# Creating an AI to play Othello

## Final Report

**Author:** Owen Shevlin

**Supervisor:** Frank Langbein

**Moderator:** Hantao Liu

**Module:** CM3203 – One Semester Individual Project (40 Credits)

**Institution:** School of Computer Science and Informatics, Cardiff University

**Date of Completion:** Friday the 5th of May 2017

## Abstract

This project's objective is to explore the possible ways to create an AI capable of playing the board game Othello. The project discusses the ways in which existing algorithms for game playing - such as Minimax and Monte Carlo Tree Searching - can be adapted to decide moves for an Othello AI to play. This report also covers the process of creating a suitable programmatic Othello game representation for the AI to use, and a GUI for viewing the progress of the game. Additionally, this project also investigates the possible ways to create an evaluation function powered by Deep Learning methods that an AI player can use to accurately determine the worth of a game state. These potential ways of creating the AI are then put into practice, resulting in a system with a customisable AI, an accurate recreation of the Othello board game, and the ability to create neural networks from a database of Othello games.

## Table of Contents

# Section 1 – Introduction

The field of Artificial Intelligence (AI) is currently at the cutting edge of Computer Science. AI systems are part of many of the world's devices and technologies, such as personal assistants, smart cars, content generation systems, and more. [1]

On a more bespoke scale, they can be created to analyse and play board games, by evaluating the possible moves a player can make and choosing the most optimal one. This approach has been used to create AIs capable of playing board games such as Chess, Checkers and Go.

My project is focused on creating an AI program that can play the board game Othello, which is based around placing counters onto a board while flipping your opponent's counters. To achieve this, I have created a programmatic version of Othello, a UI to view the state of the game with, and the necessary AI systems and interfaces, including operations for utilising deep neural networks.

The programmatic representation of Othello was needed so that an AI could accurately choose moves, understand the game rules, and manipulate the game state. As a result, I had to fully understand the rules of Othello before implementation, and extensively test the game logic upon completion of the game classes.

To allow for easy viewing of the game state, it was also necessary to create a graphical user interface. This in turn also enabled human players to input moves to when playing against an AI opponent.

Finally, with the Othello game programmed and the UI in place, I created a suite of artificial intelligence classes which represent different game playing and analysis techniques. These classes can be combined together to create AIs of varying difficulty, from random move selecting players, to highly intelligent opponents with deep learning knowledge.

The result of this work is an Othello program that can be used by humans to train themselves against artificial opponents of varying difficulty levels, via an easy to understand interface. It can simulate games of Othello between two AI players, and includes the ability to create and load neural networks for the most intelligent AIs to use in their evaluation of the game.

Though Othello is a game that has been solved by other solutions before, I believe that the challenge of creating the game and its UI, implementing the AI and its associated technologies, and analysing the performance of the AI has proven to be an engaging and interesting project, especially due to the level of complexity surrounding the topic.

# Section 2 – Background and Research

This section contains all the information I have gathered on the various topics related to my project; I started my research by exploring the ways which humans play and evaluate Othello, followed by examining the ways in which computers understand general game playing. With this information, I combined the two topics, and went on to research more advanced algorithms and learning approaches that would work well for creating an AI capable of interpreting Othello.

## 2.1 – Othello

### 2.1.1 – Basic Ruleset

Othello (sometimes referred to by its old name of Reversi) is a table top board game invented in the 19th century, where players place coloured counters onto an 8 by 8 board with the intention of *flipping* the other counters to their colours. [2] This is done by forming a line between a placed counter and another counter of that colour on the board, which flips all other counters along said line to the colour of the placed counter. [4] This operation of creating the line between two counters of the same colour is also known as *bracketing*. [3]

Play begins with two dark and two light counters placed at the centre four squares of the game board, with each counter being diagonally adjacent to the other counter of its colour. Players take turns placing counters onto the board, with the player using the dark counters going first. Counters may only be played onto the board in positions where an existing placed counter is within a one square radius, and if placing the counter will cause other counters to be flipped. [4]

If there are no places on the board that a player can place a counter on, then their turn is skipped, and the other player gets to play a counter instead. The game ends when neither player can play a counter on their turns, or when the board is full of counters, whichever comes first. [4]

A selection of example Othello game states have been provided in Appendix A.

### 2.1.2 – Advanced Concepts

Though I know the rules of Othello well, my research revealed many high-level concepts and tactics that professional Othello players have discovered over the years. Many of these are documented in the 1987 player's guide for Othello, *Othello: Brief and Basic* by Ted Landau. In it, Landau describes the three phases of an Othello game, which are the "Opening (opening 20 moves), Midgame (middle 20 moves), and Endgame (last 20 moves).", and discusses the various moves and tactics that can be employed in the three phases. [3]

The book also provides 21 key points that allow a player to play Othello with a greater understanding of the potential moves and game states. Some of these points, such as "Planning Ahead" and "Prioritising of Moves", would naturally become part of the AI that I have created, since game states resulting from potential moves can be analysed ahead of time, and moves will be chosen based on the value returned by the AIs evaluation function. [3]

Landau also discusses the ways to intelligently evaluate an Othello game state, in sections such as "Not All Squares Were Created Equal", which contains information about how edge and corner counters in an Othello game are more valuable than the other locations on the board, since they have a reduced number of sides to be surrounded on, and are less likely to be flipped by the opposing player. [3] Rules such as these that provide additional ways of determining the worth of a game state have been built into parts of the final AI classes of the project, though many of these game state traits did not to be explicitly defined for the AI or Deep Learning algorithms to detect them.

There are still many other key points of advanced Othello playing – such as what moves to play in each phase, recognising patterns of counters and how to best play around them, potential traps, etc. – that players should be aware of. The full list of Landau's 21 key points to playing Othello, along with brief descriptions of each one, can be found in Appendix B.

## 2.2 – Adversarial Searching and Game Playing

### 2.2.1 – Game Theory

Game theory is "the study of mathematical models of conflict and cooperation between intelligent rational decision-makers" [9], and can be used by game-playing AI to determine what move to play over others.

Game theory is also used to describe the way games operate, and thus helps to determine how they should be evaluated. For example, Othello would be described as a *sequential* game, since players do not perform actions simultaneously; the possible states and moves of a sequential game can be easily modelled as a decision tree, due to the branching nature of the game states. [9]

Furthermore, Othello is considered as a game with *perfect information*, meaning that all information about the game is visible to all players. [9] In comparison, a game with *imperfect information* has some information that is not always visible to a player. For example, in a game of poker, a player has no way of knowing an opponent's hand until the end of the round, so the information they hold about the game, and by extension their estimation of their chances of winning, is imperfect.

