

AI for games: Making the human think

by

James Norwood

In partial fulfilment of the
requirements for the degree of
Bachelor of Science
in
CS408: Individual Project



Department of
Computer and Information Sciences

March, 2023

Contents

1	Introduction.....	3
1.1	Project Aims	3
1.2	Project Objectives.....	3
1.3	Report Structure	4
2	Background Literature	5
2.1	Minimax.....	5
2.2	Alpha-Beta Pruning	7
2.3	Monte-Carlo Tree Search	10
2.4	The Passive Player.....	12
3	Specification and Design	13
3.1	Analysis	13
3.2	Methodology.....	14
3.3	Requirements	14
3.3.1	Functional Requirements	15
3.3.2	Non-Functional Requirements	15
3.4	Design	16
3.4.1	Interface Design	16
3.4.2	System Design	17
4	Product.....	19
4.1	The Game Class.....	20
4.1.1	The Game State	20
4.1.2	Playing a Move.....	20
23		
4.1.3	Updating the Game State.....	26
4.1.4	The Utility Function.....	27
4.1.5	Additional Methods.....	27
4.2	The GUI Class	28
4.2.1	The board	28
4.2.2	The Information Panel.....	29
4.3	The Player Classes	30
4.3.1	Player Super Class.....	30
4.3.2	The Human Player Class	31
4.3.2	The Random Player Class	31
4.3.3	The Killer Player Class.....	31
4.3.4	The Basic Monte Carlo Player Class	35
4.3.5	The Monte-Carlo Tree Search Player Class	35
4.3.6	The Passive Player Class	38
4.3.7	The Challenging Player 1 Class.....	39
4.3.8	The Challenging Player 2 Class.....	42
5	Results & Evaluation	44
5.2	Results of Evaluation.....	45
5.3	Returning to the Research Questions	47
6	Discussion & Reflection.....	49
6.1	Interpreting the Results.....	49
6.2	Reflection	49

6.3	Challenges	49
6.4	Limitations	50
6.5	Future Work.....	50
7	Conclusion	1
8	References	2
9	Appendix	4
9.1	Ethical Approval Form	12
9.2	Participant Information Sheet.....	13
9.3	Consent Form.....	15
Figure 1:	Visualisation of a Search Tree [5]	5
Figure 2:	Minimax Example [8].....	6
Figure 3:	Minimax Pseudocode [7].....	7
Figure 4:	Minimax with Alpha-Beta Pruning Pseudocode [7]	8
Figure 6:	Minimax with Alpha-Beta Pruning Example Part 2 [6]	9
Figure 5:	Minimax with Alpha-Beta Pruning Example Part 1 [6]	9
Figure 7:	Monte Carlo Search Tree Flowchart [2].....	11
Figure 8:	Possible Moves Design.....	16
Figure 9:	Othello Board Design	16
Figure 10:	Game Information Panel Design	17
Figure 11:	UML	17
Figure 12:	UML with Player Sub Classes	18
Figure 13:	Game Class Constructor	20
Figure 14:	flipTiles() Method Declaration.....	21
Figure 15:	flipTiles() Method	21
Figure 16:	Relative String Position Visualisation	22
Figure 17:	Board Edge String Positions	23
Figure 18:	findFlippedTiles() Method.....	24
Figure 19:	findFlippedTiles() Recursive Code	24
Figure 20:	Recursive Case 1	25
Figure 21:	Recursive Case 2.....	25

Figure 22: Recursive Case 3.....	26
Figure 23: setGameState().....	26
Figure 24: GUI Button Creation.....	28
Figure 25: Tile Designs	29
Figure 26: GUI	30
Figure 27: Minimax playMove() Method	32
Figure 28: Minimax Terminal State Check	33
Figure 29: Minimax Max Method.....	33
Figure 30: Minimax Selecting Move	34
Figure 31: Minimax Min Method.....	34
Figure 32: MCTS Variable Declaration.....	35
Figure 33: MCTS Starting Node Creation.....	36
Figure 34: Setting the Starting Node	36
Figure 35: UCB1 Calculation Method.....	37
Figure 36: MCTS Rollout and Back Propagation.....	38
Figure 37: Passive Player Terminal State Check	38
Figure 38: Challenging Player 1 Variable Declaration	39
Figure 39: Challenging Player 1 Constructor.....	40
Figure 40: Challenging Player 1 Score Ranking.....	40
Figure 41: Challenging Player 1 Relative Value Calculation.....	41
Figure 42: Challenging Player 1 Move Selection.....	41
Figure 43: Challenging Player 2 Relative Value Calculation.....	42
Figure 44: Number of Games Played.....	45
Figure 45: Participant AI Evaluations	46
Figure 46: Participant Experience Evaluation.....	47
Figure 47: Participants' Familiarity with Othello	48

Abstract

This project aims to develop a challenging AI for a two-player game that encourages thoughtful and strategic moves from the user. A fully operational and user-friendly two-player game will be created, along with a range of AI opponents with unique characteristics and levels of difficulty. The project objectives are to select a suitable game, develop the game mechanics, create a user interface, implement search tree algorithms, and develop a challenging AI derived from the optimal AI. The final stage involves user testing and feedback analysis to evaluate the success of the project. This report will provide a background on the research, specification and design of the project, details on the code produced, discussion on the user testing and results, and a conclusion on the project's outcome and main findings.

Acknowledgements

I would like to express my appreciation for my supervisor Dr John Levine for his support and guidance throughout this project. Without the meetings, suggestions and feedback he provided this project would not have been possible.

1 Introduction

1.1 Project Aims

This project aims to develop an AI that can play a two-player game. However, instead of playing optimally, this AI will be designed to challenge the user. To achieve this, the AI will not win every time, as this would discourage the user. Instead, it should put the user in challenging situations that can only be won by thoughtful and strategic moves, encouraging the user to improve their gameplay skills.

To achieve these goals a fully operational and user-friendly two-player game will be implemented. In addition to creating an enjoyable game, a range of AI opponents will be developed, each with unique characteristics and levels of difficulty. The primary focus will be on developing the challenging AI, which when implemented will give the users an experience of strategic and challenging gameplay.

1.2 Project Objectives

The objective of this project is to create a challenging AI for a two-player game. To achieve this goal, the project can be broken down into several distinct stages.

To begin the project, a suitable game will need to be selected for development. There are a few criteria that the game must meet to be eligible: it must be a two-player game, it must have perfect information, and the rules must be easy to understand. Only games that match all of these criteria will be considered.

The second step of the project will involve the development of the game. This will include implementing the functionality and mechanics. This section will focus on making the game playable, ensuring only legal moves can be played, and making the game easy to understand and use.

A user interface will then be developed to complement the game. It should be able to display the events of the game effectively, this will ensure the user understands what is happening. It should be user-friendly and should update in real time so the user can interact with the game.

Next, several AIs will be developed to play against the user. These will range from a random playing AI, up to more advanced optimal AIs. This will involve the implementation of various search tree algorithms, such as Minimax and Monte Carlo tree search. Variations of these will be created to determine the most successful among them.

1 INTRODUCTION

This is when the challenging AI will be developed. It will be derived from one of the algorithms used in the previous step for an optimal AI. As the criteria for this algorithm is harder to define, ideas will be taken from similar research projects and combined with my work. All of this is to create a player that can encourage the player to play well.

This leads to the final stage of the project which will be user testing on the AI. As this player should adapt to whomever it is playing, users with a varying set of skill levels will be selected for testing. They will give their feedback in the form of a questionnaire, stating their opinions on categories such as: how challenging they found it, if they enjoyed it, if they felt encouraged to play more etc. This information will be used to determine how successful the project has been.

1.3 Report Structure

The next section of my report will be on the background reading and research that was completed to be able to implement the code. This section will provide explain the key concepts that will be used within the project and will provide a foundation for the rest of the report to build from.

Section 3 will focus on the specification and design of the project. It will show the methods and processes used to achieve the objectives outlined previously.

Section 4 will focus entirely on the code that has been produced. It will go into extensive detail as to how the game has been created, showing what variables, classes and methods are used, and how they all interact.

Section 5 will discuss how the user testing was conducted, and will show the results that were returned in the form of tables and graphs.

Section 6 will take a deeper look into the results returned from the section before. The feedback will be analysed and then used to evaluate the success of the project.

Finally, my conclusion will be written in section 7. The outcome of the project and the main findings will be stated here.

2 Background Literature

2.1 Minimax

Minimax is a very popular search tree traversal algorithm, typically used in gaming to determine a player's best possible move. In computer science, a search tree is a hierarchical data structure, made by first defining a singular root node, from which leaf nodes can then be added to the tree as children. This makes the root node a parent node, and the leaf nodes are child nodes. These leaf nodes may then become parent nodes if more leaf nodes are added as children to them. A visualisation of this structure is shown below:

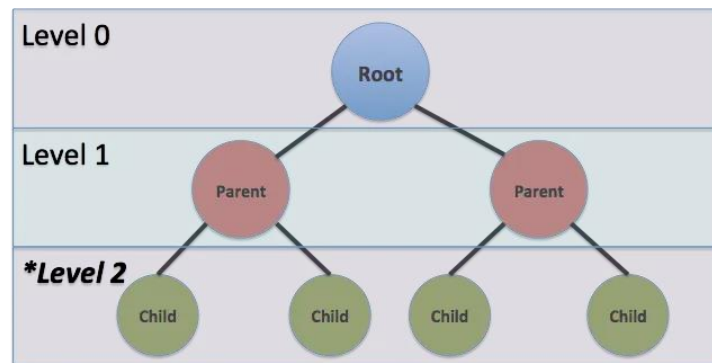


Figure 1: Visualisation of a Search Tree [5]

Putting this in terms of a two-player board game. The root node would represent the current position of the game. The next level of nodes represents the next possible states after a move has been played. This process is then repeated with the next layer representing the next moves and so on.

The size of the tree can be determined in one of two ways. The first is where every possible game state is represented. This would mean layers of nodes are added until they come to terminal states in the game. However, some games' search trees can quickly become too big, when the game has too many, or in some cases, an infinite amount of states. The solution to this problem is to assign a maximum depth to the tree, this ensures that it can only have a certain amount of levels, and therefore cannot become too large for a computer to handle.

Now that the tree is complete, essentially a map has been created of every possible outcome of the game; The Minimax algorithm can use this map to find the optimal move. But first, a utility function

2 BACKGROUND LITERATURE

must be created. A utility function is used to evaluate, and then give a score to a node. This score should reflect how well a player is doing in that state of the game, the higher the score the more likely that position will lead to a winning game.

Minimax works on the assumption that the opposing player is also playing optimally. If the opponent is not playing well, it may affect the success of the Minimax player. But typically, Minimax should still play well.

The algorithm is comprised of three functions: Maximise, Minimise and Utility function. It is made up of recursive calls; Maximise calls Minimise, Minimise calls Maximise and this is repeated until the maximum depth of the tree, or a terminal state is reached. When this happens the Utility function is called, and the score of that state is returned.

In Figure 2 below, an example is shown of a situation where the computer (orange) is trying to maximise the score, and the human player (blue) is trying to minimise it.

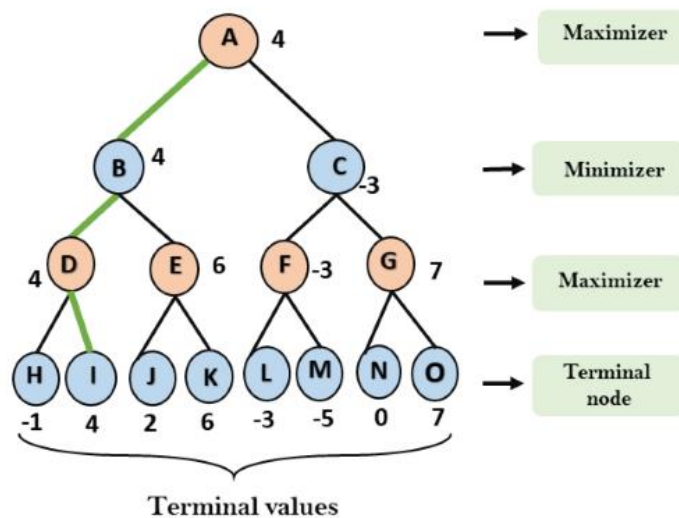


Figure 2: Minimax Example [8]

At the top layer, the computer player first calls the maximise function; This returns the highest score out of the two child nodes. The score of these child nodes (blue) is determined by calling the minimise function; This returns the lowest score out of its child nodes. This process is repeated, alternating

2 BACKGROUND LITERATURE

between calling the maximise and minimise functions, until a terminal state is reached. Once this happens the utility function is called on the terminal nodes, and the score of those nodes is passed back up the tree.

