the AI is winnable in a minimum of 9/10 games on an android platform. The project will be broken down into stages to help with its completion, completing each one in succession until the final project is complete. The Primary stages are as below:

* Develop the game and game rules for player vs player
* Develop an AI for 3x3 noughts and crosses
* Develop an AI for 9x9 noughts and crosses winnable by 3 in a row
* Develop an AI for 3x3 inside of 3x3, winnable by claiming one square
* Develop an AI for 3x3 inside of 3x3, winnable by claiming squares in a noughts and crosses fashion

With the secondary stages, as below:

* Developing it as an Android phone application

With a bonus but non-compulsory stage:

* Create a Deep Learning AI for the 3x3 Squared Advanced

Any variations to the original project proposal described and explained.

The bonus aim of the Deep learning AI was not originally in the project brief, with the addition of the option to play all versions of noughts and crosses from the Aims, which were originally only going to be milestones between the start and finish.

Expected practical attainments and output from completing your project and the impact they could have e.g. economic, social and environmental issues.

The practical attainments and output from the finished product is the use of the AI to be used to demonstrate the designer’s ability and understanding of AI to potential employers, with the only impact being the potential career for the designer. There are no economic, social or environmental issues caused by this project.

**Literature Review/Background Research**

One source was used for three sections (minimax, alpha-beta and neural networks) and that was a book called “*Introduction to Artificial Intelligence”* by Wolfgang Ertel (2011). It covers all areas of AI including the three included in this project, it covers the concepts with straightforward explanations, with diagrams, explanations and algorithms in pseudo code. This book (Ertel 2011) was particularly useful for gaining an understanding of all areas of AI useful for starting the research of this project, as a basic knowledge and understanding is required to identify the usefulness of a source.

**Minimax Algorithm**

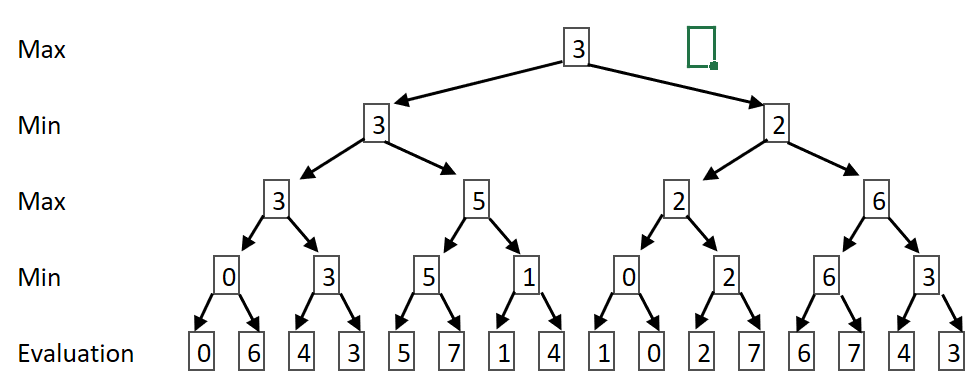
The minimax algorithm is a method for minimizing the possible worst-case scenario and is used in decision theory, game theory, statistics and philosophy. The algorithm in this case if for game theory, to be used in a pre-programmed AI for it to calculate the best move by attempting to minimize the potential ‘score’ that the opposing player can get against the AI. In simple terms, it generates a score for every possible move that the AI can make with every possible move the player can make proceeding the AI makes theirs until every combination of moves has a calculated score, for the AI’s turn it attempts to obtain the maximum score and for the player’s turn it attempts to obtain the minimum score (refer to figure 1).

Figure 1: Minimax Decision Search Tree 1

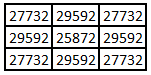
Once implemented the AI evaluates a total amount of 255168 times (the same amount as the number of unique tic tac toe games), the average number of evaluations is 28352 if the player goes first. The table to the left (figure 2) is the number of evaluations depending on where the player went first (i.e. top left square is the number of evaluations if the player’s first move is in the top left square)

Figure 2: Minimax Search Count 1

The Research required to gain an understanding of this algorithm and how to create one to suit the needs of the project will be two published papers. One being *“Rminimax: An Optimally Randomized Minimax Algorithm”* by Silvia García Díez, Jérôme Laforge, and Marco Saerens (2013), and the other being *“Efficiency of Parallel Minimax Algorithm for Game Tree Search”* by Plamenka Borovska, and Milena Lazarova (2007).

*“An Optimally Randomized Minimax Algorithm”* (García Díez, Laforge, and Saerens 2013) is a paper highlighting the mathematical side of the concept of the algorithm rather than an already created generic algorithm, and so it contains the possible ways that the algorithm can be constructed with full explanations of the mathematical formulas needed to create the algorithm. This is particularly useful as instead of a paper explaining how to create the algorithm, it explains the math behind it so the AI can be personally tailored to the project in comparison to a generic minimax algorithm, allowing for better performance and better results.

The only downside to this is that it is only the mathematical side it does not highlight at all the impact that the algorithm will have on a computer, such as performance or efficiency. That is where the other paper comes in, “*Efficiency of Parallel Minimax Algorithm for Game Tree Search”* (Borovska, and Lazarova 2007) covers the computational side of it. The algorithm, if complex enough, would need to make a very large amount of calculations and that can take time, so to reduce the time constraint, the ‘tree’ that the algorithm is searching can be broken down into sections and said sections are run on different cores of the processor, allowing them to be done in parallel. This reduction of time is limited to the number of cores the device (computer or phone or tablet) has, and the speed of the cores. This paper highlights the ways that the algorithm can be computed in parallel, the advantages and disadvantages to each way, and the efficiency of parallel computing the algorithm. However, this paper alone will not help construct the algorithm, only allow the search to be sped up by utilising the multi-core processor that modern-day devices have, and so both papers together will be needed to make the most optimal algorithm.

**Alpha-Beta Pruning**

The minimax algorithm works by creating a tree of nodes for all combinations of moves, with an increasing level of depth. Alpha-beta pruning is an algorithm to decrease the number of nodes in the tree created in the minimax algorithm. It does this when it encounters a node that it deems that it will not result in a better move it stops evaluating any nodes coming off it, or if a possible move is too good but too far away it believes the player will likely prevent that move and so also removes these nodes.