Additionally, Othello is what's known as a *zero-sum game*, which describes a game where the value gained by one player is the value lost for the other player. Knowledge of these kinds of properties and which ones Othello fits is important when it comes to understanding the game itself, and in creating an accurate simulation of the game programmatically.

### 2.2.2 – Minimax

Since Othello is a perfect information game, an Othello-playing AI should be able to predict the future moves that players will play via various search and evaluation algorithms. One such example of a decision algorithm used by AI systems is the Minimax approach.

Minimaxing is a method used when evaluating a game's decision tree to ensure that the move chosen by the player maximises the chance of a favourable outcome for them. A simplified example of this in a game of Othello would be that a minimax evaluation on the game tree would favour a move that increases the player's score by three over a move that would only increase the score by one, due to the larger perceived gain.

However, the minimax algorithm also evaluates the opponent's moves, which assumes that the opponent will always pick the move that minimises the player's gain from a move. As a result, the algorithm reflects the best outcome for the player should they choose that branch of the game tree.

The value returned by a minimax search may only be accurate up to a certain number of moves in the future, as there is a limit on how far an AI can predict, based on the available time and computational power.

Additionally, the success of a minimax algorithm is directly related to the quality of the evaluation function, which is what determines how valuable a move or game state is. As previously mentioned, an Othello evaluation function could be based only on the difference in the two player's scores, resulting in values like +3, +1, -2, etc. However, this fails to consider the multitude of other factors of an Othello game state, such as the number of corner or edge pieces the player has, the manoeuvrability of the player, and more. These factors can increase or decrease the value of the game state, and should be appropriately considered when producing an evaluation function. A competent evaluation function can be the difference between a good AI and a great AI.

The Monte Carlo Tree Search algorithm is essentially a specialised version of the Minimax algorithm, with a focus on moves that have the perceived potential to perform well. As a result, knowledge of the Minimax process will help clarify the operation of other AI searching algorithms.

### 2.2.3 – Pruning

To improve the performance of a minimax or other tree searching algorithm, branches of the tree can be pruned if expanding them would not be beneficial to the outcome of the algorithm.

For example, at a level of the tree where Player B is looking to minimise the value of Player A's next state, and where the two available branches of the tree lead to values of -4 and -2 for Player A, the -2 branch can be pruned. This is because a sensible Player B would never select a move that doesn't minimise the gain for Player A, so the -2 branch does not need to be evaluated.

This type of pruning is called Alpha-Beta Pruning, and can drastically increase the performance of a tree search algorithm that uses it. Many other pruning algorithms exist, but Alpha-Beta is well suited to a minimax-type algorithm. Effective use of pruning will allow an AI to evaluate more games states within the allotted processing time, thus improving the quality of its decisions.

### 2.2.4 – Solving a Game

A solved game is "a game whose outcome (win, lose, or draw) can be correctly predicted from any position, assuming that both players play perfectly". [10] This means that the winner of a game can be determined just by examining the current game state.

The 4x4 and 6x6 variants of Othello have been fully solved by computers that have computed all possible games to determine every possible outcome. However, for the default 8x8 Othello game, there is an estimated $10^{28}$ possible legal positions the game state can be in, making it infeasible to fully solve. [6] However, many Othello programs record the outcomes/evaluations of the first X moves in a structure called the *open book* [6], which allows Othello AI to play optimally during the early moves on the game, based on their experience. This information can be encoded into a game-playing AI to allow it to easily evaluate the starting states of a game, thus improving the number of moves it can analyse.

## 2.3 – Artificial Intelligence

### 2.3.1 – Defining Artificial Intelligence

In the context of this project, an Artificial Intelligence (AI) is a program that can make decisions based on the world around it. For example, in a game of Othello, an AI that can play Othello should be able to make moves in the game based on the data obtainable from analysing the game state, such as the position of counters, the remaining moves, the value of the board, etc. [5]

### 2.3.2 – Examples of Othello-playing AIs

My project isn't the first to explore the possibility of an Othello-playing Artificial Intelligence; numerous Othello learning tools have been created in the past, that also contain an AI program to decide what moves to play. These tools, which include NTest, Saio, Edax and Logistello, have gone on to defeat the best human Othello players in the world, and have even played against each other in some events. [6]

However, these existing solutions do not allow for variance in their artificial intelligence's ability; the AI will always perform optimally, and cannot be altered to allow players of lower skill levels to compete. As I wanted to evaluate the final version of my AI against versions of itself with different parameters, I decided to allow users to alter the difficulty of the AI they play against by allowing users to provide these parameters. This allows my AI to be used as a training tool for players of any skill level, and thus has added a unique feature to my program that isn't seen in other Othello training programs.

Additionally, many existing Othello programs have seen a declining number of updates since their respective releases. For example, NTest was retired in 2005, while Saio and Edax have not been updated since 2011. [6] [7] My Othello AI and the program housing it are much newer than these programs, and I have been able to utilise newer technologies and approaches and that were not available when the other Othello programs were released.

While researching these Othello programs, I also discovered a paper written by J.A.M Nijissen from Maastricht University, which was titled *Playing Othello Using Monte Carlo*, which sounded very similar to what I aim to achieve with my project.

In the paper, Nijissen discusses the benefits of using the Monte Carlo algorithm to evaluate what moves to play at each level of the game tree. [8] The Monte Carlo Tree Search algorithm I have implemented uses the Monte Carlo algorithm *along with* a tree to store the results of the various executions of the algorithm. Finding this paper confirmed to me that investigating the Monte Carlo Tree Search algorithm for the project would be beneficial, so I began researching the topic in depth.

## 2.4 – Monte Carlo Tree Searching

### 2.4.1 – Monte Carlo Methods

Monte Carlo methods are “a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Their essential idea is using randomness to solve problems that might be deterministic in principle.” [11] Monte Carlo algorithms are typically used in game playing applications by generating many outcomes from the current game state and observing the fraction of these that fit into one or more categories, or that exhibit one or more properties. [13] By generating enough possible outcomes, an accurate representation of the probability of success of each move or choice can be derived. [12]

A Monte Carlo (MC) algorithm for deciding what move to use in a game of Othello could involve “simulating” potential playouts from the current game state, by selecting a legal move and simulating a full game from that move onwards, typically by randomly selecting the next moves to play until the game is complete. The result of the simulation can then be stored as a win or a loss for the player, and once all simulations have been carried out, the move that resulted in the most wins could be selected as the most promising move.