The algorithm can be implemented using the pseudocode shown in Figure 3 below. The key thing to note is that the value 'v' is being set before the minimise/maximise function is being called on the children (successor in this example) nodes. This value is usually set to negative infinity for a minimal game value and positive infinity for a maximal game value.

```
Max-Value(state) returns a utility value
  if Terminal-Test(state) then return Utility(state)
  v ← MinimalGameValue
  for s in Successors(state) do
    v ← Max(v, MinValue(s))
  return v

Min-Value(state) returns a utility value
  if Terminal-Test(state) then return Utility(state)
  v ← MaximalGameValue
  for s in Successors(state) do
    v ← Min(v, Max-Value(s))
  return v
```

Figure 3: Minimax Pseudocode [7]

2.2 Alpha-Beta Pruning

As discussed earlier, the minimax algorithm may take a long time to run, as the search tree can very quickly become too big. This can be combatted by adding a minimum depth, but the fewer layers in a search tree, the poorer the performance of the algorithm. One way to decrease the time for the algorithm to run, without decreasing the accuracy of the results is to incorporate alpha-beta pruning. As the algorithm will run faster, the size of the tree can be increased; This means the speed and results of the algorithm will be improved.

The Minimax with alpha-beta pruning algorithm improves upon the standard Minimax algorithm by not searching through the entire search tree. It can determine when a branch of the tree will not affect the overall result; If one of these branches is discovered, the algorithm will waste no time traversing it. Doing this can lead to large parts of the search tree being removed, therefore a lot of time will be saved.

2 BACKGROUND LITERATURE

To achieve this, two parameters have to be added to the maximise and minimise functions: alpha and beta. Firstly, alpha is used to represent the best value that the maximise function can currently guarantee at that level. Secondly, beta is used to represent the best value that the minimise function can currently guarantee at that level. [1]

Below is the updated pseudocode for the maximise and minimise functions to incorporate alpha-beta pruning:

<pre>Max-Value(state, α, β) returns a utility value if Terminal-Test(state) then return Utility(state) $v \leftarrow \text{MinimalGameValue}$ (initialize as $-\infty$) for s in Successors(state) do $v' \leftarrow \text{Min-Value}(s, \alpha, \beta)$ if $v' > v$, $v \leftarrow v'$ $v' \geq \beta$ then return v if $v' > \alpha$ then $\alpha \leftarrow v'$ return v</pre>	<pre>Min-Value(state, α, β) returns a utility value if Terminal-Test(state) then return Utility(state) $v \leftarrow \text{MaximalGameValue}$ for s in Successors(state) do $v' \leftarrow \text{Max-Value}(s, \alpha, \beta)$ if $v' < v$, $v \leftarrow v'$ $v' \leq \alpha$ then return v if $v' < \beta$ then $\beta \leftarrow v'$ return v</pre>
--	--

Figure 4: Minimax with Alpha-Beta Pruning Pseudocode [7]

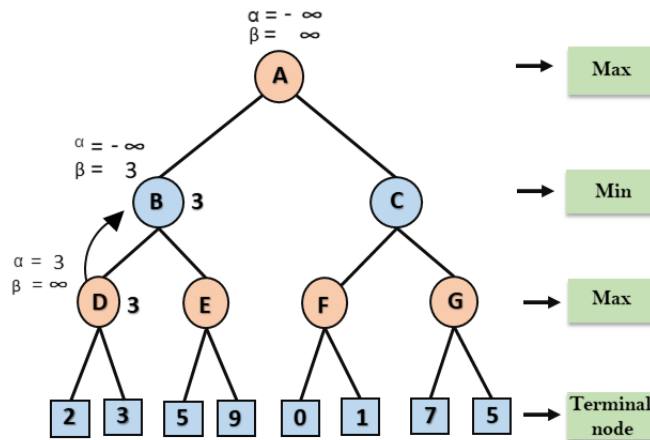


Figure 6: Minimax with Alpha-Beta Pruning Example Part 1 [6]

The best way to understand this algorithm is with an example. As shown in Figure 5 above, we can see that the algorithm starts the same as regular minimax would. When we get to node D, the value of 3 is returned. The only difference in the new algorithm is that the alpha and beta values are included. At node D, alpha is set to three as it is the best value the maximise function can return on that node. At node B the value of beta is set to 3 before calling minimise function on node E. This is because 3 is the best value that node B can currently return. Therefore, if we can determine that the value of node E will be greater than 3, we do not need to search all of its child nodes. This is shown below:

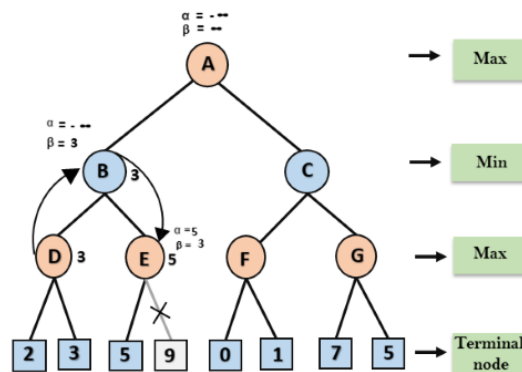


Figure 5: Minimax with Alpha-Beta Pruning Example Part 2 [6]

2 BACKGROUND LITERATURE

As we can see, when the maximise function on E is called, the first child node returns 5. Since 5 is greater than the beta value of 3, this branch of the tree can be pruned as it will not affect the outcome.

In the above example, the value of the node on the tree that is being cut off is 9, this is greater than the first node which was 5. This shows that if the higher value of 9 was considered, E would instead return 9, but since B is a minimum node, B will still return 3.

If instead, the second child node of E was lower, E would return 5, and then B would still return 3. This proves the second child node of E cannot affect what B returns, meaning the algorithm can save time, by not searching that part of the tree.

While this may seem insignificant for a relatively small tree, as in the example above. When much larger more complex search trees are being analysed, alpha-beta pruning can cause large areas of the tree to be skipped, therefore saving a lot of time.

2.3 Monte-Carlo Tree Search

Monte-Carlo tree search is another search tree algorithm that is very popular in the use of gaming to determine the best move. This algorithm however is quite different in its implementation to minimax. Some advantages of this algorithm are being able to run for a specified amount of time, it works well with larger search trees as it is not exhaustively searching through every possibility as minimax does, and there is no need to implement a heuristic function. While these advantages can be beneficial, as this algorithm doesn't search every possibility, it can sometimes be less accurate than the min-max algorithm depending on the problem it is being used for. The algorithm itself is also more complex to implement than minimax. It can be broken down into four phases [2]:

- I. Tree Traversal – The first step in MCTS (Monte Carlo Tree Search) involves finding the node that is going to be expanded. This is determined by selecting the leaf node with the highest UCB1 score. UCB1 is calculated with the following equation:

$$UCB1 = V_i + 2 \sqrt{\frac{\ln N}{n_i}}$$

- V_i = The average value of all nodes beneath this node
- N = The number of times the parent node has been visited

- n_i = The number of times the child node has been visited

If MCTS is being implemented for a two-player game, it is important to note that the 'max' is trying to find the highest UCB1 score from its children. But the 'min' player is trying to find the lowest UCB1 score.

- Node Expansion** – When the leaf node with the highest UCB1 value is found, if the node has never been visited before ($n_i = 0$) then a rollout is performed. If the node has been visited before, node expansion occurs. This means all of the current leaf nodes children are added to the tree. The first new child that is added becomes the current node, and then a rollout is performed.
- Rollout** – When a rollout is performed. A random simulation is run from the current node until a terminal state is reached. When a terminal state is reached a score is returned.
- Backpropagation** – The backpropagation phase of the algorithm involves passing the value returned from the rollout backwards up the tree through all of the parent nodes. The value being passed back up is added to the value of the predecessor nodes, and their n_i values are incremented by one.

Shown below is a flowchart of one iteration of the MCTS algorithm. This can be run for a specified number of iterations, or for a pre-determined time.

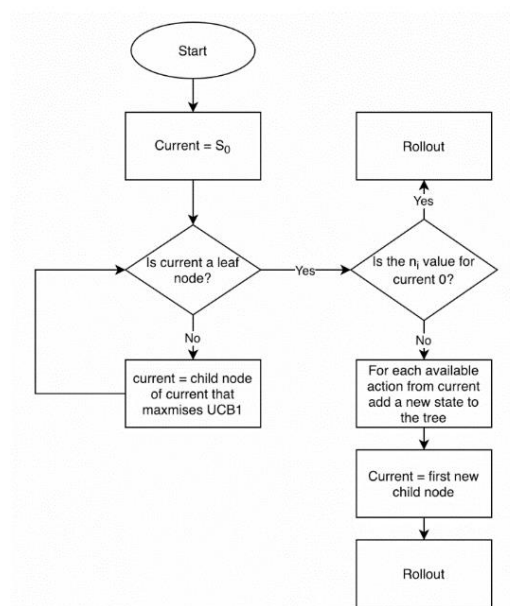


Figure 7: Monte Carlo Search Tree Flowchart [2]

2 BACKGROUND LITERATURE

This can be repeated as many times as desired, the longer it runs the more successful it will be at finding an optimal move. This is because the UCB1(Upper Confidence Bound 1) score is used to create a balance of which nodes are searched. Meaning the algorithm will search through both promising nodes that haven't been well explored, and nodes with high estimated values.

2.4 The Passive Player

The passive algorithm was developed in 2022 by University of Strathclyde student Eve Rigley as part of her fourth-year project 'An AI That Plays Fairly: The encouraging Teacher [3]'. The research in this project aimed to develop an AI that is matched to the player's skill level and creates an enjoyable game that encourages them to play more. The algorithms developed as part of the project were designed for the board game Othello (referred to as 'Reversi' in Rigley's report).

Several different AI players were developed as part of this project that were designed to work with Othello. This included random, minimax with alpha-beta pruning, Monte Carlo tree search and passive players were created. While the MCTS (Monte Carlo tree search) and random players can work with any game. Utility functions needed to be created specifically for Othello for the minimax and passive players.

Several different utility functions were experimented with, but in the end, Rigley found the best utility function made use of four factors to evaluate a game state. These factors were the difference in the number of tiles, the difference in the number of available moves, the difference in the number of corners and the difference in the number of stable tiles. These four factors were given a weighting and added together to provide a score out of 100. The equation for the final utility function was as follows:

$$\text{Score} = (0.15 * \text{Tile Difference}) + (0.15 * \text{Choice Difference}) + (0.4 * \text{Corner Difference}) + (0.3 * \text{Stability Difference})$$

This function was used to successfully evaluate the terminal states of a search tree that was used first in the minimax with alpha-beta pruning algorithm and secondly adapted to be used in the passive player algorithm.

The final iteration of the passive player that was developed for Rigley's project was based on minimax with alpha-beta pruning. However, instead of the minimax algorithm trying to find the optimal move,

it aims to find the move that will keep the game the most balanced. This was done by determining how close the utility function was to 50. Rigley has done this by evaluating the utility function on the terminal states and subtracting this number from 50, then the absolute value of this

was taken to determine how far from zero this number was. This number was then subtracted from 100 – 100 was used as an arbitrary constant so that the lower numbers that were closer to zero now had a higher value than the numbers further from zero. Since the values closest to 50 were given the highest results from the utility function, the minimax algorithm could run as normal and play a passive game instead of an optimal one.

3 Specification and Design

This section will look into the specifics of the design and tools used to achieve the objectives of the project. It will first explain the methodology used throughout the development, and why it proved advantageous. Then shall shift focus to the project's requirements, why these decisions were made and how they were prioritised.

3.1 Analysis

The game chosen for this project was Othello. Some requirements had to be met for a game to be suitable, and many options met all of these requirements. So ultimately it came down to personal preference as to which game I felt would be more comfortable and enjoyable to work with.

The first requirement was very simple: the game must have 2 players. This is because all of the search tree algorithms used in the creation of these AI are designed to work optimally with these two players.

The second requirement that had to be met was a game with perfect information. This means that each player can know the state of the game at all times, and they can see every move that has been made, and every move that can be made. Essentially nothing is hidden from either of the players and there can be no random elements that affect the game's outcome. Some examples of games with perfect information are chess, tic tac toe, and Othello.

The reason the game must have perfect information is to ensure the AI search tree algorithms work properly. These algorithms essentially work on the same premise of looking through all potential moves of the game and calculating the best one. This idea would not work if randomness because a

3 SPECIFICATION AND DESIGN

factor in the game, as it would be impossible to judge the future states of the game; the AI wouldn't be able to calculate the best move, as there is no way to know what might happen next.

And finally, the game must be easy to understand - this will be most important when doing user testing. The game must be easily accessible to a wide variety of users, this will return a wider range of opinions from people of different skill levels. As the AI that is being developed should adapt to all users, it is essential that players who have never before played be involved in testing. Therefore the game's rules should be very simple; a user should be able to understand them almost immediately. This could be games such as tic tac toe, Othello or connect four.

This led to Othello being chosen for the game to be developed as it meets all of the above criteria. It is a two-player game(expand). It has perfect information which will mean the AI algorithms that I will be implementing will work well. And it has very simple rules that anyone should be able to pick up very quickly. The rules are so simple that even the slogan Othello has been advertised for decades is "A minute to learn... A lifetime to master."