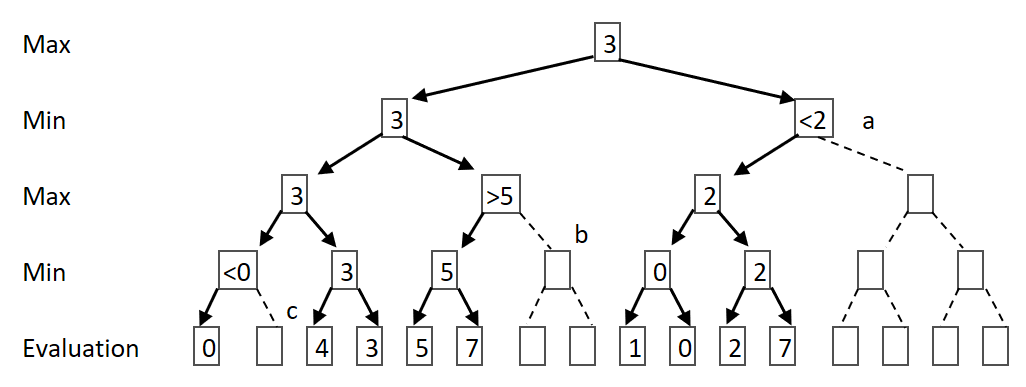
Above (figure 3) is the same decision tree used in the minimax explanation, with some changes in respect to how the Alpha-beta pruning would create the tree. Alpha-beta checks to whether searching the rest of a tree node would be beneficial or not.

Figure 3: Alpha-Beta Search Tree 1

For example node a, it has already obtained a value for the left-hand side of the tree (3) and at it would take the lowest possible value from its two children branches, one of which returns a 2. Now as node a’s highest possible value can only be 2 which is smaller than the 3 obtained from the other side of the tree. This deduction means that searching the other child branch from a would not result in the path that the Ai would take, and so it is scrapped. This potentially (depending on the order it is searched) results in a heavy reduction in the total amount of searchs.

The same applies to nodes b and c as well. In this example the total amount of evaluations is reduced from 16 to 9, and the amount of minimax checks is reduced by 4, however the total mounts of checks in the first place was small (16) whereas the number of checks needed for the AI will be much larger and so the potential reduction of evaluations is much greater.

By removing redundant nodes, it can heavily reduce the number of calculations and thus heavily reduce the time it takes. For chess, it reduces it from 6,553,600,000,000 possible combinations (worst case scenario) to 5,119,999 (best case scenario), however, that number is not always the case, it depends on the order of which the decision tree is searched. As the minimax algorithm is intended to run over multiple cores, and the depth would be quite high, being able to reduce the number of nodes would be beneficial, the algorithm will have to be made to balance accuracy and how long it would take for it to calculate the best move. If it takes 5 minutes to calculate the best move then the depth would have to be reduced to allow the user to play it in a timely fashion, so being able to reduce the time would also help keep the accuracy.

The research required to gain an understanding of the alpha-beta pruning technique, and how the minimax can be adapted for the project will be through the use of two pieces of research, *“Improved Alpha-Beta Pruning of Heuristic Search in Game-Playing Tree”* by Zhang Congpin and Cui Jinling (2009), and *“An Artificial Intelligence for the Board Game ‘Quarto!’ in Java”* by Jochen Mohrmann, Michael Neumann, and David Suendermann (2013).

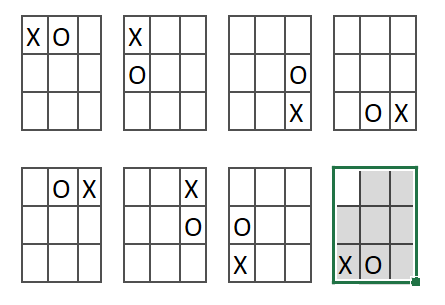
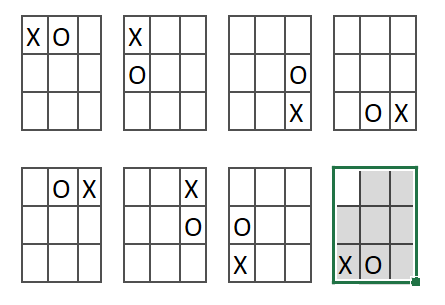
The first, *“Improved Alpha-Beta Pruning of Heuristic Search in Game-Playing Tree”* (Congpin and Jinling 2009), goes through applying a simple version of the technique to a minimax specifically for tic-tac-toe, improving it, and the effects it will have on node reduction and speed. This will be particularly useful for this project, it is designed on the same basic rules of as the projects, plus the section on improving it will allow for a significant reduction of nodes especially with a number of calculations the AI will have to make. For example, for a depth of 4, the minimax algorithm has 462 nodes, with the alpha-beta having 121 however once improved the alpha-beta has 50 nodes, with the depth being higher the reduction would be greater.

However, as the algorithm is designed for the basic tic-tac-toe it cannot be directly used for the project, as this version is several times more complex, and so it will have to be heavily adapted same as the minimax algorithm within this project. This is where the second paper comes in, *“An Artificial Intelligence for the Board Game ‘Quarto!’ in Java”* (Mohrmann, Neumann, and Suendermann 2013), it uses the same technique but for a different turn-based game. This will be useful to grasp an understanding of Alpha-Beta Pruning in different scenarios so it can, in turn, adapt it to this project specifically, as this project is a unique version of tic-tac-toe and so copying one already designed for the original game would not be sufficient, and so it would have to be modified.

**Rotational Symmetry**

The 8 grids to the left (figure 4) are all the same in the sense that it results in the same outcome, running the minimax algorithm would output the same results (refer to the table of calculations in the minimax explanation section) because each of those grids are either a symmetrical mirror or a rotation of the others.

Figure 4: Rotational Symmetry Example 1



Therefore, if the player’s move is either the side or corner (such as the picture) then the maximum amount of possible moves the AI can make is 5 rather than the 8 squares that are possible, in retrospect the number of possible starting moves is only 3 rather than the 9 squares that are available. This means that the number of branches can be reduced from the very top of the tree, thus heavily reducing the number of evaluations.

For the second picture (figure 5), with the X being in the middle, each corner search (blue) would result in the same decision tree as each other and each side search (red) would also result in the same decision tree as each other, thus reducing the number of search branches from 8 to 2, heavily reducing the number of evaluations.