Since MC methods represent a broad range of algorithms, the method mentioned above is not the only way the MC methodology could be applied to Othello.

### 2.4.2 – Adding Tree Searching

One flaw of the previously mentioned MC method for choosing a move in Othello is that it only examines the game tree to a depth of 1 lower than the depth of the current game state, and since all the moves chosen in the simulation after the initial move are random, the results can be somewhat inaccurate.

The Monte Carlo Tree Search (MCTS) algorithm does not suffer from these problems, due to the game tree that the algorithm manages when evaluating the available moves. Each node of the tree represents a game state, with the edges branching from it representing the moves that change one game state into another. Each node also stores the number of game simulations (also referred to as playouts by some texts) that the game state was featured in, and the number of those simulations that resulted in a victory for the player. [14]

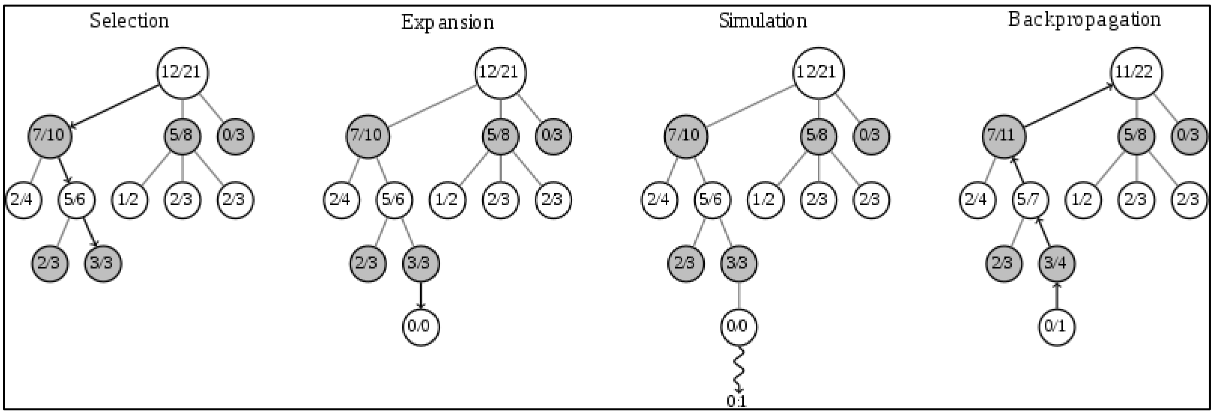


*Figure 1: The 4 Main Steps of the MCTS Algorithm (https://en.wikipedia.org/wiki/File:MCTS_(English).svg)*

The figure above shows a visualisation of a possible MCTS algorithm's game tree. The root game state has three possible moves that can be played, as represented by the three branches originating from the root node of the tree. Additionally, the root node also stored the ratio "12/21", which represents that out of the 21 total simulations the root game state was featured in, the player won 12 of them. The way the MCTS expands the game tree is based on which game states are the most promising to evaluate. [14]

The MCTS' process for constructing the search tree can be described in four steps:

1. **Selection**: To expand the tree, the MCTS algorithm begins at the root node, and selects successive child nodes until a leaf node is reached. The node chosen at each level of the game tree is the one deemed the most promising to explore, which can be based on which node has the most victorious simulations versus their total simulations, or via a specialised selection measure.
2. **Expansion**: The leaf node L is then expanded, with one or more new leaf nodes added as children of L to represent the possible moves from the game state (assuming that the game can continue from L). [14]
3. **Simulation**: One of these new child nodes is then selected, and a simulation of the game from that node onward is run. [14] The simulation starts from the game state represented by node L, and applies random legal moves to it in the normal order of play, to simulate the two players repeatedly playing moves, until the simulated game reaches the end.
4. **Backpropagation**: Once the simulation is complete, the result is propagated back up through each of the nodes between the child node and the root node. [14] All the affected nodes increment their total simulations counter by one, and can increment their victorious simulation counter by one, but only if the simulation resulted in a victory for the player.

Once the algorithm has generated enough simulations, the move with the highest win percentage is chosen at the move to play.

The advantage that the game tree gives to MCTS over a normal MC algorithm is that it can analyse moves further down the game tree, which provides the AI with a greater understanding of the future of the game. Parts of the game tree can also be reused, since a child of the root node can be made the new root node if the game state that the child node represents is chosen as the next move. Finally, the MCTS spends much less time computing moves that are not promising, since they are much less likely to be selected for simulation.

However, care needs to be taken when selecting moves to simulate, as a balance needs to be struck between selecting moves with a high chance of being successful versus selecting moves that have been simulated very few times. One way of maintaining a balance between these two types of moves/nodes is to use a specialised node selection formula, such as the UCT (Upper Confidence Bound 1 applied to Trees) formula. UCT produces a value based on the various parameters of the algorithm, tree and current node to produce a value; the node with the highest value from this formula should be used as the most promising node in the Selection stage of the algorithm. [14] The full UCT formula and its parameters can be found in Appendix C.

MCTS algorithms also have the option of using light or heavy simulations; light simulations are quick to run but less accurate (e.g. random move selection), whereas heavy simulations use heuristic methods (such as minimax searching) to determine moves to play, but take longer to run each simulation. [14]

The competency of the MCTS can be improved by incorporating domain specific knowledge; one such example is knowing whether or not the same game state can be reached via many different permutations of moves. [14] In Othello, two game states that are rotated in different orientations or that are reached via different sequences of moves may have the same board layout, thus allowing knowledge gathered about one state to be shared with another.

Overall, the MCTS algorithm is best suited to non-probabilistic perfect information games with a finite number of moves per game, one of which is Othello. As a result, I chose to make the MCTS algorithm one of the main focuses of my project.

## 2.5 – Deep Learning

### 2.5.1 – Machine Learning and Deep Learning

Machine learning is a field within Artificial Intelligence that researches ways in which computers can become capable of learning without the need for these lessons to be explicitly programmed. Machine learning has aided many fields, such as computer vision, by creating ways in which a computer can learn characteristics of a set of example data, and then using its knowledge to classify previously unseen sets of data. [15]

Deep learning differentiates itself from machine learning through its use of Artificial Neural Networks (ANNs) to understand and interpret the data it is given. ANNs mimic the human brain's structure by creating a network of neurons (also known as nodes) that are connected by various links; each link has a different weight to it, and neurons can send signals along these links depending on how strong the signal they are receiving is. When data is passed to an ANN, the various layers and clusters of neurons pass signals to each other until the output layer of the network is reached, at which point the value(s) of the output nodes are returned. [15]

The key factor that allows ANNs to be used for learning tasks is that each link's weight can change over time as the deep learning process is run; by informing the ANN of whether or not it was correct in its interpretation of the input data, it will readjust the weights of the links between nodes to correct its outputs. [15] As more data is passed through the ANN, the more accurate its interpretations become.