3.2 Methodology

The main objective of this project can be compartmentalised into some key stages for development as shown in 1. Introduction. The three areas that needed to be developed to make a playable version of Othello were: the GUI and the AI players, and of course the game itself. This led to the use of an Agile methodology throughout the implementation. This method allowed for more flexibility in the development, as different sections could be worked on in no particular order. Going back and forth between the development of the GUI and the game made it easier to find and fix bugs that were present, in each of them

3.3 Requirements

Before the development of Othello began, all of the functional and non-functional were decided upon. These rules and guidelines were taken from the 'World Othello Organisation's official rules [4]. In this guide all requirements for that game are laid out. This includes the size and shape of the board, instructions for gameplay and the rules. All of this information can be condensed into these functional and non-functional requirements.

3.3.1 Functional Requirements

The functional requirements are what is needed to develop a functioning game. All of these goals must be developed for the game to be considered complete. Below they are listed in the order of development, this is from the most essential being at the top to the least essential at the bottom.

- The game board must be 8x8 squares in size
- The starting state of the game must have 2 black and 2 white tiles in the board's centre
- The game state must update when a move is played
- The game state shouldn't update when an illegal move is played
- The board must update when the game state changes
- The game must detect when it has ended
- The GUI must display who has won when it has ended
- The user must always play first
- The game should be able to be forfeited
- A new game should be able to be started

3.3.2 Non-Functional Requirements

The non-functional requirements are not essential to the gameplay, these focus on how the user should experience the game.

- The game should be easy to use/understand
- The GUI should update after the user selects their move
- The user should feel challenged when playing the challenging AI
 - They should not be able to win easily
 - They should not feel discouraged
 - They should be able to win if they play good moves
- It should be clear what move the AI has played

3.4 Design

3.4.1 Interface Design

The GUI for this project will be based on the official Othello game board. The board is comprised of an 8x8 grid of green squares, black or white circular tiles can be placed inside any of these squares. Figure 9 below shows how the starting position of the game should look with two black and two white tiles placed in a diagonal pattern in the centre of the board.

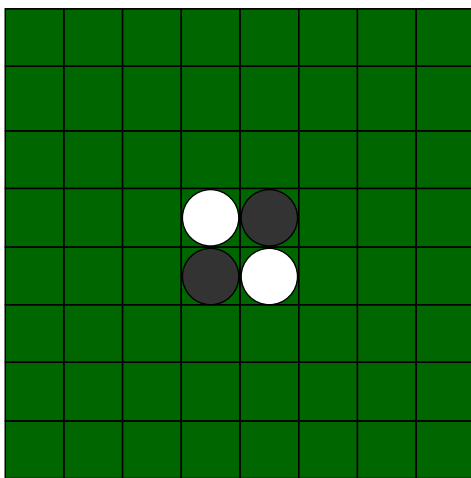


Figure 9: Othello Board Design

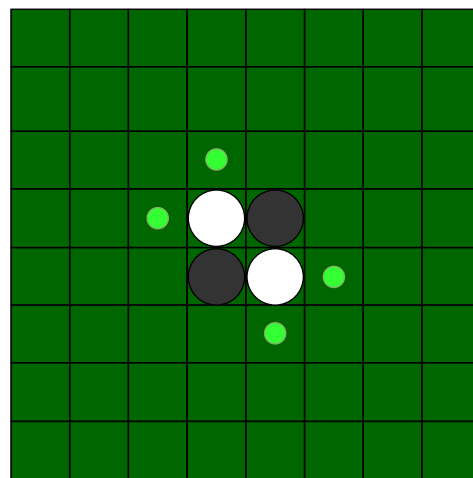


Figure 8: Possible Moves Design

Figure [8] above shows an additional feature that is not on the original game board. Pictured here are small, brighter green circles that appear on the tiles where a player has the option to play a move. Adding these circles will make the rules of the game clearer to new players, and will make the game easier to play.

3 SPECIFICATION AND DESIGN

A panel will also be shown below the game board containing the score of the game, whose turn it is, and who has won when the game is over. It will be minimalistic in its design and only provide this essential information as shown below:

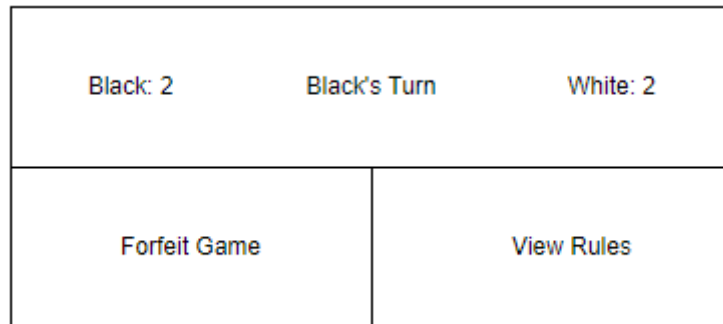


Figure 10: Game Information Panel Design

3.4.2 System Design

The design of this project required a lot of classes that all worked together. Because of the sheer number of methods and variables within these classes, a simplified UML can be seen below that shows all of the relationships and dependencies between the classes. The specifics of methods are variables will be discussed later in section 4.

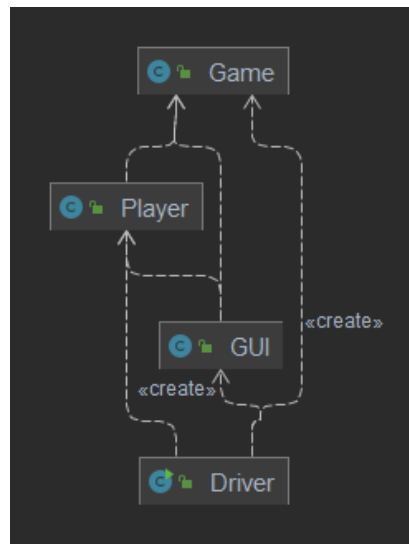


Figure 11: UML

3 SPECIFICATION AND DESIGN

As can be seen in figure[11] The driver is the point from which all other classes are created. From within here, a menu is provided allowing the user to select what AI they would like to play against. Once selected the driver will create a new game object, a new GUI object, and two new player objects – one a human player and the other is determined by the user's choice.

It can also be seen that the GUI class interacts with both the player classes and the game class. This is due to all of the human interaction that occurs within this class. Once a button is clicked here, the play move method is called in the player class, and from there the flip tiles method is called in the game class. The GUI can then interact directly with the game object to check the new game state to update its display.

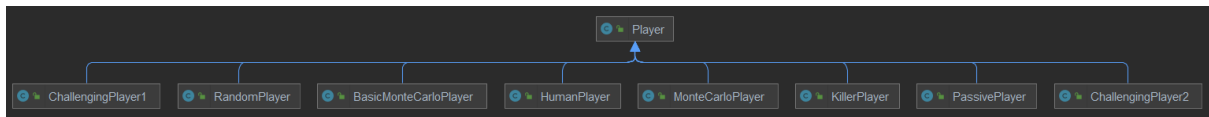


Figure 12: UML with Player Sub Classes

Above the player, the class is shown with all of its implementations. Again the methods contained within these classes will be expanded upon in section 4, but the main methods that are required in all player sub classes are `playMove()` and `getColor()`. `playMove()` is the method that is called from the GUI, each sub class has a vastly different implementation of this method to determine what move to play. But they all end with the `flipTiles()` method being called in the game class to update the game's state.

4 Product

This section will review the implementation of the game's code. It will take a look at what language was used, the code's structure, and why these decisions were made. The latter section delves into what testing was completed to ensure that all of the requirements as stated in 3.3 were met.

Before starting implementation, a language had to be decided upon to code in. It was decided that Java was the most suitable language to undertake this challenge. The reasons for this are: Firstly, Java is an object-oriented language, which will help to create efficient and reusable code. This will be done by creating modular code that is written across multiple files and classes. Secondly, Java provides many pre-existing data structures and libraries. As these data structures and libraries already exist it will save a lot of time during the implementation of the game the GUI and the AIs. And finally, Java is a very powerful and high-performance language. This will help to improve the speed at which the search tree algorithms are executed.

The code for this project was completed completely from scratch, and doing this allowed for a complete understanding of how the code operated and how it was structured: This proved extremely useful when implementing classes that interact with each other. As explained in section 3.1 an agile methodology was used throughout the development process. An agile methodology allowed for work to be done on the three main sections of the project simultaneously, this being the game, the GUI, and the AI players.

While the classes were developed simultaneously, a foundation had to be created for the rest of the project to build from. This led to the most important class – the game class – being developed first. It was decided that this was most important, as the majority of the functional requirements were to be contained within this class.

4.1 The Game Class

4.1.1 The Game State

The game itself is contained within one class. When a new game is started from the driver, a new instance of the game class is created. This class's main responsibilities are: storing the state of the game, updating the state of the game when a move is played, and running a utility function to give each position a score.

The most integral part of the code is the game's state. It is of course essential to the functionality of the game and the GUI, but it is also used in all of the AI search tree algorithms. As the game's state is used across all classes, it must be stored in a versatile format. This means the variable type should be easy to understand when coding, but it also must be able to run quickly when used in a search tree. It was decided that the state would be stored as a 64-character string – a character for each position on the board. A '-' will represent when there is no tile on a square of the board. An 'X' will represent where a black tile has been placed, and an 'O' will represent where a white tile has been placed.

When a new instance of the game class is created, the game state will be initialised inside the constructor as follows:

```
public class Game {  
  
    private String gameState;  
    public Game() { gameState = "-----X0-----0X-----"; }  
}
```

Figure 13: Game Class Constructor

This sets the game's state to the starting position of the game.

4.1.2 Playing a Move

A move can be played in one of two ways: By a human player, or by an AI player. For both cases a position on the board is selected, this is a number between 1 and 64. 1 represents the top left position of the board, and 64 represents the bottom right. As the number increases the tiles are determined from left to right, then top to bottom. The specifics of how these tiles are selected will be discussed later, but for now, the functions that play the move will be explained.

When a tile is selected, the tile's number, along with the player's colour, is passed into a function, inside the game class named flipTiles().

```
public ArrayList<Integer> flipTiles(int position, char color) {

    ArrayList<Integer> flippedTiles = new ArrayList<>();
```

Figure 14: flipTiles() Method Declaration

When the flipTiles() method is called, first an ArrayList is declared. This will hold the positions of every tile that should be flipped when the move that is passed into the function is played.

Next, there is some error checking in place on line 35 as shown below. This line checks to see if a tile has already been placed in the position that was selected. If a tile has yet to be placed in the position selected, the function will go on to find all tiles that need to be flipped as a result of a new tile being added at that position.

```
35     if (getColor(position) == '-') {
36
37         //The exclusions ArrayList is used to avoid out of bounds errors
38         //If the position of the tile is at an edge, the algorithm won't try to look beyond the edge
39         ArrayList<Integer> exclusions = new ArrayList<>();
40         if (position < 9) { exclusions.addAll(new ArrayList<>(Arrays.asList(-7,-8,-9)));} // Top row
41         if ((position - 1) % 8 == 0) { exclusions.addAll(new ArrayList<>(Arrays.asList( 7, -1,-9)));} // Left Column
42         if (position % 8 == 0) { exclusions.addAll(new ArrayList<>(Arrays.asList(-7, 1, 9)));} // Right Column
43         if (position > 56) { exclusions.addAll(new ArrayList<>(Arrays.asList( 7, 8, 9)));} // Bottom row
44
45         flippedTiles.add(position);
46
47         for (int i = 7; i < 10; i++) {
48             if (!exclusions.contains(i)) {
49                 flippedTiles.addAll(findFlippedTiles(position, i, color, new ArrayList<>()));
50             }
51             if (!exclusions.contains(-i)) {
52                 flippedTiles.addAll(findFlippedTiles(position, -i, color, new ArrayList<>()));
53             }
54         }
55         if (!exclusions.contains(1)) {
56             flippedTiles.addAll(findFlippedTiles(position, direction: 1, color, new ArrayList<>()));
57         }
58         if (!exclusions.contains(-1)) {
59             flippedTiles.addAll(findFlippedTiles(position, direction: -1, color, new ArrayList<>()));
60         }
61     }
62
63     return flippedTiles;
```

Figure 15: flipTiles() Method

The tiles that need to be flipped are determined by calling a recursive function named `findFlippedTiles()`. This function is called 8 times tile, once for each direction from where the tile has been placed.

Here is a visualisation below, this shows a 3x3 segment of the top left of the board. The position of the new tile being placed is in the centre – in this case, it would be position 10. The arrows indicate the 8 directions in which the `findFlippedTiles()` method will be called. Since the game state is a string, these directions are stored as the positions of the tiles relative to the tile being placed. This can be shown in the figure on the right.