Figure 5: Rotation Symmetry Diagram

A useful source for gaining an understanding of how to implement it in the project is “Symmetry Detection in General Game Playing” by Stephan Schiffel (2010). It explains the usefulness of symmetry detection, with an algorithm on how to implement it, plus the results of symmetry detection in a variety of games (such as connect 4 and tic tac toe).

These reductions mean that the Ai can afford to make the decision tree have a much lower depth in the same time frame as before, resulting in a better AI, making fewer mistakes.

**Deep Learning AI**

Both the minimax algorithm and Alpha-Beta pruning, are techniques used to create what is called a pre-programmed AI, the alternative to this type of AI is a deep learning AI. A deep learning AI is not programmed to be able to beat a person at a game but programmed to learn from its mistakes and successes. This method of AI is far more sophisticated and harder to program but allows for a much better AI. The main aspect to deep learning AIs is the neural network that is constructed to allow it to learn, much like how a human learns through neurons, and like a human the neural network is very large contains lots of layers of neurons before getting to the final conclusion.

A neural network essentially breaks down the input data, each neuron interpreting the data differently and sending forth calculations to another layer of neurons, and each neuron interprets the data differently, until it reaches the end and the output takes the information from all the neurons from the final layer and calculates whatever the AI is intended for (in this case the best move). The way the neurons interpret the data is through the use of formulas, weighing the input data and the weights for each input data is determined during the training phase. Once the neural network is created the AI is trained, it is given data and it attempts to predict the answer, its then compared to the real answer and then goes back through the network and changed the weight and keeps doing so until the values match, this is called backpropagation. Once the AIs Output matches the ‘correct answer’ it then receives a set of new data and it does the same before, each time refining it until its complete.

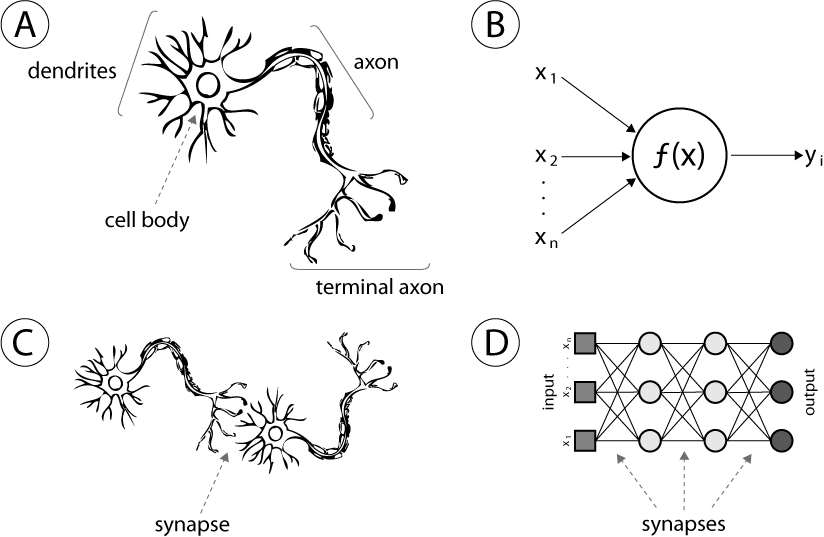


Figure 6: Neural Network (Intech 2013)

Above (figure 6) is a picture showing the neural network both within a brain and a computer. Section A is a neuron, which receives data, manipulates it and passes it to the next neuron, such as section B is a neuron within a computer working in the same fashion. Section C demonstrates how the neurons are connected within the brain, same as section D but for the computer.

An online book called *“Neural Networks and Deep learning”* by Michael Nielson (2015) covers all aspects of creating a deep learning AI from the theory of neural networks, to creating a backpropagation algorithm, to creating the finished AI which is this case can read and interpret handwritten digits with a 98% accuracy. This online book is particularly useful as it includes all the necessary information such as mathematical formulas but its broken down in a way that anyone new to deep learning can understand it, with useful diagrams and graphs, and with good explanations for all the concepts introduced with deep learning.

Another useful tool in deep learning AI is the Bellman Equation, which is used more in scenarios like mazes or computer games whereas the book by Michael Neilson is more for recognition (but useful nonetheless), created by Richard Bellman in the early 1950s, the algorithm essentially calculates a reward for each possible move the AI can make, similar to the minimax algorithm, however the reward changes depends on the previous times the AI made that move, which is why it differs from the minimax because it learns and this equation is integrated into the neural network. The Equation has been changed and revamped many times over the years allowing for such concepts as ‘Living Penalty’ which simply is a penalty for each move forcing the AI to complete its task in the least amount of moves possible.

The equation differs from project to project depending on its input data, its possible moves and its output data and so it will need to be tailored to this project specifically. The perfect starting point for learning how to create one would be a paper called *“The Theory of Dynamic Programming”* by Richard Bellman (1954), and it is a mathematically heavy paper with lots of formulas and is fairly complicated, but it successfully explains how to create such an equation.

However, like said before, the equation has been improved and revamped many times over the years and so another piece of literature would be required to create the equation, such as *“Markov Decision Processes: Concepts and Algorithms”* by Martijn van Otterlo (2009). This paper is more recent and includes concepts such as ‘Living Penalty’ and a thing called ‘Q-Learning’ which is one of the most important factors of a deep learning AI. Q-Learning works similarly to what is explained in the original Bellman equation but with a slight difference, while the original Bellman equations focus on the location the AI can go to (maze scenario), Q-learning focuses on the actions rather than location (so the journey rather than the destination), this is particularly useful in situations if there is any aspect of probability in play. The paper covers such things and more in a somewhat simple manner without removing any of the complicated mathematical equations required to make it.