### 2.5.2 – Classification vs Regression

The types of ANNs can be split into two categories, classification and regression.

When deep learning is used for classification, the network produces a class for each piece of training data that it has been given, which can then be compared to the label that has been assigned to said data. The network changes the weights of the links in its structure depending on how many pieces of data the network can correctly classify in the training phase. As Classification ANNs typically output a discrete class for the data, they can be used to divide data into subsets based on their attributes.

Regression-based ANNs mainly differ in what data is returned from the ANN, as they provide a numerical evaluation of the data on a continuous scale, instead of a class for the data. Regression ANNs can be trained using labelled or unlabelled data, which can make them more suited for tasks where providing labels for the data is difficult or impossible.

For my project, given that both methods seemed applicable to Othello at the project's outset, I planned to carry out various experiments using both classification and regression ANNs, and I shall discuss the success of the process in the Evaluation section.

### 2.5.3 – Using Deep Learning with Othello

In theory, any Artificial Intelligence with deep learning has limitless potential for learning its task and improving at it. When it comes to game-playing AI programs, it is entirely possible for an AI to improve its decision-making abilities just by playing against itself. [15] In March of 2016, the AlphaGo AI defeated the Go world champion Lee Sedol, becoming the first AI to do so. AlphaGo was trained by playing it against another version of itself, which allowed its deep learning algorithms to tune its internal ANN to a high skill level. [15] [16] However, as I felt that implementing the necessary functionality to allow for continuous learning was too large of an undertaking for this project, I instead focussed on producing the best results from the initial ANN creation process, and testing the resulting networks in the AI system of my project.

As previously mentioned, an AI typically makes use of an evaluation function to determine the value of a game state. I decided to use deep learning to create a new type of evaluation function that my AI could use, which returns a valuation of the game state by passing the locations of the counters to an ANN constructed before the game began. This would create an evaluation function capable of evaluating game states based on thousands of examples of Othello games, thus providing the AI it is part of with an accurate way to determine which moves are better than others.

For this purpose, I initially expected regression to be the type of output I would use, but I was able to create classification networks that provided the same evaluation functionality. I was also able to find a database of tournament-level Othello games, which I used to train and test the ANNs for use as evaluation functions.

All in all, this research suggests that the best AI for Othello would use a Monte Carlo Tree Search algorithm to determine what move to play, while using a Deep Learning-trained evaluation function to run its heavy simulations accurately.

# Section 3 – Design of Othello and the AI

This section contains various plans and pieces of information about the Othello implementation and the AI program, such as potential UI designs, the class structure of the implementation, utilisation of deep learning/ANN functionality, etc.

## 3.1 – UML Class Diagram

To provide myself with a clear class structure to follow in the development stage, I created a UML Class Diagram to outline the functionality of the most important classes in the implementation.

### 3.1.1 – Overall Structure

Classes associated with the Othello game logic are outlined in blue, classes associated with the UI are outlined in green, and classes that will contain AI routines are outlined in red. Though the diagram does not list every class included in the final program, the classes shown are the most important to the Othello game's logic and display, and the AI program's operation. Each class has a name, example methods and fields, and connections to related classes.