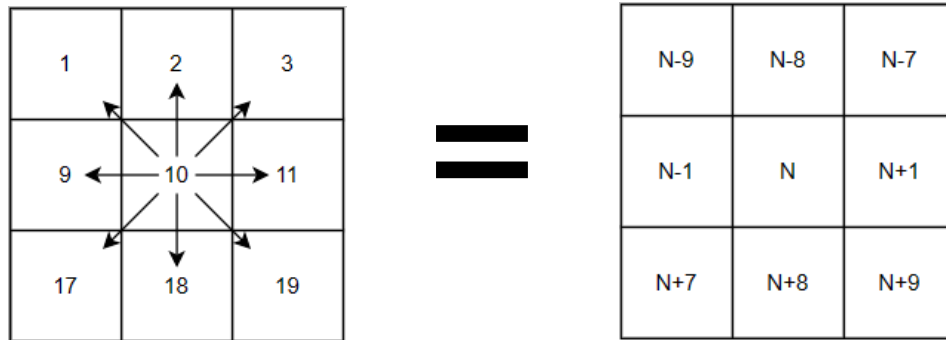


Figure 16: Relative String Position Visualisation

Before `findFlippedTiles()` can be called 8 times, there needs to be a check in place to determine if a new tile is being placed at the edge of the board. If this check wasn't there it would mean 'out of bounds errors' or logic errors would occur when the algorithm tries to search beyond the edges of the board. This problem is managed in lines 39-43 in the above extract of code. Here an `ArrayList` called 'exclusions' is created and populated. It contains the directions that the `findFlippedTiles()` method should not be called in. Below is shown an 8x8 grid representing the board, and the coloured lines show how a tile is determined to be on the edge. If it is shown to be on an edge, the three directions that would point beyond the edge of the board are added to exclusions.

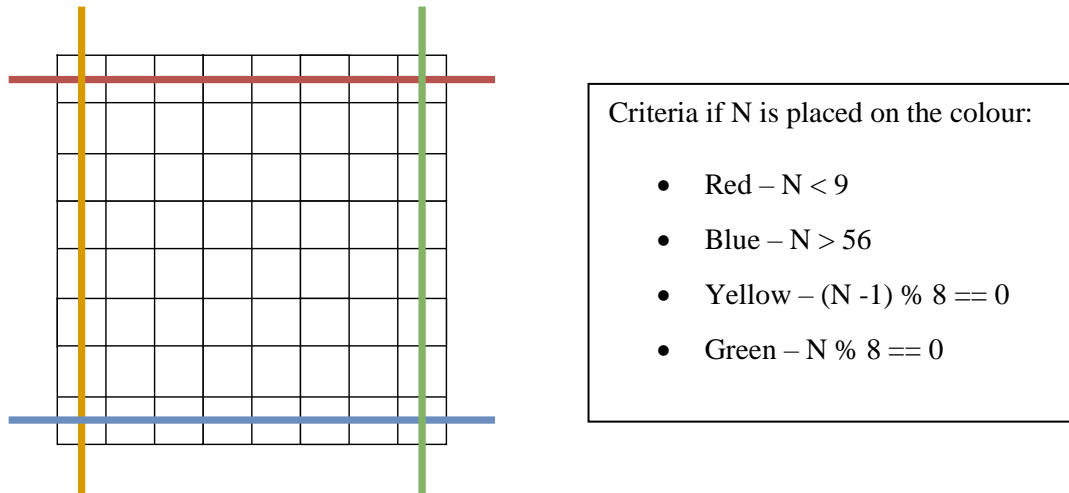


Figure 17: Board Edge String Positions

If the position of the new tile that is being placed matches any of the criteria shown above, the relative exclusions are added. This ensures the next segment of the code (lines 46-60) runs without fault. These lines are responsible for looping through all 8 directions from the new tile (excluding those directions contained within the ArrayList 'exclusions') and calling the recursive function `findFlippedTiles()` once per direction. The function returns what tiles need to be flipped and adds them to the ArrayList 'flippedTiles' that was defined at the start of this method.

The `findFlippedTiles()` method as shown below, is a recursive function that takes the parameters:

- current – the position of the move selected
- direction – the direction this method is searching
- color – the player's colour
- tiles – this is an ArrayList of all tiles that need to be flipped for this direction

The method returns the tiles that need to be flipped in the form of an ArrayList of Integers that contains their positions in the game state string. If no tiles need to be changed, then an empty ArrayList is returned.

```

67 public ArrayList<Integer> findFlippedTiles(int current, int direction, char color, ArrayList<Integer> tiles){
68
69     //If at the edge of board, return empty array
70     if ((current < 9) && (direction == -7 || direction == -8 || direction == -9)) {return new ArrayList<>();}
71     if (((current - 1) % 8 == 0) && (direction == -9 || direction == -1 || direction == 7)) {return new ArrayList<>();}
72     if ((current % 8 == 0) && (direction == -7 || direction == 1 || direction == 9)) {return new ArrayList<>();}
73     if ((current > 56) && (direction == 7 || direction == 8 || direction == 9)) {return new ArrayList<>();}

```

Figure 18: findFlippedTiles() Method

The above extract of code shows a similar process as that that was used in the flipTiles() method. This returns an empty ArrayList if the edge of the board is met, and a 'sandwich' hasn't been made with the player's original tile yet. This means that no tiles need to be flipped.

If this is not the case, three other possibilities might occur when calling this method. These are shown in the section below:

```

75     if (getColor( position: current + direction) == '-') { //If this direction ends on an empty tile
76         //Return an empty array as no tiles need to be flipped
77         return new ArrayList<>();
78     } else if (getColor( position: current + direction) != color) { //If the next tile is not the same as the players
79         ArrayList<Integer> newTile = new ArrayList<>();
80         newTile.add(current + direction);
81         ArrayList<Integer> flippedTiles = findFlippedTiles( current current + direction, direction, color, newTile);
82         if (flippedTiles.size() == 0) {
83             //if finFlippedTiles returns 0 tiles, no new tiles need flipped, an empty array is returned
84             return new ArrayList<>();
85         }
86         tiles.addAll(flippedTiles);
87         return tiles;
88     } else{ //If the next tile is the same as the players
89         //Return the input tiles. No new tiles must be flipped
90         return tiles;
91     }
92 }
93

```

Figure 19: findFlippedTiles() Recursive Code

- I. Line 75 - The first possibility is that the next direction has no tile placed on it already. If this is true, no tiles need to be flipped in this direction as the last tile should be the same colour as the player. Therefore a new, empty ArrayList will be returned. An example is shown below where this is true. A black tile has been placed in the top left, searching to the right.

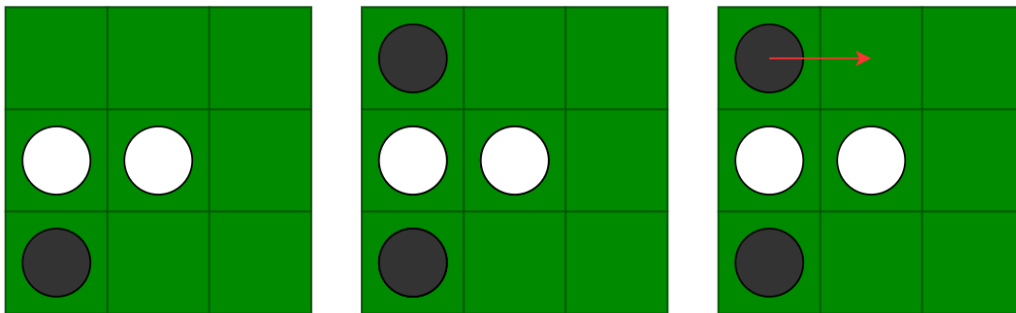


Figure 20: Recursive Case 1

- II. Line 79 – The second possibility is that the direction being searched is the opposite colour from the player. Since it is unclear how many tiles in the row may need to be flipped, the use of the recursive algorithm is required. The findFlippedTiles() method is called again, starting at the same tile, but this time pointing towards the bottom right. An example is shown below where this recursive part of the function would be called once, and the no tile would be found the second time it is called (similar to the previous example)

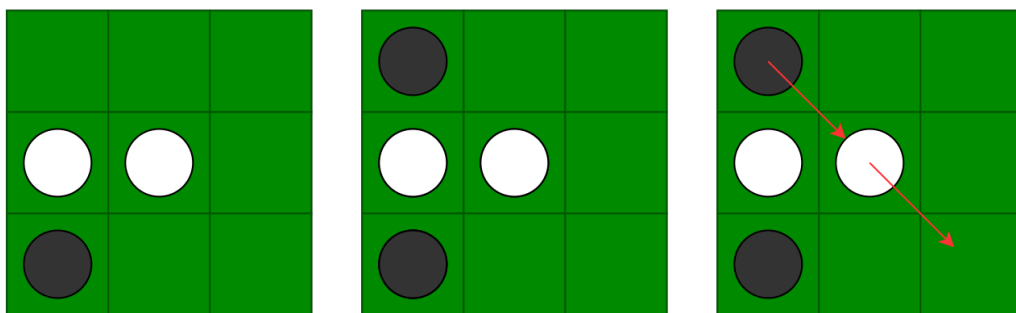


Figure 21: Recursive Case 2

This example also returns an empty ArrayList.

- III. Line 90 – The final possibility is that the next tile in the direction being searched is the same colour as the player. When this is the case all tiles in-between are returned in an ArrayList. Below is an example where the first tile is white – making the recursive call – and the second is black.

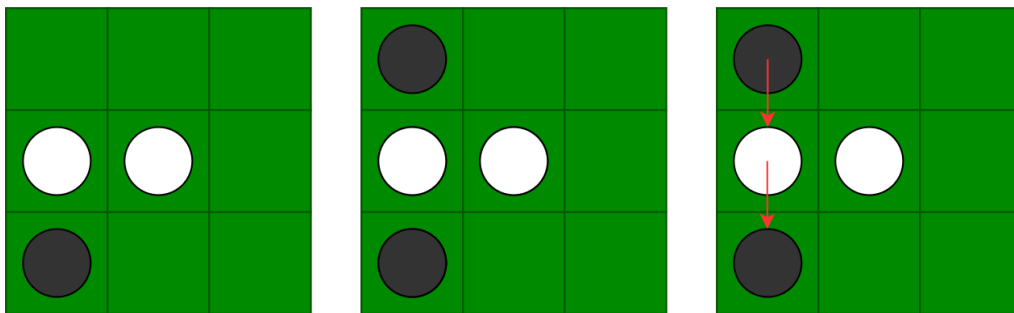


Figure 22: Recursive Case 3

This example returns an ArrayList containing the position in the game state string of the top-left tile and the middle-left tile.

Once the findFlippedTiles() function has been called for every direction, the ArrayLists returned are combined into the ArrayList that was declared earlier in Figure 14.

4.1.3 Updating the Game State

Once the flipTiles() method has been called, an ArrayList will be returned containing the position of every tile that needs to be flipped. To apply these changes, the setGameState() method needs to be

```

13 @ public void setGameState(ArrayList<Integer> tilesToFlip, char color){
14
15     for (Integer flippedTile : tilesToFlip) {
16         gameState = gameState.substring(0, flippedTile-1) + color + gameState.substring(flippedTile);
17     }

```

Figure 23: setGameState()

called. The parameters being passed in are the output from the flipTiles() method and the player's colour. Here is the method below:

The method itself is rather simple; For every Integer in 'tilesToFlip', the colour of the player is inserted into the integer's position in the game state. This is done by creating two sub-strings from the game state; The first sub-string contains the game state up to one position before the integer, and the second contains the rest of the game state after the integer. The character representing the player's colour ('X' for white, 'O' for black) is inserted in between these two substrings, and the 'gameState' variable is then set to be this new string.

4.1.4 The Utility Function

The implementation of the utility function is based on the research done by Eve Rigley as discussed in section 2.4 previously. This requires the difference in tiles, possible moves, corners and stable tiles to be calculated. This function will be called from within the AI player classes to determine the scores of terminal states in search trees.

4.1.5 Additional Methods

The game class also contains some additional methods that can be called from other classes. These mainly being getters, setters and the utility function used for the min-max algorithms.

The three getters are used to get the game state, get the colour of a tile at a specific position, and get an ArrayList of all the possible moves at the current state.

The setter method used is called 'resetGameState'. This takes a fully new state as its parameter and completely resets the game's state.

The other methods are all used for the utility function to evaluate the current state of a game. They were all implemented based on the research covered in section 2.4. Functions were created to get the difference in the number of corners, the difference in the number of tiles, the difference in the amount

of possible moves, and the difference in the number of stable tiles. The utility function was then created to combine the values of these methods to return a score for the game state. As Eve Rigley found success with her algorithm last year, the same weightings were used on each variable as Rigley used last year.

4.2 The GUI Class

4.2.1 The board

The GUI has been designed to fulfil the design requirements as described in section 3.3. This includes more of the functional requirements as well as non-functional requirements. It has also been created to match the designs shown in Figure 9 and Figure 8.