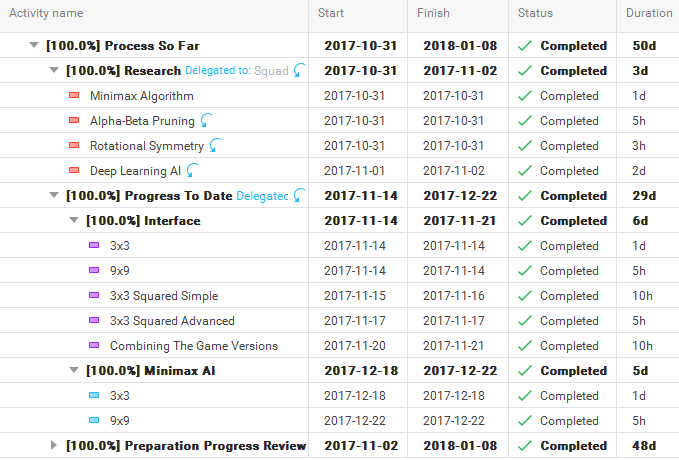
The only downside to this paper is that it only includes aspects such as the improved Bellman Equation and so the online book by Michael Nielson (2015) would still be needed to help create the neural network and so both pieces are needed. The paper by Richard Bellman will be useful to grasp an understanding of the Bellman Equation and to allow the equation laid out in Martijn van Otterlo’s paper (2009) to be changed to meet the needs of the project.

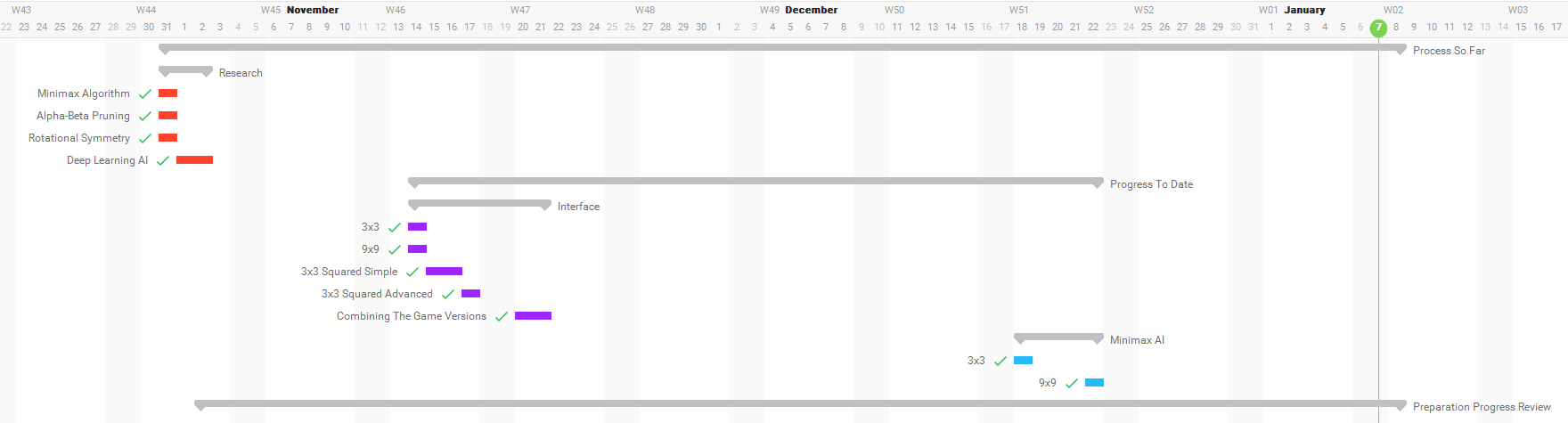
**Conclusion**

In conclusion, there are two very different types of AI that can be used for this game, one being the pre-programmed AI, the other being the deep learning AI. The deep learning AI would create a much better AI, making it more difficult to beat but would be very difficult to create and a lot of theory required to create it, plus it would require an initial training period before it can be used. Which leaves the pre-programmed AI, it can perform as well as the deep learning AI but much less efficiently, for the 3x3 squared advanced version the pre-programmed would have to calculate a tremendous number of potential moves to figure out the correct move which can take too long, and so the depth of the decision tree must be limited thus making it easier to beat, but in the end, it is also easier and quicker to create.