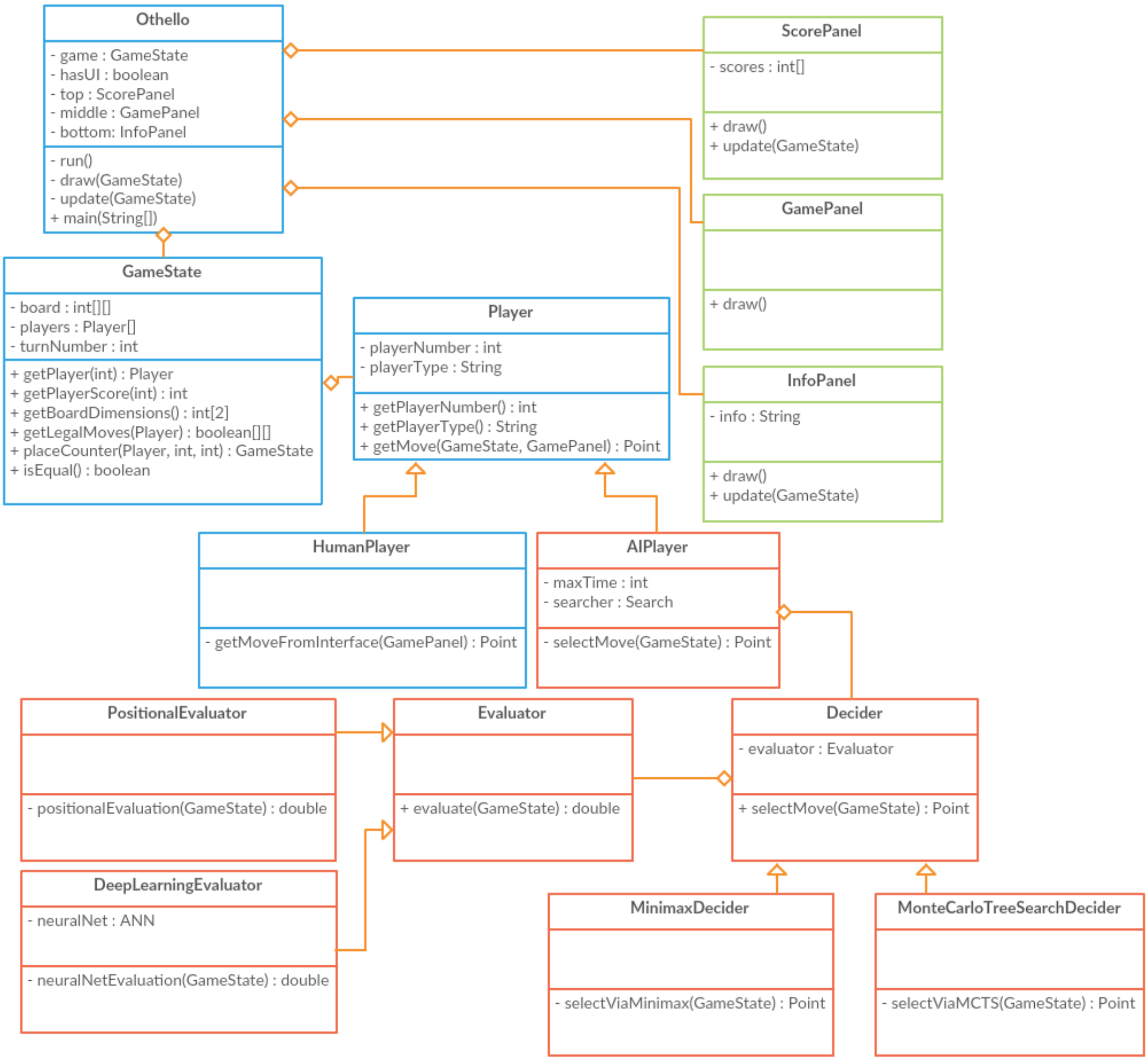


*Figure 2: Structure of my final Othello game and AI program*

### 3.1.2 – Game Logic Classes

Since the game logic classes were the most sensible ones to create first, I designed them without any dependencies on the UI and AI classes, meaning they could be implemented without worrying about coupling.

The Othello class is the main program that is launched, and constructs all of the necessary objects at run time; it essentially manages the game and the UI to ensure data is correctly exchanged between the two.

One of the most important classes in the project is the GameState class, as this represents a possible state in the game of Othello. Since many GameState objects may be held in memory at one time, I have kept the class' data footprint as small as possible, by storing the state of the board as a 2D array of integers, and only storing the players and the turn number alongside this information. I have also provided examples of many of the methods this class provides as its interface. These methods allow the AI players to manipulate and evaluate the GameState as much as they need.

An important method of the GameState is the "getLegalMoves" method, which returns a Boolean array that describes the places that the current player can place counters; by passing the coordinates of a True position in this array to the "placeCounter" method, the GameState will be updated with a counter placed for that player at that location. I initially planned to use integer coordinates to describe locations on the Othello board, but I decided to use Java's built in Point class instead, so that I only had to return one object from any method that wanted to provide coordinates.

Additionally, I defined the Player class, which represents an entity that is capable of making moves in the game. This can be a HumanPlayer, which receives moves from the UI of the program, or an AIPlayer, which will programmatically determine what moves to play. Both classes will have the Player interface, meaning that the GameState doesn't have to treat the two types of Player class differently, though the player's internal type is tracked to allow the classes to be distinguished if necessary.

### 3.1.3 – UI Classes

The three UI classes represent three different panels I have built the interface out of. All three panels have "draw" methods, which are used to visualise their part of the interface based on the GameState provided, and the top and bottom panels also have variables to cache some of the GameState data, along with an "update" method to update their caches.

The most notable class is the GamePanel, which needed to be capable of detecting mouse events to determine a move the player wants to play. As a result, it uses Java's built in MouseListener interface, which enables it to receive MouseEvent objects, which can then be converted the coordinates of player moves.

One important note that is not represented on the class diagram is that the three UI classes are extensions of the JPanel class in Java's Swing package, which allowed me to easily organise their layout within the window, thanks to the class' existing methods and behaviour.

### 3.1.4 – AI Classes

The last section of classes to be implemented was the various AI classes, all of which are connected to an AIPlayer object in some way. The AIPlayer class stores some basic information, such as the time allowed for the player to make a move.

The AIPlayer's behaviour is then determined by the other classes provided to it during construction. The Evaluator classes represent various evaluation functions that can determine the value of a GameState, while the Decider classes are the classes that determine which move should be played by searching through the available moves. The Decider class' "selectMove" will be used by the AIPlayer to retrieve the coordinates that they should play a counter on.

I chose to construct the AIPlayer class via aggregation for two reasons:

1. I wanted to create a simple AI capable of playing Othello before implementing the complex Monte Carlo Tree Search and Deep Learning functionality. Using this aggregation method of creating the AI would allow me to swap in the MCTS and Deep Learning classes later in the project.
2. As previously mentioned, I wanted the final program to have varying difficulty levels for human players to compete with. By allowing for various Decider and Evaluator algorithms to be used by an AI, this gives the user the ability to adjust the AI's intelligence to their liking.

## 3.2 – Programming Language

For my project, I decided to use the Java programming language. During my time at Cardiff University, I have used Java many times to complete various assignments, and as a result I am very familiar with how it works. By using Java, I didn't need to learn any new programming languages for the project, which increased the amount of time I could spend on other tasks. This also gives the final program the benefit of being able to be played on any operating system with the Java Virtual Machine (JVM) installed on it.

Additionally, since Java is an Object-Oriented programming language, I was easily able to create and manage the various classes that will make up the game and AI, including how they interface with one another, which was important for allowing the AI to interact with the Othello game, and for allowing the AI's decider and evaluator classes to be chosen at runtime.

Finally, Java provides numerous packages and classes for easy UI creation, such as Swing. By creating an interface using these packages, I was able to quickly create the necessary graphical interface to easily show the game state.

### 3.3 – Approach to Implementing the Game Logic

One of the most important factors of the project is ensuring that my programmatic representation of the Othello ruleset is completely accurate to the original game. To achieve this, I decided to place all of the game logic code into the GameState class, and provided methods to allow manipulation of the game, without allowing illegal moves or board manipulation to occur. Before beginning programming the GameState class, I set out these rules:

1. Do not allow direct manipulation of the internal variables (board state, turn number, etc.) other than through appropriate methods (e.g. methods for placing a counter, or viewing the board without changing it)
2. When changing the game state, such as via placing a counter onto the board, return a new GameState object, rather than changing the existing object.
3. Provide methods to allow easy access to statistics for AI agents to use, such as the player scores, game end determination, number of moves remaining, empty spaces remaining, etc.
4. Enforce internal checking to ensure only legal moves can be played, and provide AI agents with a list of legal moves. Subsequently, this should mean that only legal GameState objects can exist.
5. Optimise the class' computation time and data footprint as much as possible, since it will be a frequently created and used class, and optimisations in the game state class will affect the speed of the AI's decisions.

I mainly derived these rules from a previous piece of coursework I had received involving AI behaviour, as they are related to problems I experienced there. For example, I created rule 2 to make it so that each GameState object is a representation of *one* possible board configuration, and cannot be changed into any other board state, while still allowing creation of child states of itself via its methods. This prevented errors related to unexpected game state mutation from occurring.

To ensure that the logic was correctly implemented, I decided to test the GameState during development by recreating legal layouts and checking to ensure that all values returned matched with the expected values for that layout (e.g. accurate scores calculated, correct list of legal moves returned, etc.)