The board section of the GUI had to be comprised of 64 tiles in an 8x8 grid, each with the ability to be updated with new tiles, and selected by the user to play a move. It was decided that this would be done using 64 JButtons. This allowed for easy implementation of eventListeners to tell which move had been played and also created an efficient way to update what tile was shown on the square by changing the button's image icon.

The creation of the 64 buttons can be shown below, along with a new ArrayList called 'possibleMoves'. 'possibleMoves' uses the findPossibleMoves() method from within the game class to show all available moves a player has. Since black always plays first, the character for which possible moves are found is set to 'O'.

```

51         final char[] color = {'O'};
52
53         JButton[] buttons = new JButton[64];
54         ArrayList<Integer> possibleMoves = new ArrayList<>(game.findPossibleMoves(color[0]));

```

Figure 24: GUI Button Creation

A method named setColors() was then created to be used to update the board at the start of the game, and every time the game's state changes. It takes the JButtons, the previous game state and the current game state, and the possible moves as its parameters. It then creates a loop that iterates through all 64 buttons, if the position of that button has changed from the previous state to the current state, its image icon is updated to a GIF to make it look like it has been flipped over. It will then be changed back to a stationary tile, along with all of the other tiles that were never flipped. This

loop will also update any of the tiles that are possible moves to have a bright green circle inside of them. Below is shown the final design for the board's tiles:



Figure 25: Tile Designs

Once the board has been initialised, event listeners are then created for each of the buttons. The position of the button that is pressed in the array is used as a parameter to call the `playMove()` method within the 'HumanPlayer' class – this method is discussed in 4.3.2. If a legal move was selected (if the game state has changed) then the `setColors()` method is called, and then the `playMove()` method is called for the AI player. Once the AI has moved, the board is updated again and the human player will be allowed to select another move.

4.2.2 The Information Panel

The information panel appears below the game board. This will contain the scores of the game, whose turn it is, and when the game is over it will show the winner.

From here the user will also be able to forfeit the game to be able to start a new one. Or they will be able to view the rules of the game.

The final version of the GUI is shown below:

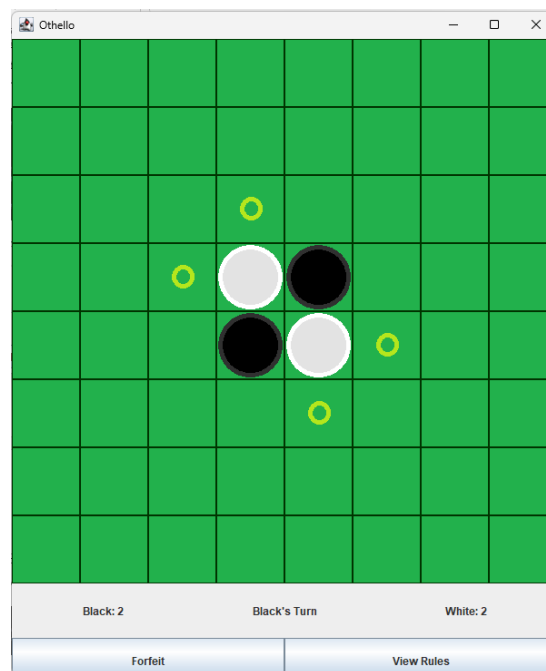


Figure 26: GUI

4.3 The Player Classes

As laid out in the game's UML in section 3.4 There are 8 different player classes that all extend the super class named 'Player'. These sub classes are comprised of 7 AI players, and 1 human player class. The development of the majority of these AI players is based on the research detailed in section 2 of this report.

4.3.1 Player Super Class

Within this class, all of the methods that can be used by any of the sub-classes are defined here. This includes methods for playing a move, getting the player's colour, and ranking the human player's moves (this will be used in the challenging players. While these methods are defined here, they are not implemented; This is instead completed in the relevant sub-classes, as the functionality should change from class to class.

4.3.2 The Human Player Class

The human player class is the only sub class of player that is not an AI; this makes it far more simple than all of the other classes. This makes playMove() class very short since the move has already been selected by the player.

As described in section 4.2 this class is called from the GUI with the number of the tile that was selected and the game object as its parameters. The method flipTiles() within the 'Game' class is then called. If this returns an ArrayList of size greater than one, a setter method is called within the 'Game' class to update the game's state.

4.3.2 The Random Player Class

The random player class is very similar to the human player class. The only difference here is that the position of the move is not passed in as a parameter, but instead generated randomly.

This is done by making use of the findPossibleMoves() method within the game class to get an ArrayList of all the moves this AI player can choose from. The library 'java.util.random' is imported and used to select a random one of these moves.

Once the move has been selected, the game's state is updated in the same fashion as the human player class.

4.3.3 The Killer Player Class

The 'KillerPlayer' class is named as such because it is designed to play optimally. This has been done through the use of the minimax with alpha-beta pruning algorithm. This required two new methods to be added that weren't apparent in the super class; These new methods are max and min.

The optimal move is determined by calling the max method on the current state of the game. Again this is done from within the playMove() method. The extract below shows how this call is made.

```
12  @ public void playMove(Game game){  
13      String gameState = game.getGameState();  
14  
15      double alpha = -100;  
16      double beta = 100;  
17  
18      double move = max(alpha, beta, game, color, gameState, depth: 0);
```

Figure 27: Minimax playMove() Method

On line 18 it is shown that max takes in six parameters, these being:

- I. Alpha - Used to represent the best value that the maximise function can currently guarantee at that level.
- II. Beta - Used to represent the best value that the minimise function can currently guarantee at that level
- III. Game – the game object
- IV. Color – This player's colour
- V. GameState - The position of the game when the function is called
- VI. Depth – How many layers deep the algorithm has already searched

When the max function is called, the current node is checked to see if it has reached a terminal state or its maximum depth. If this is the case the utility function within the 'Game' class is used to calculate, and then return the score of the node. This code can be seen in the extract below:

```

35      //If there is no more moves return the number of black tiles
36      if (depth == 6 || (game.findPossibleMoves(color).size() == 0)){
37          return game.get_weighted_score(this.color, type: "Killer");
38      }

```

Figure 28: Minimax Terminal State Check

If it is decided that this node isn't a terminal state, a new ArrayList of strings is created called 'nextStates'. This contains all of the game's states after each of the player's possible moves. These are then used when calling the min method on the next layer of nodes.

The rest of the max function is shown below, this is based directly on the pseudocode that was shown in Figure 4 in section 2.2.

```

52      double v = alpha;
53
54      if (color == 'X'){
55          color = 'O';
56      } else{
57          color = 'X';
58      } //Change color
59
60      for (int n = 0; n < nextStates.size(); n++){
61          double vprime = min(alpha, beta, game, color, nextStates.get(n), depth: depth+1);
62          if (vprime > v){
63              v = vprime;
64              bestMove = n;
65          }
66          if(vprime >= beta){
67              if (depth == 0){
68                  return bestMove;
69              }
70              return v;
71          }
72          if (vprime > alpha){
73              alpha = vprime;
74          }
75      }
76      if (depth == 0){
77          return bestMove;
78      }
79      return v;

```

Figure 29: Minimax Max Method

The key things to note here are: firstly lines 54 to 58 are responsible for changing the colour of the player before calling the min function. Secondly, on lines 67 and 76 there is an if statement to check if the depth is equal to zero. This is to ensure that if the max function is operating on the first layer of the search tree, the index of the best node in the 'nextStates' ArrayList is returned, instead of the best score being returned – as would normally happen. Getting the index of the next state makes it easier to call the flip tiles method, as the index can be used on the 'possibleMoves' ArrayList, as it is the same size as the 'nextStates' ArrayList. This can be seen in the extract below which is used to update the game's state after the best move has been found:

```

18      double move = max(alpha, beta, game, color, gameState, depth: 0);
19      game.resetGameState(gameState);
20
21      ArrayList<Integer> possibleMoves = new ArrayList<>(game.findPossibleMoves(color));
22      ArrayList<Integer> tilesToFlip = new ArrayList<>(game.flipTiles(possibleMoves.get((int) move), color));
23
24      if (tilesToFlip.size() > 0) {
25          game.setGameState(tilesToFlip, color);
26      }

```

Figure 30: Minimax Selecting Move

The min method works in almost the same fashion. But since max is always called first for these players, it is not necessary to check if the depth is equal to zero before returning the score. This means the min function is a carbon copy of the pseudocode shown earlier in Figure 4. Here it is shown below:

```

111      for (int n = 0; n < nextStates.size(); n++){
112          double vprime = max(alpha, beta, game, color, nextStates.get(n), depth: depth+1);
113          if (vprime < v){
114              v = vprime;
115          }
116          if(vprime <= alpha){
117              return v;
118          }
119          if (vprime < beta){
120              beta = vprime;
121          }
122      }
123      return v;
124  }

```

Figure 31: Minimax Min Method

4.3.4 The Basic Monte Carlo Player Class

This AI is a precursor to the fully functional Monte-Carlo Tree Search player. In this basic version of the Monte Carlo tree search, rollouts are performed on all of the next possible moves for the AI player. The move with the highest score returned from these rollouts is played.

Each rollout is run 500 times, this means that the AI doesn't play particularly quickly, but should lead to generally better moves. The rollouts themselves work off of the same algorithm used for the random player that was discussed earlier. This random algorithm is run, switching between each player's every move until there are no more moves left to play.

When the game is over, if the AI player wins a score of +1 is returned, if it loses -1 is returned, and 0 is returned for a draw. All 500 of these scores are added together to give an estimate of how likely playing that move will lead to a win. The highest of these scores is selected and the respective move is played.

4.3.5 The Monte-Carlo Tree Search Player Class

The full MCTS algorithm is far more complex in its implementation than the basic Monte Carlo algorithm. But the rollout functionality between these two algorithms functions in the same way, meaning that there was a foundation to build from when creating this player class.

Three new private variables were needed to implement this algorithm, these being an ArrayList of nodes, an ArrayList of game states, and the best move. This was done to improve the efficiency of the algorithm by preserving the search tree in-between turns. The tree is preserved until it becomes too large to traverse in a reasonable amount of time, it is then reset and started from fresh.

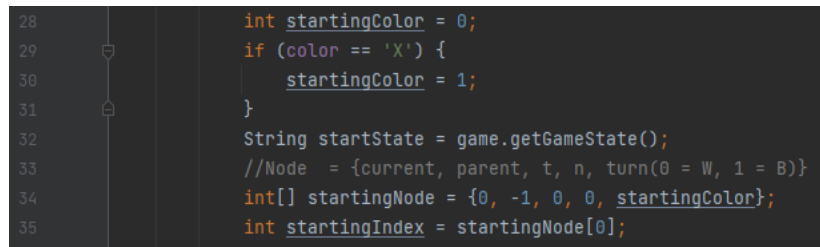
```
private final ArrayList<int[]> nodes = new ArrayList<>();  
private final ArrayList<String> states = new ArrayList<>();  
private int bestMove = 0;
```

Figure 32: MCTS Variable Declaration

A node is stored in the format of an array of five ints. These five values represent the following information about each node:

Node = {current node, parent node, V, n, turn (black = 0, white = 1)}

The root node of the search tree is then initialised with its parent node being set to -1. This is used to stop backpropagation when the root node is reached.



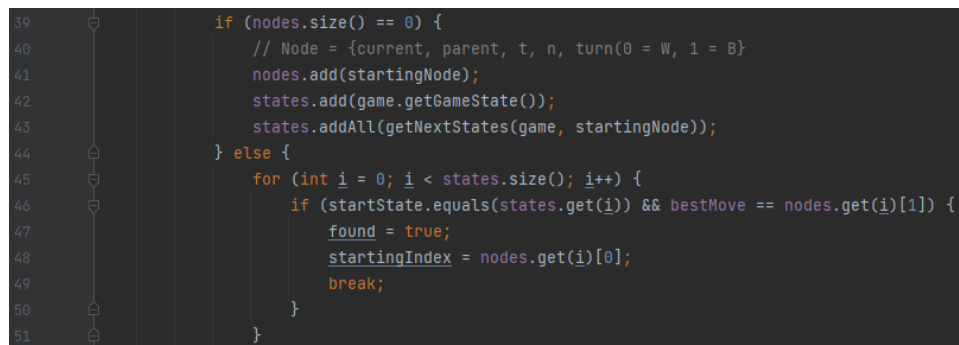
```

28     int startingColor = 0;
29     if (color == 'X') {
30         startingColor = 1;
31     }
32     String startState = game.getGameState();
33     //Node = {current, parent, t, n, turn(0 = W, 1 = B)}
34     int[] startingNode = {0, -1, 0, 0, startingColor};
35     int startingIndex = startingNode[0];

```

Figure 33: MCTS Starting Node Creation

A check is then done to see if the search tree already contains information, or if it has been reset and must be started from scratch. If the size of the tree is equal to zero then the starting node that was created in Figure 33 is added to the ArrayList of all nodes. If the tree size is not zero, the tree is traversed until the current position of the game is found and the starting index is updated.



```

39     if (nodes.size() == 0) {
40         // Node = {current, parent, t, n, turn(0 = W, 1 = B)}
41         nodes.add(startingNode);
42         states.add(game.getGameState());
43         states.addAll(getNextStates(game, startingNode));
44     } else {
45         for (int i = 0; i < states.size(); i++) {
46             if (startState.equals(states.get(i)) && bestMove == nodes.get(i)[1]) {
47                 found = true;
48                 startingIndex = nodes.get(i)[0];
49                 break;
50             }
51         }

```

Figure 34: Setting the Starting Node

If the current position of the game is not found within the tree. The 'nodes' and 'states' ArrayLists are cleared and the tree is restarted from the current game position.