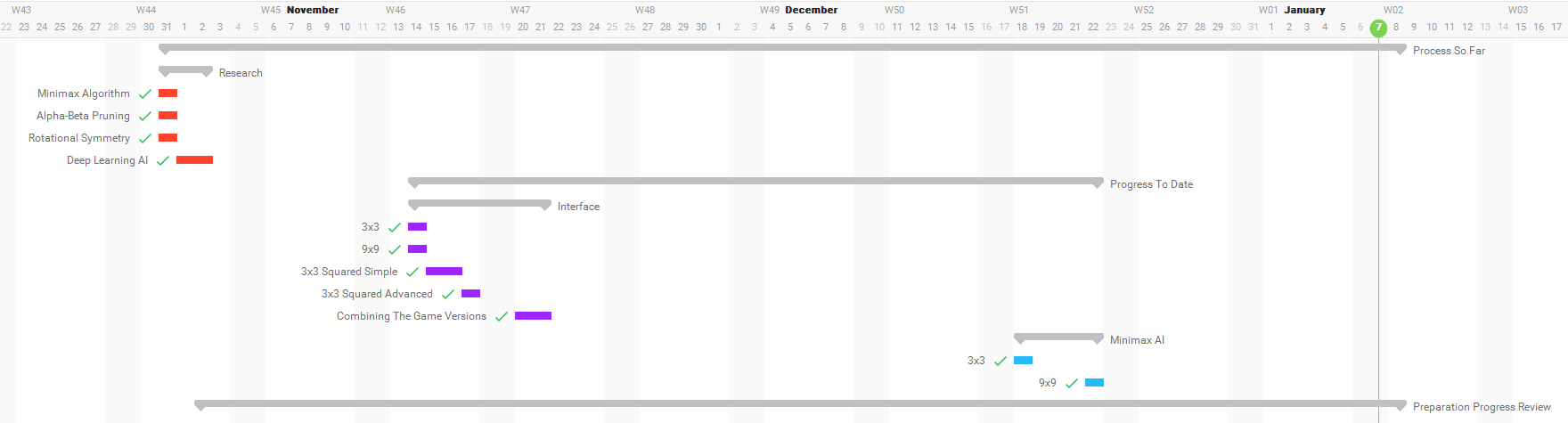
**Project Management**

Figure 7: Time Plan





**Figure 8: Gantt Chart Part 1**



**Figure 9: Gantt Chant Part 2**

**Progress to Date**

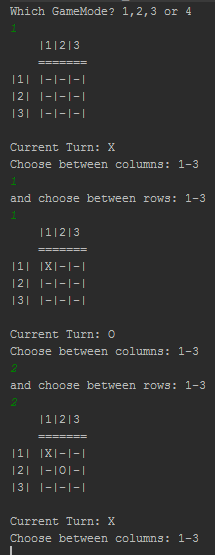
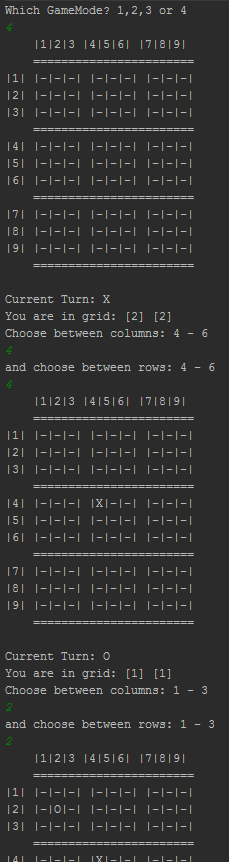
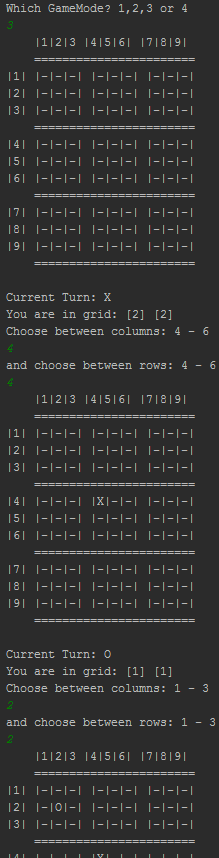
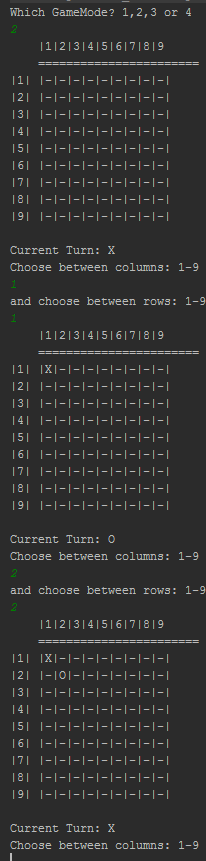
A player vs player version of the game has been created for all version of noughts and crosses (3x3, 9x9, 3x3 squared basic, and 3x3 squared advanced) with a console based interface. This will be used as a testing platform for the AI, plus also the availability to collect data for player vs player games or player vs AI games, with the ability to view the game and the moves that were made, so that the AI can be assessed at whether it is making the best possible moves.

Figure 10: 3x3 Interface

To the left is a picture of the 3x3 interface (figure 10), and below are the pictures demonstrating the interface for the other three version. The user chooses out of the 4 possible game modes. The first picture is the selection of which game mode. The first picture is the 3x3 (figure 10), the second picture is the 9x9 (figure 11), the third picture is the 3x3 squared simple (figure 12), and the fourth is 3x3 squared advanced (figure 13).



**Figure 11: 9x9 Interface Figure 13: 3x3 Squared Advanced Interface**

**Figure 12: 3x3 Squared Simple Interface**

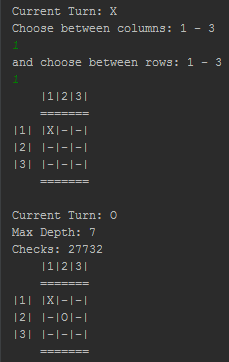
The simplest version of the AI for the 3x3 has been created, featuring only the minimax algorithm, the 9x9 Ai with minimax is also done but without the optimizations of alpha-beta and rotational symmetry it would take far too long to perform the brute force method of minimax, performing roughly 769,230 evaluations a second which for the 3x3 is not an issue (with a maximum of 255,168) but when moving onto larger boards it would take too long (roughly 1.64e107 years for the 9x9) and so optimisations are needed, and the 3x3 will be used as a testing platform. To the right (figure 14) is a picture of the 3x3 minimax only AI working.

Figure 14: 3x3 Minimax

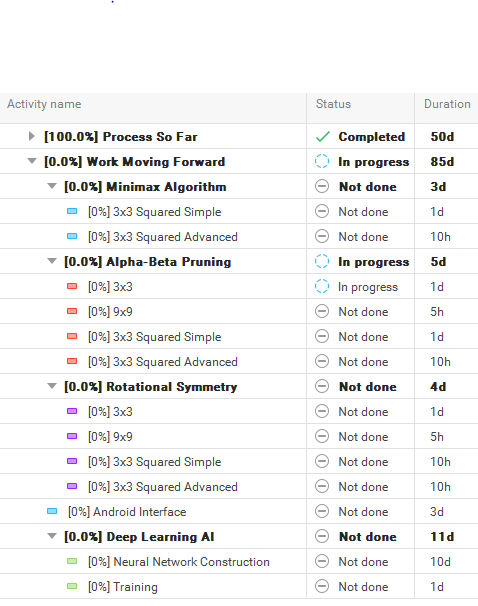
**Planned Work Ahead**

The next step is to create the alpha-beta pruning algorithm for the 3x3 and 9x9, whilst it is possible and rather straightforward to adapt the minimax for the 3s3 squared versions, without the alpha-beta or rotational symmetry the depth of the minimax would have to be low (such as 3) and that’s not sufficient for a working AI for the 3x3 squared, with a depth that low a lot of the moves the AI would make would be random and so alpha-beta is a priority.

Once that is done then adapt the minimax and alpha-beta for the 3x3 squared simple and test whether that is sufficient, if not then the next move is to create the rotational symmetry algorithm for the 3x3 and repeat what was done for the alpha-beta until the 3x3 squared simple has a sufficient depth for it to be altered for the 3x3 squared advanced.

Once completed, the next step is to create the Android interface for it, so that it can be playable on an Android device in a user-friendly way, whereas the console interface is not the most user-friendly.

If all of this is done in a timely fashion then the next step is to create the neural network to be used to create a deep learning AI for the game. Once all of that is done, the last step would be to train the AI and implement it into the pre-programmed AI, with the option for the user to pick between the two because the deep learning AI has the capability to be more advanced making it harder to beat.