Additionally, I also had access to many archived Othello tournament games, which contained the sequence of moves played and the final scores of each game. In theory, if these games could be simulated without error when using the GameState class, the implementation would be faithful to the official ruleset of Othello. I decided to postpone this method of evaluation until development of the project was almost complete, as the GameState was likely to change many times throughout development as functionality was added and changed.

## 3.4 – UI Design

To ensure the users of the final program will understand how to operate the Othello game and its parameters, I designed the UI and command line arguments ahead of time.

### 3.4.1 – Visual Design

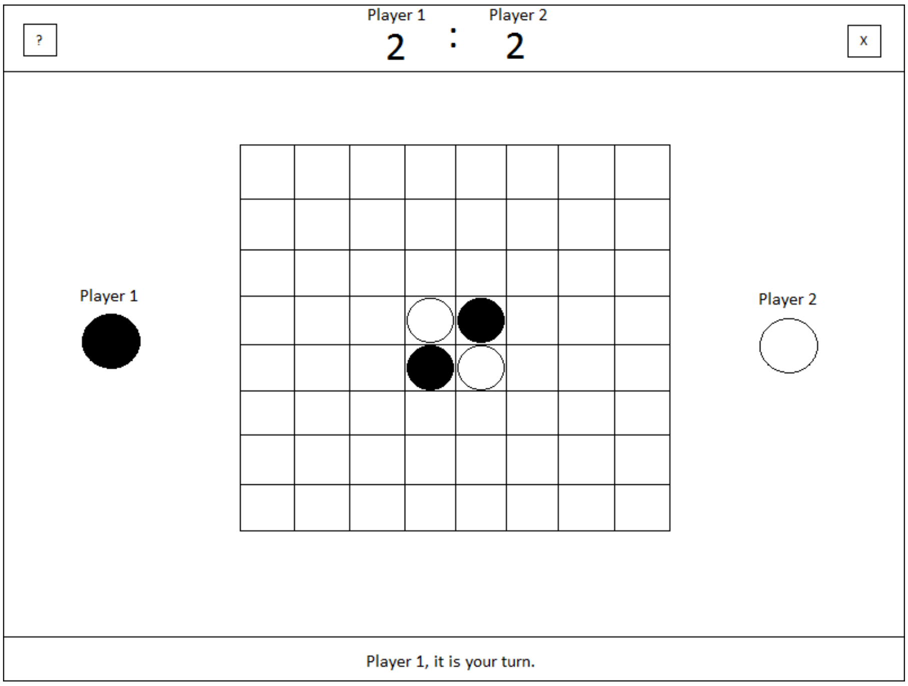


*Figure 3: Initial design for the Othello game UI.*

This user interface was designed to be easily implementable in a Java program via Java's Swing packages, while also still providing the functionality necessary for the Othello game. The interface is split into three panes; the top pane is used to display the current score for each player in the current Othello game, while the bottom pane is used as a notification bar to display messages about the game, such as who is currently taking their turn.

Most importantly, the middle pane is the main display pane, containing the Othello board and counters, along with displaying which player is in control of which counter type. This gives the human player(s) a clear image of the Othello board. This pane is where human players can play counters onto the board, by simply clicking on the square they wish to play on, assuming it is their turn. The whole interface is contained within a simple window that will be displayed on the user's desktop upon launching the program.

### 3.4.2 – Launching the Game and Changing Arguments

As the game is launched from the command line, the user can change various values of the program via the command line arguments for the program. For example:

```
java OthelloAIProgram -player1 Human -player2 AI(Minimax,Positional)
```

The final program could interpret a command like this by setting the first player in the game to a human-controlled player, while the second player's decisions will be run by an AI. This system allows various other values to be changed, such as which player moves first, whether the UI is displayed or not, and even the size of the board.

## 3.5 – Game and AI Pseudocode

I created the following pieces of pseudocode to allow for the functions they represent to be quickly understood and implemented when I needed to begin work on them. The code shown below represented the methods that I believed would be the most difficult to implement, hence why I planned their functionality and structure prior to starting development.

### 3.5.1 – Game State Functions

**Board Representation, Turn Number and Score Calculations:**

```
board = an 8 x 8 integer array

int[][] viewBoard() { Return a clone of this.board }

int getTurnNumber() {
        int total = -3         // Since there will always be 4 counters at the start
        For each space in this.board {
                If the space has a counter in it {
                        Increment total
                }
        }
        Return total
}

int getScore(Player p) {
        int total = 0
        For each space in this.board {
                If space has a counter and counter matches the player’s colour {
                        Increment total
                }
        }
        Return total
}
```

As mentioned in Section 3.1.1, I chose to use an integer array to store the board state to reduce the GameState class’ overall footprint. This board can then be cloned via Java’s built in methods when requested via the viewBoard method; this allows a copy of the board to be viewed and manipulated without affecting the object’s internal board.

I have also provided code that shows how the turn number and player score could be determined; in the final program, an integer field tracks the turn number, but a similar method of scanning the board is used to calculate the players’ scores.

### Legal Move Determination and Move Placement

```
Point[] getLegalMoves(Player p) {
    Point[] moveList = empty list
    For each space in this.board {
        If space has a counter and counter matches the player’s colour {
            For each horizontal, vertical and diagonal direction from the space {
                If there is another player counter in that direction
                and there are opponent’s counters inbetween {
                    Add the space to the moveList
                }
            }
        }
    }
    Return moveList
}

boolean hasLegalMoves(Player p) {
    If getLegalMoves() returns >= 1 moves {
        Return true
    } else {
        Return false
    }
}

GameState placeCounter(int x, int y, Player p) {
    Point move = new Point(x, y)
    Point[] legalMoves = getLegalMoves(p)
    If move is in legalMoves {
        GameState newState = a clone of the current state
        Place the player’s counter onto the board of newState
        Flip the necessary counters on newState’s board
        Return newState
    } Else {
        Refuse the move, by returning the same state, or by throwing an exception
    }
}
```

The most important methods for allowing players to complete a game of Othello are for determining where moves can be made, and for playing said moves.

The getLegalMoves function shows that legal moves can be determined by using the rules of Othello (see Section 2.1.1); the program loops through the board array and determines if there are any vertical, horizontal, or diagonal lines between the current empty space and any counters of the current player’s colour. If a line is discovered that would also flip counters of the opponent’s colour, then the move can be considered legal.

The method of determining legal moves shown above places each legal move coordinate in a list; this is a decision I later changed, as the final program uses an array of Booleans to describe where the legal moves are found.