Next, the algorithm itself is run for 500 iterations. This is the implementation of Figure 7 as shown in section 2.3 of the background research. Many helper functions were created to help with the readability and modularity of the code. The first of these being `isLeafNode()`. This function will take in a node as a parameter and search the tree to determine if it is a leaf node. It will then return true if it is and false if not.

The `isLeafNode()` method is used to run a while loop on the condition that the current node is not a leaf node. This loop will update the current node to be the child of the current node with the highest UCB1 score (lowest if the node's position is the opponent's turn). This process is repeated until a leaf node is reached.

The UCB1 score is calculated using another helper function which takes a node as its only parameter and returns the score as a double. This function implements the equation shown in figure []:

```

186  @ private double getUCB1(int[] current){
187      if(current[3] == 0) //If it is the first time the node is visited
188          if (current[4] == 0) { //If it is blacks turn
189              return -100000;
190          }else {                //If it is whites turn
191              return 100000;
192          }
193      int i = current[2] / current[3];    //Average Value of the node
194      if(current[4] == 1){
195          return i - 2 * (Math.sqrt(Math.log(getN(current))/current[3]));
196      } else{
197          return i + 2 * (Math.sqrt(Math.log(getN(current))/current[3]));
198      }
199  }

```

Figure 35: UCB1 Calculation Method

Within this 'getUCB1' function, yet another helper function is used called `getN()`. This function returns the n value of the parent node of the node it is called on.

Once the leaf node has been found, it is checked to see if it has been visited before. If the node has never been visited a rollout is performed. This is done using the same method used for rollouts in section 4.3.4 for the basic Monte Carlo algorithm. Backpropagation is then completed, passing the rollout value back up the tree until the starting node is reached. This can be seen below:

```

108 //rollout
109 nodes.set(current[0], new int[]{current[0], current[1], rollout(game, current[4]), 1, current[4]});
110
111 //back propagate
112 if (current[0] != startingIndex) { // If current node is not the starting node
113     int[] parent = nodes.get(current[1]); // Parent = parent node of current node
114     while (parent[0] != startingIndex) { // Loop until starting node is reached
115         nodes.set(parent[0], new int[]{parent[0], parent[1], (parent[2] + current[2]), parent[3] + 1, parent[4]});
116         parent = nodes.get(parent[1]);
117     }
118     nodes.set(parent[0], new int[]{parent[0], parent[1], (parent[2] + current[2]), parent[3] + 1, parent[4]});
119 }

```

Figure 36: MCTS Rollout and Back Propagation

If however, the leaf node has been visited before, all of its child nodes are added to the search tree. This rollout and backpropagation process shown above is then performed on the first new node that was added.

Once this process is repeated 500 times (this number can be changed to any size desired) the best move is selected based on the node with the highest V value.

4.3.6 The Passive Player Class

The passive player functions in the same way that the minimax with alpha-beta pruning algorithm works, the only difference here is the value returned when a terminal state is reached. This is shown in the below extract:

```

41 //If there is no more moves return the number of black tiles
42 if (depth == 6 || (game.findPossibleMoves(color).size() == 0)){
43     return 50 - Math.abs(50 - game.get_weighted_score(this.color, type: "Passive"));
44 }

```

Figure 37: Passive Player Terminal State Check

Since this player is designed to try and keep the score as close to 50 as possible, the utility score is subtracted from 50, and then the absolute number of this is calculated. Now the value that is the smallest is the best possible move, but since this is a minmax function, the highest number should be the best move. To account for this, the number is subtracted from 50 – the number chosen here is arbitrary, as long as it is consistent throughout.

4.3.7 The Challenging Player 1 Class

The first challenging class that was developed was based on the minimax with an alpha-beta pruning algorithm and the passive player algorithm. This was due to the minimax algorithm having the best performance out of the optimal AI developed, and it being easier to implement, adapt and understand than MCTS.

The min and max functions operate in the same way as they did in the killer player, however, instead of trying to keep the score as close to 50 as possible, the AI tries to match the opponent's skill level based on the previous moves the opponent has made. This requires the use of a new method called `rankOpponentsMove()`.

Since this algorithm requires looking at the opponent's previous move, some new private variables have to be created to store this information between turns. These can be shown below:

```
private final char opponentsColor;  
private final ArrayList<Double> whitesMoves = new ArrayList<>(); //Used to rank all of opponent's moves  
private final ArrayList<String> statesAfterMoves = new ArrayList<>(); //Used to determine what move opponent played  
private final ArrayList<Double> allMoveScores = new ArrayList<>(); //Used to save all of opponent's moves. To monitor how well they are playing  
private int numberOfMoves; //Used to count how many moves have been played
```

Figure 38: Challenging Player 1 Variable Declaration

As the AI requires previous moves to have been made before it can decide on its move, the AI has been designed to always play second. And since the opponent's moves will be ranked after the AI has played its move. The ranking for the first set of the opponent's moves must be predetermined in the constructor. Luckily, the first four available moves all rank the same, this means they can all be given the same score of any value. These scores will all be set to 100 and added to the 'opponentsMoves' ArrayList. The next four game states will then be added to the 'statesAfterMoves' ArrayList. Here the constructor is shown below:

```

16 public ChallengingPlayer1(char color){
17     this.color = color;
18     if(color == '0'){
19         opponentsColor = 'X';
20     }else{
21         opponentsColor = '0';
22     }
23
24     while(whitesMoves.size() < 4){
25         whitesMoves.add(100.0);
26     }
27     statesAfterMoves.add("-----000-----0X-----");
28     statesAfterMoves.add("-----X0-----000-----");
29     statesAfterMoves.add("-----X0-----00-----0-----");
30     statesAfterMoves.add("-----0-----00-----0X-----");
31
32     numberOfMoves = 0;
33 }

```

Figure 39: Challenging Player 1 Constructor

The playMove() method then required updating. Instead of calling the max function on the current game state to calculate the best move, the min function is called on all of the next potential states. This essentially does the same thing as calling max on the first state. But by calling each min that would normally be called from within the max function individually, the rating for each potential move can be saved. Therefore, instead of max automatically picking the highest valued move, the move can be selected based on what the opponent has played. The scores returned from the min functions are then added to an ArrayList and ordered from lowest to highest inside a new ArrayList. This is shown below:

```

105 ArrayList<Double> scores = new ArrayList<>();
106 for (String s : nextStates) {
107     scores.add(min( alpha: -100, beta: 100, game, opponentsColor, s, depth: 1));
108 }
109
110 ArrayList<Double> scoresOrdered = new ArrayList<Double>(scores);
111 Collections.sort(scoresOrdered);

```

Figure 40: Challenging Player 1 Score Ranking

Next, the average rating from the opposing players' past three moves is calculated. A moves rating is based on how well the move ranks among the other available options. For example, if the opponent had four available moves and they chose to play the second best, that moves score would be 0.75 (3/4). This value is stored as a double called 'relativeValue'. Once it has been calculated, it is then used to determine the score of the AI's move that rating is closest to that of the opponents. This process is shown in the figure below:

```

114         double relativeValue = 0;
115         if(numberOfMoves < 3){
116             for (int i = 0; i < numberOfMoves; i++){
117                 relativeValue = relativeValue + allMoveScores.get(i);
118             }
119             relativeValue = relativeValue/(double)numberOfMoves;
120         }else{
121             for (int i = numberOfMoves-3; i < numberOfMoves; i++){
122                 relativeValue = relativeValue + allMoveScores.get(i);
123             }
124             relativeValue = relativeValue/(double)3;
125         }
126
127         Double relativeScore = scoresOrdered.get((int)( relativeValue*((double)scoresOrdered.size()-1)));

```

Figure 41: Challenging Player 1 Relative Value Calculation

Line 127 above shows how the score is determined by multiplying the opponent's average rating by the number of available moves that the AI can play and using this number to find the relative score within the ordered ArrayList.

When the value of the move that will be played is determined, the corresponding value in the unordered scores list is found. The position of the score in the ArrayList is then used as the move that the AI will play.

```

129         int move = 0;
130         for (int i = 0; i < scores.size(); i++) {
131             if (scores.get(i) == relativeScore) {
132                 move = i;
133                 break;
134             }
135         }
136
137         game.resetGameState(gameState);
138         ArrayList<Integer> tilesToFlip = new ArrayList<>(game.flipTiles(possibleMoves.get(move), color));
139
140         if (tilesToFlip.size() > 0) {
141             game.setGameState(tilesToFlip, color);
142         }

```

Figure 42: Challenging Player 1 Move Selection

4.3.8 The Challenging Player 2 Class

The second version of the challenging player works in a very similar way to the first version. The difference here is that this AI will check to see if the opponent's average rating for the last three moves has increased or decreased since the previous move. If the opponent's performance improves or decreases, so too should the AI's moves.

A new private variable is required to store the score of the last average of three moves. This is a double called 'lastAverage' and is initialised to zero within the constructor.

The rest of the code is largely the same as the previous challenging player. The only difference comes in when the relative score for the AI's move is selected. If the opponent's average score for the last three moves is better than their average score on the last turn, then the AI will play a move one ranking higher than the AI's average score. If the opponent's average score for the last three moves decreased since the previous turn, the AI will play a move one ranking lower than the AI's average score.

```

124         if(relativeValue > lastAverage){
125             if((int)( relativeValue*(double)scoresOrdered.size() + 0.5) == scoresOrdered.size()) {
126                 relativeScore = scoresOrdered.get(((int) (relativeValue * (double) scoresOrdered.size() + 0.5))-1);
127             } else{
128                 relativeScore = scoresOrdered.get(((int) (relativeValue * (double) scoresOrdered.size() + 1.5))-1);
129             }
130         }else{
131             if((int)( relativeValue*(double)scoresOrdered.size()) == 0){
132                 relativeScore = scoresOrdered.get((int)( relativeValue*(double)scoresOrdered.size()));
133             } else{
134                 relativeScore = scoresOrdered.get((int)( relativeValue*(double)scoresOrdered.size() - 1));
135             }
136         }

```

Figure 43: Challenging Player 2 Relative Value Calculation

Lines 124 – 129 above show the scenario if the opponent's average increases. Line 125 is a check to see if the relative score would be the best possible move for the AI, if this is the case a better move cannot be selected as this would lead to an out-of-bounds error. The test is done by adding 0.5 to the result of the calculation before casting to type int; this ensures that the number is rounded to the nearest whole number. If this value is the same as the number of moves available, then it cannot

be increased and therefore is played on line 126. If the number does not exceed the number of available moves, the number is increased so a better move is played on line 128. (Note that 1 is subtracted on lines 126 and 128 before getting the score from the ArrayList – this is to avoid out-of-bounds errors).

A similar process is repeated on lines 130 – 135. These lines are for when that opponent's average score decreases. The nested if statements in the section ensure the AI's move is only decreased if it will not cause an out-of-bounds error.

4.4 Verification & Validation

JUnit testing was completed on the functions within the game state and player class. Doing this confirmed that all of the correct tiles were flipped when a move was played, the game state was updated correctly, the game found the correct possible moves, illegal moves weren't played and it found the correct winner when the game was over. All JUnit tests can be shown in Appendix B

As the AI were harder to create JUnit tests for, simulations were run of the AI playing against each other to evaluate their effectiveness. The full results from this can be found in Appendix A.

5 Results & Evaluation

5.1 Evaluation Process

The evaluation process involved participants with a wide range of familiarity with the game Othello playing and evaluating the performance of the challenging AI. The results from this evaluation have been analysed and used to determine whether an AI that can challenge, but not discourage the player has been successfully created.

The testing involved asking each participant to play one game against two separate AI players. The first was the minimax, as it was decided that this was the best AI that had been developed. The second AI was the challenging player 2. The hope is that when the user plays against the minimax player, it will provide a common experience of an AI creating an unfair and unenjoyable game. But when the user plays against the newly developed AI, the game should still be challenging, but not so much as to discourage the player. The participants were then asked to play as many more games as they would like against any player they would like.