**Figure 15: Future Time Plan**

**Summary and Conclusion**

With the minimax already done for the 3x3 and 9x9, adding the alpha-beta pruning would be rather straightforward, and so I believe to be ahead of schedule as I’ve almost done half of the initially proposed design. As for the non-compulsory bonus objective of the deep learning AI, it is too early to tell whether I’d get it completed.

**References**

Bellman, R. (1954). *The Theory of Dynamic Programming.*

Borovska, P., & Lazarova, M. (2007). Efficiency of Parallel Minimax Algorithm for Game Tree Search. *International Conference on Computer Systems and Technologies - CompSysTech'07*.

Congpin, Z., & Jinling, C. (2009). Improved Alpha-Beta Pruning of Heuristic Seach in Game-Playing Tree. *2009 WRI World Congress on Computer Science and Information Engineering*, 672-674.

Ertel, W. (2011). *Introduction to Artificial Intelligence.* London: Springer-Verlag.

García Díez, S., Laforge, J., & Saerens, M. (2013). Rminimax: An Optiamally Randomized MINIMAX Algorithm. *IEEE Transactions on Cybernetics*, 385-393.

Intech. (2013). *Applications of Artificial Neural Networks in Chemical Problems*. Retrieved 01 06, 2017, from https://www.intechopen.com/books/artificial-neural-networks-architectures-and-applications/applications-of-artificial-neural-networks-in-chemical-problems

Mohrmann, J., Neumann, M., & Suendermann, D. (2013). An Artificial Intelligence for the Board Game 'Quarto!' in Java. *PPPJ '13 Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 141-146.

Moses, A. (2018). *305AAE Preparation Progress Report.*

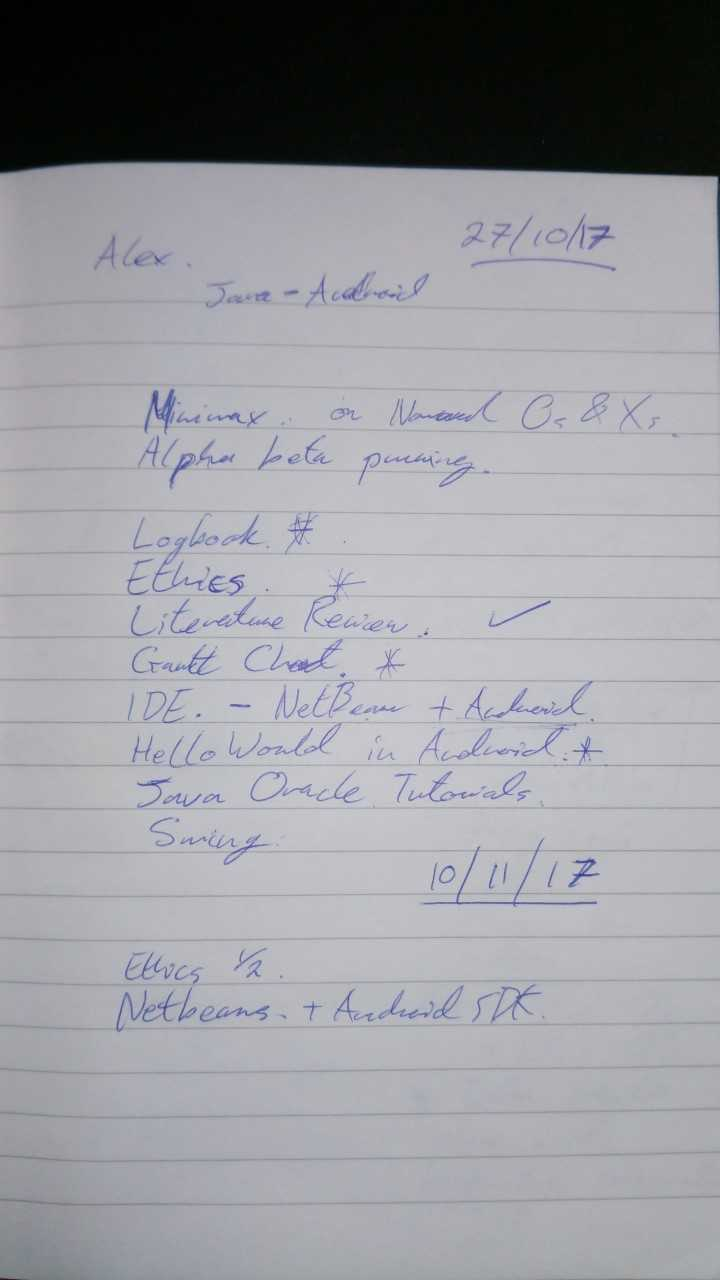
Nielson, M. (2015). *Neural Networks and Deep Learning.* Determination Press. Retrieved from neuralnetworksanddeeplearning: neuralnetworksanddeeplearning.com

Schiffel, S. (2010). Symmetry Detection in General Game Playing. *AAAI Conference on Artificial Intelligence*.

van Otterlo, M. (2009). Markov Decision Processes: Concepts and Algorithms. *SIKS course on ”Learning and Reasoning”*.

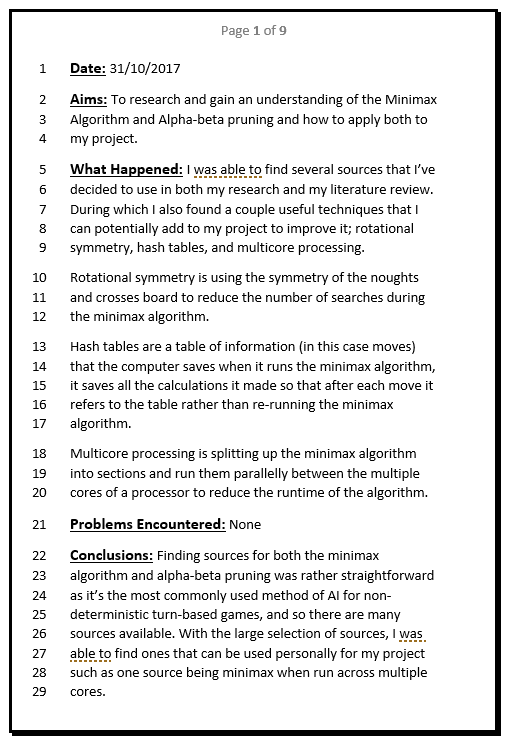
**Appendices**

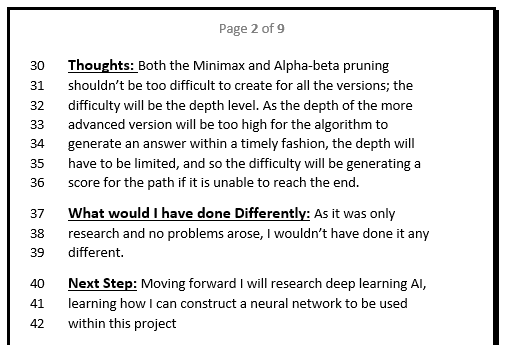
Notes from the first meeting with project supervisor