Meanwhile, the placeCounter pseudocode shows how the getLegalMoves method can be used to ensure only legal moves can be played onto the board, while also ensuring new GameState objects can be returned from the method. This means that the created GameState is not linked to the current GameState in anyway, and can be manipulated without affecting the original object.

### 3.5.2 – Minimax Methods

Code structure based on pseudocode found at reference [17].

**Decide Method:**

```
Point decide(Player p, GameState g) {
        int defaultDepth = 6
        float bestScore = -∞
        Point bestMove = null
        For each move in g.getLegalMoves(p) {
                float moveScore = getMinValue(g.otherPlayer(p), g.placeCounter(move, p),
                                                depth – 1, alpha, beta)
                If bestMove == null or bestScore < moveScore {
                        bestScore = moveScore
                        bestMove = move
                }
        }
        Return bestMove
}
```

This is the standard decide method that all deciders must provide. For the Minimax player, the move to play is decided by running a minimax evaluation on all of the available moves, and then selecting the move with the highest score.

**Minimaxing Methods:**

```
float getMaxValue(Player p, GameState g, int depth, float alpha, float beta) {
        If depth == 0 { return evaluation of g }

        float best = -∞

        For each move in g.getLegalMoves(p) {
                float moveScore = getMinValue(g.otherPlayer(p), g.placeCounter(move, p),
                                                depth – 1, alpha, beta)
                best = max(best, moveScore)
                alpha = max(alpha, moveScore)
                If beta <= alpha {
                        Return best
                }
        }

        Return best
}

float getMinValue(Player p, GameState g, int depth, float alpha, float beta) {
        if depth == 0 { return evaluation of g }

        float worst = ∞

        For each move in g.getLegalMoves(p) {
                float moveScore = getMaxValue(g.otherPlayer(p), g.placeCounter(move, p),
                                                depth – 1, alpha, beta)
                worst = min(worst, moveScore)
                beta = min(beta, moveScore)
                If beta <= alpha {
                        Return worst
                }
        }

        Return worst
}
```

The Minimax methods for determining the max/min value at a level of the game's decision tree needed to be recursively linked together in such a way that the correct one is called to correctly evaluate the player's or opponent's moves. I made sure to correctly set this recursion up in my pseudocode to ensure the implemented versions of these functions worked correctly.

Additionally, to improve their performance a suitable pruning method needed to be selected; I chose Alpha-Beta pruning, as it was easy to implement while providing a large gain in performance. Furthermore, to ensure the recursive calls end, a base case was needed to terminate the process once a certain depth was reached. This can be seen as the first line of the two methods above.

### 3.5.3 – Monte Carlo Tree Search Methods

**Decide Method:**

```
Point decide(Player p, GameState g) {

        Create and initialise a tree structure, wich each node storing
        a win counter and a total counter

        While time limit is not exceeded {

                // Selection stage
                TreeNode nodeToExpand = root of the tree
                TreeNode[] trail = [ root of the tree ]
                While nodeToExpand isn't a leaf node {
                        nodeToExpand = child of nodeToExpand that maximises UCT score
                        Add new value of nodeToExpand to trail
                }

                // Expansion and simulation stage
                Select a possible child of nodeToExpand, and create it
                boolean victory = simulate(nodeToExpand's new child)

                // Backpropogation stage
                For each node in trail {
                        If victory is true {
                                Increment win and total counters of the node
                        } Else {
                                Only increment the total counter of the node
                        }
                }

        }

        Return the child of the root node with the highest win percentage

}
```

The decision process for the MCTS algorithm is notably more complex than any other decider. The method above shows how the tree is constructed via the four main stages of the MCTS algorithm: node selection, node expansion, game simulation, and backpropagation of results.[1]

---

[1] For more information on these stages, see Section 2.4.2.

**Simulation Processes**

```
boolean simulateHeavy(Player p, GameState g, TreeNode n) {

        GameState current = advance g until it matches where node n is in the tree
        Player playing = p

        While current.isOver() is false {
                Point move = run minimax evaluation on current to determine best move
                current = current.placeCounter(move, playing)
                playing = current.otherPlayer(playing)
        }

        Return current.isWinner(p)

}
```

```
boolean simulateLight(Player p, GameState g, TreeNode n) {

        GameState current = advance g until it matches where node n is in the tree
        Player playing = p

        While current.isOver() is false {
                Point move = select random move from current.getLegalMoves(playing)
                current = current.placeCounter(move, playing)
                playing = current.otherPlayer(playing)
        }

        Return current.isWinner(p)

}
```

Since I wanted to support both heavy and light simulations in the final MCTS algorithm I produced, I created pseudocode for the two ways of simulating the games. These methods represent the Monte Carlo part of the MCTS algorithm, as they provide the algorithm with a large amount of simulation data over their various executions to allow a decision to be made.

This pseudocode helped me to realise an important decision about the structure of my classes; as the MCTS decider had to be capable of using the minimax methods for its heavy simulations, it also had to be a subclass of the Minimax class to inherit all of the necessary functionality. This saved me a lot of time, as I didn't have to reimplement the minimax code into the MCTS decider class.

### 3.5.4 – Deep Learning Evaluator Functionality

**Construction:**

```
DeepLearningEvaluator(String annFilePath) {
        Load saved ANN from the provided file path
        Store the ANN in a private field for later use
}
```

Since I had decided to separate the ANN construction process from the actual Othello game, I knew that for creating the DeepLearningEvaluator class, I only needed to re-load the created network into the memory of the object at construction.

**Evaluate Function:**

```
float evaluateClassification(GameState g, Player p) {
        Pass the current game state to the loaded in ANN.
        Receive the classification of the class.
        Convert the classification into a scalar value, and return it.
}

float evaluateRegression(GameState g, Player p) {
        Pass the current game state to the loaded in ANN.
        Receive the result as a scalar value.
        Return the scalar value.
}
```

As I initially expected to create both regression and classification networks, I created pseudocode to show how both could be utilised as evaluation functions. Every evaluator must provide an evaluate function that can return a value based on the game state and the player it is provided. Though I didn't know the specific details of how the network would function at the time, I was able to roughly predict how the game state, player and the ANN would interact, based on the ANN research I carried out.

## 3.6 – Deep Learning Framework and Approach

### 3.6.1 – Deciding how to Create ANNs

At the outset of the project, I was somewhat familiar with the Deep Learning process, thanks to my studies. After some research on how I could create an ANNs for my AI to utilise, I felt that it would be necessary to spend as much time creating and evaluating a variety of networks, as it was very much an incremental process; determining how to structure the internal nodes of the network, choosing how to provide the data to the network, and producing the network itself is a very time consuming process.