Once the participants have played all the games they desire, they were asked to fill out two surveys to evaluate the two AI players (one survey per AI). These surveys were split into two main sections, the first evaluated how the user thought the AI performed – if it was too difficult/too easy. The second half evaluated the user's experience playing the AI – whether or not they enjoyed their games. Feedback was recorded using a scale from 1 (strongly disagree) to 5 (strongly agree) for every question. Some other additional information was gathered about the users as well to help with the evaluation. This information is how much experience they have playing Othello, how many games they played and if they had any other comments.

To prevent biases from forming before the users played. They were not told that they were playing an optimal AI and a challenging AI. Instead, these were presented to the user as player A and player B. These letters represent the minimax player and the challenging 2 players respectively.

5.2 Results of Evaluation

As players were given the option to play either of the AI players multiple times, the first question asked was how many times they played that AI. Below are shown the number of times each AI player was played:

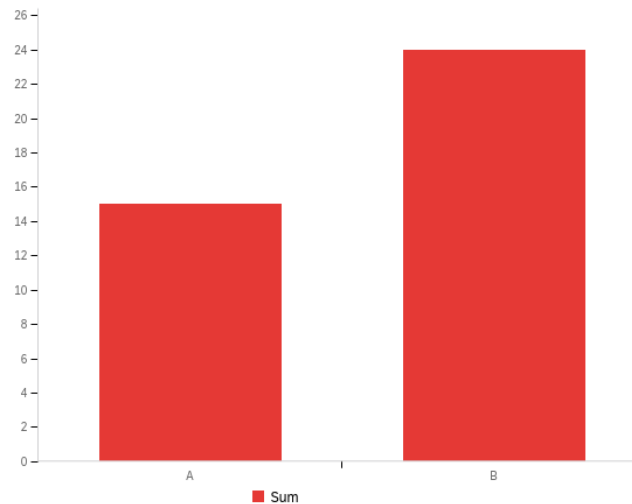


Figure 44: Number of Games Played

From initial viewing of the first question. It appears that the challenging AI was by far the more popular of the two with almost double the number of games played. This figure alone implies that the challenging AI is more enjoyable than the standardised min-max.

Next are shown the average results from the user's evaluation of each AI's performance ranking from 1 being strongly disagreed to 5 being strongly agreed:

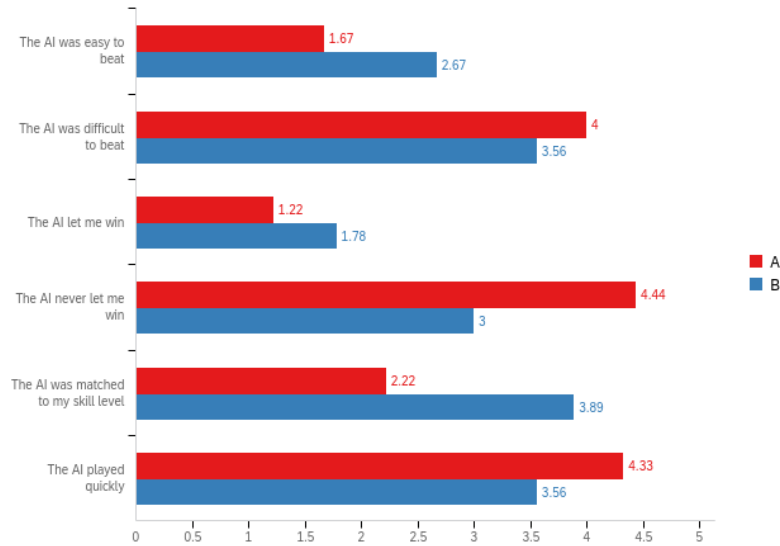


Figure 45: Participant AI Evaluations

From this data, it is clear to see that the minimax player performs optimally. It has been recorded that on average, players found it to be very difficult to beat. Users felt like it would not let them win, and the level that it played at was not fairly matched to their skill level.

The responses for the AI player, on the other hand, show that users still generally found the player to be difficult to beat, but not as difficult as the optimal player. However on average participants generally agreed that this player was matched to their skill level.

The final set of questions that are shown below was designed to evaluate the user's experience playing each AI. Again the results from each question ranged from 1 being strongly disagreed to 5 being strongly agreed:

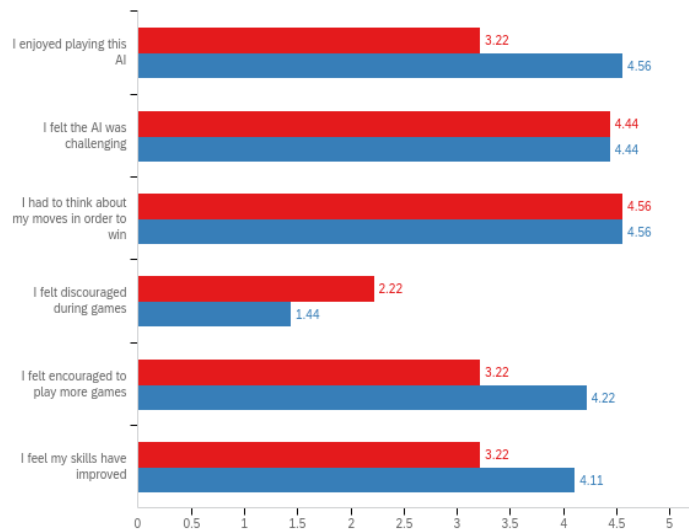


Figure 46: Participant Experience Evaluation

These results show that on average participants agreed-strongly agreed that playing against both AI was challenging and required them to think about their moves to win. The difference in opinions comes when looking at how much the users enjoyed their experience. When asked if they enjoyed their games and felt encouraged to play more against player A, the average response was 'neither agree nor disagree'. However, when asked these questions about player B, the average response was in-between agree and strongly agree. Users also felt their skills improved more by playing against player B than A.

5.3 Returning to the Research Questions

This project aimed to create an AI that could play a two-player game that would put the user in challenging situations, that require strategic and thoughtful moves to win. As this AI should adapt to any opponent testing was performed on a set of participants with varying degrees of

familiarity with the game to ensure that the game wasn't biased toward any particular skill level. Participants' skill levels are shown below (note participants filled out the survey twice, so the numbers are doubled):

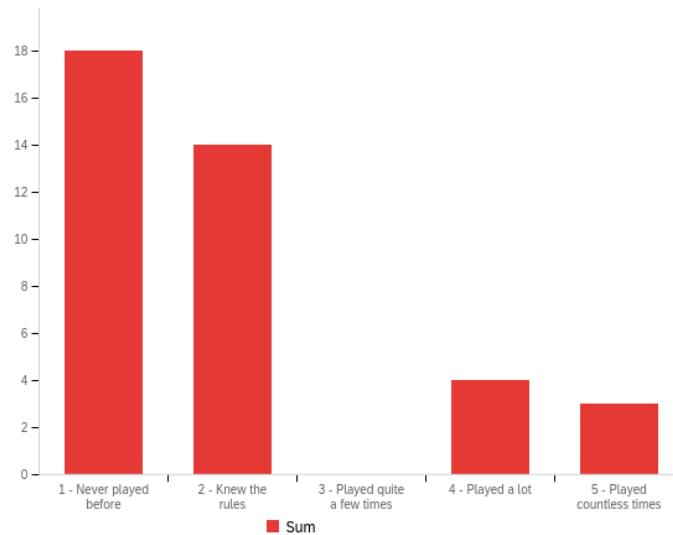


Figure 47: Participants' Familiarity with Othello

As seen in Figure 45 users found the performance of player B to be less dominant than the optimal AI player but still reported the game to be challenging. It was agreed that participants had to think equally as hard about their moves if they wanted to play well against both players. This information shows that the analysis performed by the AI player to determine the level of the human player's moves was successful. The AI still typically selected good moves based on this information, but did not play optimally.

This in turn led to users having a more enjoyable time while playing against this AI when compared to the optimal player. This statistic can also be supported by the results in Figure 46 saying users felt less discouraged when playing the challenging AI and were encouraged to play more games. This was likely due to the AI taking into account the average rating of the human players' last few moves. The AI could predict when the user was starting to struggle and adjust its moves accordingly.

The challenging AI was also on average reported to have helped the user's skill level improve slightly more than it did while playing the optimal AI. There is a strong chance that the reason for this is the

same as before. Since the AI was not dominating the users, they felt their moves had more of an effect and therefore required more thought. This would of course lead to a better understanding of the game.

6 Discussion & Reflection

6.1 Interpreting the Results

As discussed in section 5.3, the consensus was that the challenging AI gave the user an enjoyable, but still challenging experience. This seems to be mostly due to the AI being matched to each user's skill level.

Going forward similar techniques as those used in this project for determining the opponent's skill could be researched further. This type of AI player could be reused and adapted for many different two-player games. This would lead to better, more personalised and enjoyable games against AI players, and would give users a far better alternative to common, well-researched optimal AI.

6.2 Reflection

Upon reflection, less time should have been spent developing the Monte Carlo Tree Search AI. While it was an interesting challenge to implement and compare against other optimal AI players. Ultimately it did not prove useful to the overall aims and objectives of the project.

The time spent on developing this AI could have been better spent on the development and testing of the main goal of the project – this being the challenging AI player. More time allocated to testing would have allowed for a more detailed analysis of the challenging player, which could have possibly led to its design being improved upon.

6.3 Challenges

One of the main difficulties during the development of this project was testing and debugging all of the AI players that were developed. As the game of Othello using some of the more complex AI can

takeover 10 minutes to complete, and it was sometimes difficult to determine if they were playing as they should or where they were going wrong.

If by the end of the games, it was clear the AI agents were not performing as planned there were then a lot of factors that could be effecting the output. This involved extensively debugging the code by keeping track of all variable values and method outputs until errors were found. This process took a long time as there were often multiple bugs to be found, making the process a lot harder.

6.4 Limitations

The accuracy of the results of this project may be affected by the limited number of participants with medium to high skill levels in the game. 9 participants were involved in testing with, 7 of them having little to no knowledge of Othello beforehand. This was due to the difficulty of finding participants that met these requirements.

While this helped with the understanding of how well the challenging AI played against unskilled players, the results may not be entirely accurate when playing medium-skilled to advanced players. More testing would need to be done to conclusively prove this AI played well at all skill levels.

6.5 Future Work

While this research seems to provide a solid version of a challenging player, further improvements could be implemented to try and improve upon it.

Firstly, the AI could experiment with a different implementation of how it tries to match the user's moves. This implementation in this project focuses on how good the move selected by the human player is compared to their other moves. However, a different implementation could be created based on Eve Rigley's work last year. In Rigley's 'Passive Player,' the AI tries to play the move that tries to balance the utility score to fifty percent; This could potentially be adapted so that the AI tries to play the move that most closely matches the utility score of the player's move.

Secondly, the number of moves consider to gauge how well the player is performing could be altered. For this AI, the human player's previous three moves were recorded and monitored to see if they were playing better or worse throughout the game. This number could be increased or decreased to find the optimal value.

7 Conclusion

In conclusion, this project successfully developed a challenging AI player for the board game Othello. After thorough research into different AI agents, the minimax algorithm was able to be adapted to create an AI that was able to play intelligent moves, while still producing an encouraging and enjoyable game for the user. However, the reliability of the results may be limited by the small number of participants that had experience with the game before testing. Future research could explore various implementations of the move selection process and different move analysis methods. By experimenting with these techniques, it may be possible to discover a more effective way of mapping the AI's moves to the human player's moves.

Overall this project provided solid research into the development of challenging AI players and has created a strong foundation for future experimentation in this area.

8 References

- [1] "Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning) - GeeksforGeeks," 30 March 2023. [Online]. Available: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>. [Accessed 30 March 2023].
- [2] A. Choudhary, "Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo, Analytics Vidhya," 30 March 2023. [Online]. Available: <https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/>. [Accessed 30 March 2023].
- [3] E. Rigley, "An AI That Plays Fairly: The Encouraging Teacher," 2022.
- [4] "World Othello Federation - Official Rules for Othello," 30 March 2023. [Online]. Available: <https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english> . [Accessed 30 March 2023].
- [5] M. Eppes, "How a Computerized Chess Opponent "Thinks"—The Minimax Algorithm," 20 March 2023. [Online]. Available: How a Computerized Chess Opponent "Thinks"—The Minimax Algorithm (2019). Available at: <https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1> (Accessed: 30 March 2023).. [Accessed 20 March 2023].
- [6] "Artificial Intelligence | Alpha-Beta Pruning - Javatpoint," 23 March 2023. [Online]. Available: <https://www.javatpoint.com/ai-alpha-beta-pruning>. [Accessed 23 March 2023].
- [7] D. J. Levine, *University of Strathclyde CS310 Lecture Slides*, 2022.

- [8] "Artificial Intelligence | Mini-Max Algorithm - Javatpoint," 23 March 2023. [Online]. Available: <https://www.javatpoint.com/mini-max-algorithm-in-ai>. [Accessed 23 March 2023].