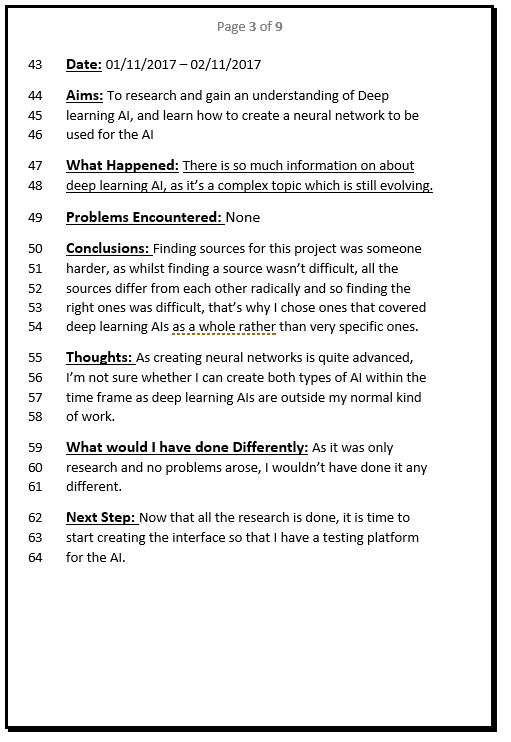


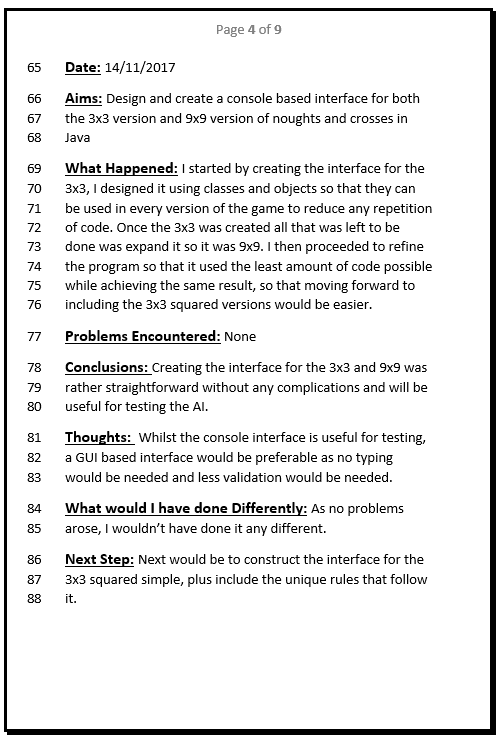
**Figure 16: Notes from Project Meeting**