9 Appendix

Appendix A – AI simulation results

Game	Monte Carlo Player	Minimax Player
1	19	45
2	5	59
3	13	51
4	17	47
5	17	47
6	8	56
7	3	59
8	19	45
9	12	52
10	13	51
11	12	52
12	10	54
13	10	54
14	17	46
15	8	56
16	12	52
17	3	60
18	1	57
19	5	59
20	3	61

Game	Monte Carlo Player	Passive Player
1	42	22
2	27	37
3	27	37
4	32	32
5	33	31
6	33	31
7	35	29
8	14	50
9	29	35
10	30	34
11	38	26
12	29	35
13	34	30
14	29	35
15	33	31
16	36	28
17	26	38
18	41	23
19	21	43
20	42	22

Game	Random	Minimax
1	18	46
2	4	60
3	7	56
4	20	43
5	0	58
6	5	59
7	10	54
8	1	62
9	2	60
10	2	62
11	9	55
12	23	41
13	6	58
14	5	59
15	2	61
16	9	55
17	17	47
18	17	47
19	16	48
20	7	57

Game	Random	Challenging 1
1	27	37
2	50	14
3	13	51
4	25	39
5	29	35
6	14	50
7	42	22
8	28	36
9	30	34
10	27	37
11	24	40
12	46	18
13	17	47
14	36	28
15	34	30
16	42	22
17	52	12
18	30	34
19	38	26
20	29	35

Game	Random	Challenging 2
1	40	24
2	9	55
3	18	46
4	14	50
5	27	37
6	40	24
7	48	16
8	38	26
9	25	39
10	41	23
11	37	27
12	24	40
13	23	41
14	22	42
15	28	36
16	4	59
17	51	13
18	21	43
19	33	31
20	20	44

```

23 @Test
24 public void testFlipTiles() {
25     Game game = new Game();
26     ArrayList<Integer> expectedTiles = new ArrayList<>(List.of(27, 28));
27     assertEquals(expectedTiles, game.flipTiles( position: 27, color: '0'));
28
29     String expectedGameState = "-----000-----0X-----";
30     game.setGameState(game.flipTiles( position: 27, color: '0'), color: '0');
31     assertEquals(expectedGameState, game.getGameState());
32
33     expectedTiles = new ArrayList<>(List.of(35, 36));
34     assertEquals(expectedTiles, game.flipTiles( position: 35, color: 'X'));
35
36     expectedGameState = "-----000-----XXX-----";
37     game.setGameState(game.flipTiles( position: 35, color: 'X'), color: 'X');
38     assertEquals(expectedGameState, game.getGameState());
39
40
41     expectedTiles = new ArrayList<>(List.of(43, 36, 35));
42     assertEquals(expectedTiles, game.flipTiles( position: 43, color: '0'));
43
44     expectedGameState = "-----000-----0X-----0-----";
45     game.setGameState(game.flipTiles( position: 43, color: '0'), color: '0');
46     assertEquals(expectedGameState, game.getGameState());
47
48 }

```



```

80 @Test
81 public void testCountTiles() {
82     Game game = new Game();
83     assertEquals( expected: 2, game.countTiles( color: 'X') );
84     assertEquals( expected: 2, game.countTiles( color: 'O') );
85
86     game.resetGameState("000000000000000000000000000000000000000000000000000000000000XX");
87     assertEquals( expected: 2, game.countTiles( color: 'X') );
88     assertEquals( expected: 62, game.countTiles( color: 'O') );
89
90     game.resetGameState("0000000000000000X0000000X000X0XX0000XXX0000000000000000000000XX");
91     assertEquals( expected: 10, game.countTiles( color: 'X') );
92     assertEquals( expected: 54, game.countTiles( color: 'O') );
93 }
94
95 @Test
96 public void testPlayMove(){
97     Game game = new Game();
98     Player player = new HumanPlayer( color: 'O' );
99
100     String expectedGameState = "-----000-----0X-----";
101     player.playMove(game, final: 26);
102     assertEquals(expectedGameState, game.getGameState());
103
104     expectedGameState = "-----000-----000-----";
105     player.playMove(game, final: 37);
106     assertEquals(expectedGameState, game.getGameState());
107
108     player.playMove(game, final: 2);
109     assertEquals(expectedGameState, game.getGameState());
110
111     player = new HumanPlayer( color: 'X' );
112     game.resetGameState("X-----0-----0-----X-----");
113     expectedGameState = "X-----X-----X-----X-----X-----";
114     player.playMove(game, final: 18);
115     assertEquals(expectedGameState, game.getGameState());
116
117 }
118 }

```

```


50 @Test
51 public void testFindPossibleMoves() {
52     Game game = new Game();
53     ArrayList<Integer> expectedMoves = new ArrayList<>(List.of(20, 27, 38, 45));
54     assertEquals(expectedMoves, game.findPossibleMoves(color: 'O'));
55
56     game.resetGameState("--XXX-----OOO-----");
57     expectedMoves = new ArrayList<>(List.of(18, 19, 20, 21, 22));
58     assertEquals(expectedMoves, game.findPossibleMoves(color: 'X'));
59
60     game.resetGameState("-----");
61     expectedMoves = new ArrayList<>();
62     assertEquals(expectedMoves, game.findPossibleMoves(color: 'X'));
63 }
64
65 @Test
66 public void testFindWinner() {
67     Game game = new Game();
68     assertEquals(expected: "It's a Draw", game.findWinner());
69
70     game.resetGameState("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
71     assertEquals(expected: "White Wins!", game.findWinner());
72
73     game.resetGameState("OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOX");
74     assertEquals(expected: "Black Wins!", game.findWinner());
75
76     game.resetGameState("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
77     assertEquals(expected: "It's a Draw", game.findWinner());
78 }

```

✓	Test Results	38 ms
✓	JUnitTesting	38 ms
✓	testFlipTiles()	32 ms
✓	testCountTiles()	1 ms
✓	testPlayMove()	1 ms
✓	testFindWinner()	
✓	testResetGameState()	1 ms
✓	testFindPossibleMoves()	2 ms
✓	testGetGameState()	1 ms

9.1 Ethical Approval Form

30/03/2023, 20:17
CIS Ethics Approval System – Computer and Information Sciences – local.cis



Computer and Information Sciences - local.cis
departmental information for staff and students

[Home](#)
[Ethics](#)
[Events](#)
[Safety](#)
[Systems Support](#)
[Teaching](#)
[Utilities](#)

[Browse Home / Utilities / CIS Ethics Approval System](#)

CIS Ethics Approval System

You are James Norwood (CS2019 - 201919347)

[Return to Main](#)

Application ID: 2180

Title of research:
AI for games: Making the human think

Summary of research (short overview of the background and aims of this study):
To develop a 'challenging' artificial intelligence (AI) that can play a two player board game. The AI should adapt to whomever it is playing, so that the user feels they have had a challenging game. This means that they should not be discouraged by the AI playing too well, and should not find the game so easy that they win with little effort.

How will participants be recruited?
Recruitment will be completed by contacting friends and family, and asking if they would like to participate.

What will the participants be told about the proposed research study? Either upload or include a copy of the briefing notes issued to participants. In particular this should include details of yourself, the context of the study and an overview of the data that you plan to collect, your supervisor, and contact details for the Departmental Ethics Committee.
[PDF File: View document](#)

How will consent be demonstrated? Either upload or include here a copy of the consent form/instructions issued to participants. It is particularly important that you make the rights of the participants to freely withdraw from the study at any point (if they begin to feel stressed for example), nor feel under any pressure or obligation to complete the study, answer any particular question, or undertake any particular task. Their rights regarding associated data collected should also be made explicit.
[PDF File: None.](#)
Consent will be demonstrated by showing participants the participant information document and the consent for at the start of every survey. And only proceeding to the survey if they agree.

What will participants be expected to do? Either upload or include a copy of the instructions issued to participants along with a copy of or link to the survey, interview script or task description you intend to carry out. Please also confirm (where appropriate) that your supervisor has seen and approved both your planned study and this associated ethics application.
[PDF File: View document](#)
[PDF File: View document](#)
The participation information sheet contains instructions for what the participant must do. The questionnaire is shown as a PDF here, but will be presented to the participants as a URL they can follow.

What data will be collected and how will it be captured and stored? In particular indicate how adherence to the Data Protection Act and the General Data Protection Regulation (GDPR) will be guaranteed and how participant confidentiality will be handled.
The questionnaire is completely anonymous. The only data that will be stored is the results from the survey, which all be stored on the Strathclyde Qualtrics server.

How will the data be processed? (e.g. analysed, reported, visualised, integrated with other data, etc.) Please pay particular attention to describing how personal or sensitive data will be handled and how GDPR regulations will be met.
The data from all surveys will be collated and analysed. The data will be shown in my final report in the form of graphs, tables and text. The information gathered will be used to draw conclusions as to how well the AI have met their goal of being challenging, and to determine any improvements that could be made.

How and when will data be disposed of? Either upload a copy of your data management plan or describe how data will be disposed.
[PDF File: None.](#)
The data will be stored on the Strathclyde Qualtrics server. I will dispose of the questionnaire within one month of receiving my project grade.

<https://local.cis.strath.ac.uk/wp/extras/ethics/index.php?view=2180>
1/2

9.2 Participant Information Sheet

Participant Information Sheet for 'AI for games: Making the human think'

Name of department: Computer and Information Sciences

Title of the study: AI for games: making the human think

Introduction

My name is James Norwood I am 4th year Computer Science undergraduate student at the University of Strathclyde.

What is the purpose of this research?

For my project, I have aimed to create an artificial intelligence (AI) that can give any user a challenging but fun game of Othello. This study aims to evaluate your experience when playing against the AI, to determine how difficult and how enjoyable you found it.

Do you have to take part?

Participation in this study is voluntary. Your decision to either take part in this research or not, will have no affect as to how you are treated. You may also withdraw from this research at any time without detriment.

What will you do in the project?

You will be presented with the option of playing a game of Othello against two different AIs. You will be asked to play each AI at least once. But you will be able to play each AI as many times as you like, if you would like to get a better understanding of how they play. Each game should take roughly 5-10 minutes to complete. After you have completed playing all of your games you will be asked to fill out a questionnaire for each AI. These will ask you questions about your experience of the games you played.

Why have you been invited to take part?

As my project aims to create an artificial intelligence that adapts to any user who plays it, I am looking for participants with varying levels of familiarity with the game. I will need to find how well the AI performs against both players who have never played Othello, players who are very familiar with the game, and players of every skill level in-between.

What information is being collected in the project?

The only information that is being collected is the results from the questionnaire. All of these questions will be about your familiarity with Othello, and your experience of playing the AI. No personal or identifiable data will be used or saved.

Who will have access to the information?

Your role in this study is completely anonymous. The data recorded from the questionnaires will be stored, analysed, and combined with other participants data to draw conclusions about the AI. These conclusions will be presented in this projects final report in the form of graphs, tables and written analysis.

Where will the information be stored and how long will it be kept for?

The data will be stored on the Strathclyde Qualtrics server, and will be disposed of within one month of the project grade being released.

Thank you for reading this information – please ask any questions if you are unsure about what is written here.

What happens next?

If you wish to participate in this study you will first need to sign a consent form. Then I will present the game to you on my personal device. The questionnaire can then be filled out online with a URL that I will provide you with.

If you have any further questions about this study, you can contact me through my email that is provided below.

If you do not wish to participate. Then I thank you for taking the time to read this document.

Researcher contact details:

James Norwood: james.norwood.2019@uni.strath.ac.uk

Chief Investigator details:

John Levine: john.levine@strath.ac.uk

This research was granted ethical approval by the Department of Computer and Information Sciences Ethics Committee.

If you have any questions/concerns, during or after the research, or wish to contact an independent person to whom any questions may be directed or further information may be sought from, please contact: ethics@cis.strath.ac.uk

9.3 Consent Form

Participant Information Sheet

[Participant information sheet](#)

Consent Form

Name of department: Computer and Information Sciences

Title of the study: AI for games: Making the human think

- I confirm that I have read and understood the Participant Information Sheet for the above project and the researcher has answered any queries to my satisfaction.
- I confirm that I have read and understood the Privacy Notice for Participants in Research Projects and understand how my personal information will be used and what will happen to it (i.e. how it will be stored and for how long).
- I understand that my participation is voluntary and that I am free to withdraw from the project at any time, up to the point of completion, without having to give a reason and without any consequences.
- I understand that I can request the withdrawal from the study of some personal information and that whenever possible researchers will comply with my request.
- I understand that anonymised data (i.e. data that do not identify me personally) cannot be withdrawn once they have been included in the study.
- I understand that any information recorded in the research will remain confidential and no information that identifies me will be made publicly available.
- I consent to being a participant in the project.

Click below to acknowledge that you have read the participant information sheet, and that you agree to the terms specified in the consent form above.

If you do not agree you are free to not participate and may leave this website.

☐ I agree