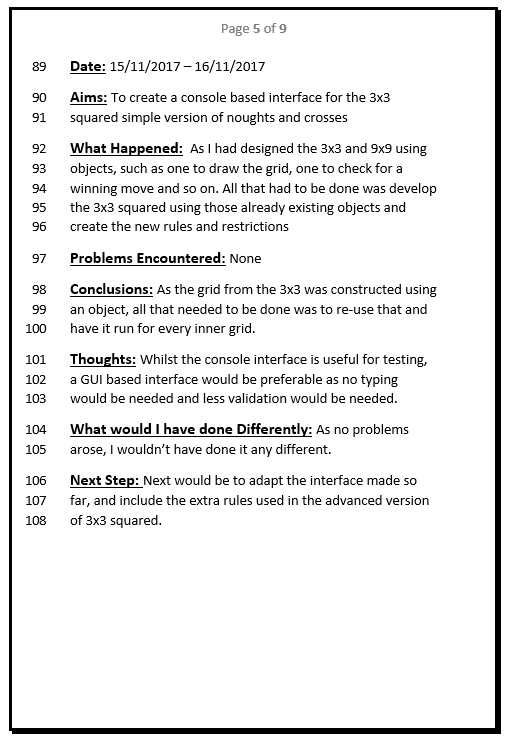
**Logbook**

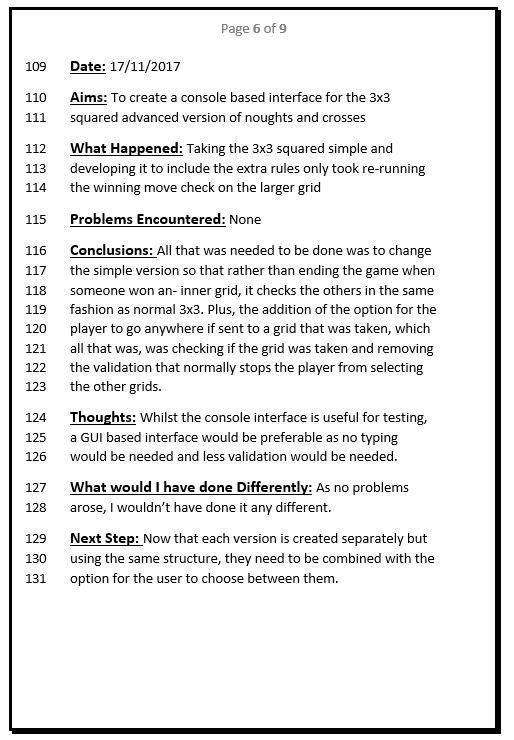


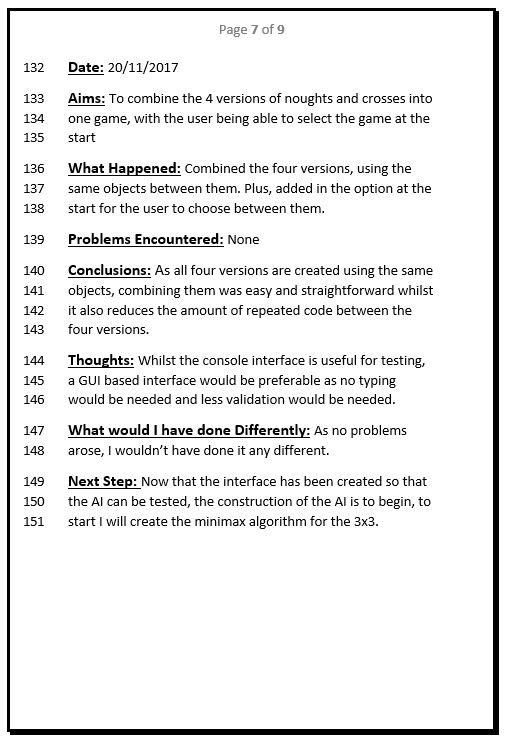


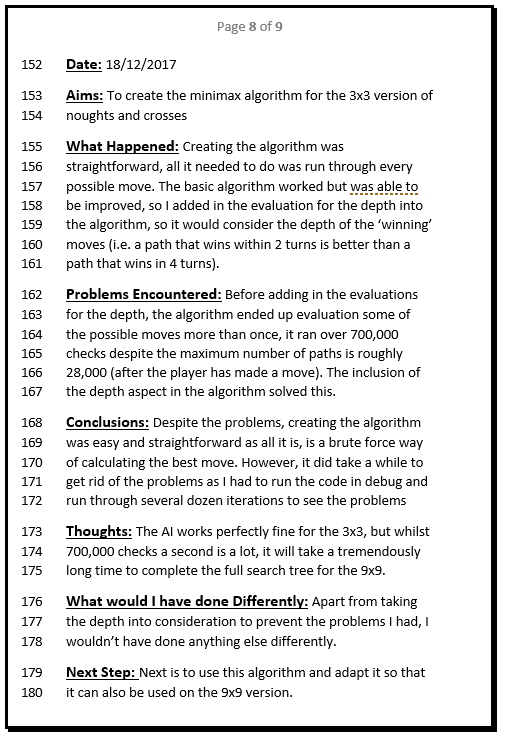


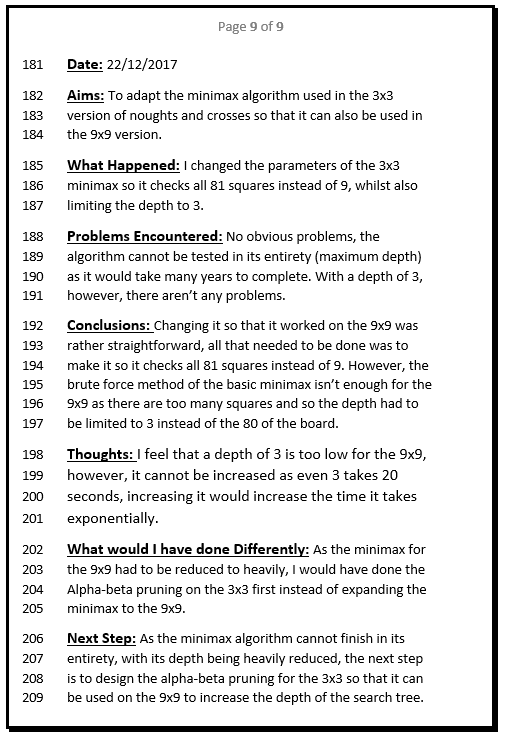


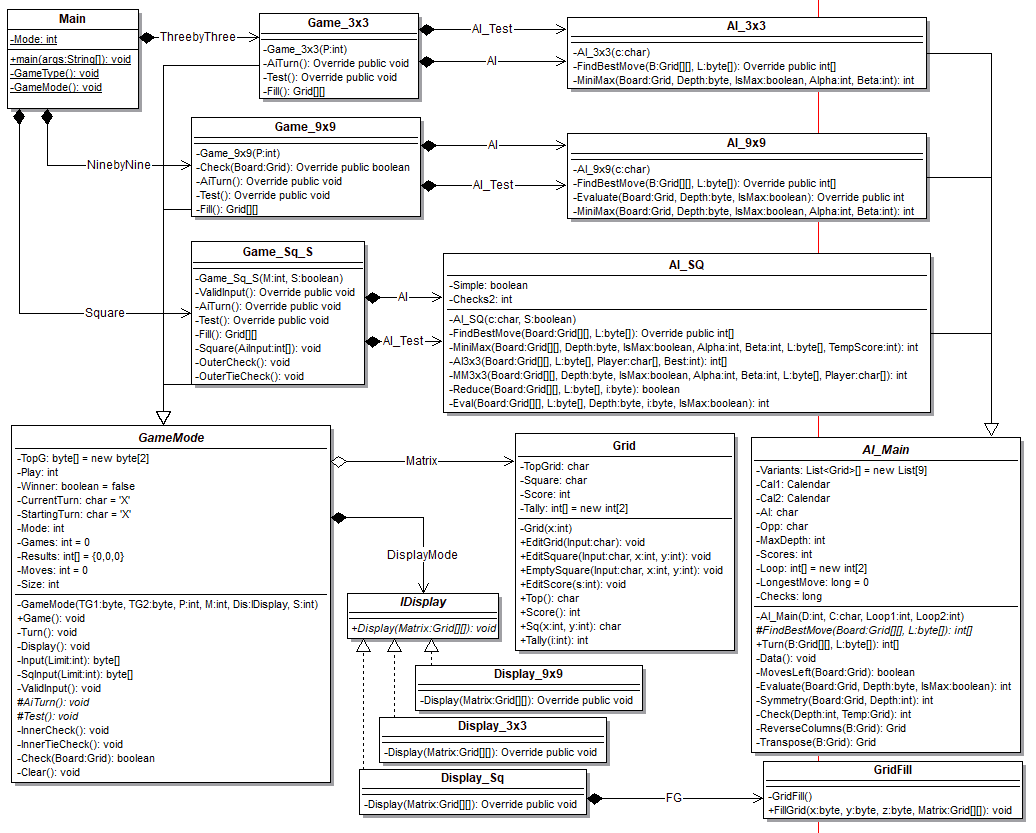










Appendix B – Class Diagram