From this information, I determined that the best way to approach this stage of the project was to first create a system for easily producing the networks using various data formats, network structures and other variables. Assuming this system would allow for ANNs to be easily created, I could use the remaining development time to evaluate and improve these ANNs for the project by experimenting with different data formats and network configurations.

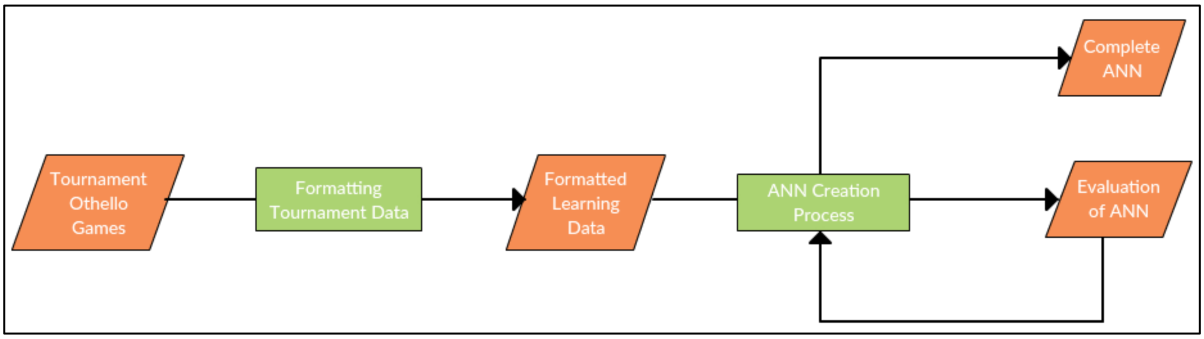


*Figure 4: Flowchart of how I created the ANNs for the program. The evaluation of each ANN was to be used to improve the next network that was produced.*

### 3.6.2 – Utilising an Existing Framework

Additionally, to reduce the amount of work required to implement the Deep Learning functionality necessary for the project, I chose to use a Deep Learning framework to provide some of the necessary functions for creating and using ANNs. I eventually settled on using the Deep Learning For Java (DL4J) framework, which provided all of the functionality that I needed. I was able to download some of the necessary JAR files from their website[2], while the others were obtained via an automated dependency manager known as Maven. However, this process took me some time; as I had never used Maven before, so I initially didn't know how to configure its files and settings to deliver the JARs to my project.

Additionally, after I had successfully obtained all of the files necessary to begin using DL4J, I encountered another problem that I had not expected; due to DL4Js small user base, it lacked any extensive documentation about its various classes and functions, and only provided a short tutorial for beginners[3]. This led to me spending a lot of ANN development time on debugging various DL4J errors and researching how to utilise the framework via unofficial help articles. Though the official DL4J website offered a help chat, the users within were less than friendly. Eventually, I gained a sufficient understanding of the framework to allow me to use it in my project, but the process could have been much smoother had there been more help available.[4]

---

[2] Official website: https://deeplearning4j.org/

[3] Tutorials available at: https://deeplearning4j.org/quickstart

[4] I also used the tutorials from the following YouTube channel to aid my learning: https://www.youtube.com/channel/UCa-HKBJwkfzs4AgZtdUuBXQ

# Section 4 – Implementation of the Game and AI Systems

With the necessary information gathered and the program designed, I set out to program the Othello-playing Artificial Intelligence, and its associated classes, such as the Othello game state representation and a graphical user interface. This section discusses they ways I went about implementing the necessary functionality and the issues I faced along the way.

## 4.1 – Overview

To help organise the classes in the project, I divided the code into packages based on their usage; each section will list the packages associated with it.

- Deciders – Contains classes used to search through potential moves an AI player can make.
- Evaluators – Contains evaluation functions for determining the worth of a game state.
- Games – Contains classes that represent Othello and enforce its ruleset.
- Learning – Contains deep learning functions and classes for creating ANNs.
- Main – Contains the main Othello class for running the game.
- Players – Contains all the player classes used by the game.
- UI – Contains classes used to display information to the screen.
- Util – Contains static classes containing useful methods.

## 4.2 – Game Logic Development

Relevant packages: Games, Main, Players

### 4.2.1 – Creating the Game Representation

I started development by creating the programmatic representation of the Othello game, which mainly involved implementing and testing the functions of the GameState class. I began by adding the methods required to allow a basic game of Othello to be run to the class, such as a constructor, accessors for the various fields, legal move checks and counter placement operations. Once these were in place, I created additional methods on top of these to provide an easier to use interface for the AI player to utilise, including score calculation, victory or loss determination, and state comparisons.

The biggest obstacle I encountered at this stage was determining where a player could play legal moves; though I had pseudocode from the Design stage to show how to structure the algorithm, I struggled to determine how to analyse if counters could be flipped in each direction from a selected board space. The final version of the GameState class uses a series of for loops to evaluate each horizontal, vertical and diagonal line from the selected space; though this way of brute forcing the algorithm is not ideal in terms of efficiency, I have added code to terminate the check once the space is confirmed as a legal move location. Additionally, since the GameState class uses an array of integers to represent the board state, the comparisons needed for these calculations can be carried out quickly. Some of the code used in the getLegalMoves method can be seen in Figure 5 below.

```
/**
 * Determines how many counters would be flipped in a certain direction
 * if a counter is placed at the provided coordinates.
 */
private int getFlippedCounters(int initRow, int initCol, int deltaRow,
                              int deltaCol, int counterType) {
   // Initialising counter and coordinate variables.
   int lineLength = 0;
   int row = initRow + deltaRow;
   int col = initCol + deltaCol;

   // Loop to travel along the line specified by the delta parameters.
   while (row >= 0 && row < boardSize && col >= 0 && col < boardSize) {

      // Determine what counter is at the current locaation in the line.
      if (getBoardValue(new Point(row,col)) == counterType) {
         // Player's counter found, return number of counters
         // between initial counter and this counter.
         return lineLength;
      } else if (getBoardValue(new Point(row,col)) != COUNTER_EMPTY) {
         // Opponent counter found, increment number of counters on line
         // that can be flipped.
         ++lineLength;
      } else {
         // Empty space found, no bracketing possible.
         return 0;
      }

      // Move to next location.
      row += deltaRow;
      col += deltaCol;

   }

   // Edge of board reached, no bracketing possible.
   return 0;

}
```

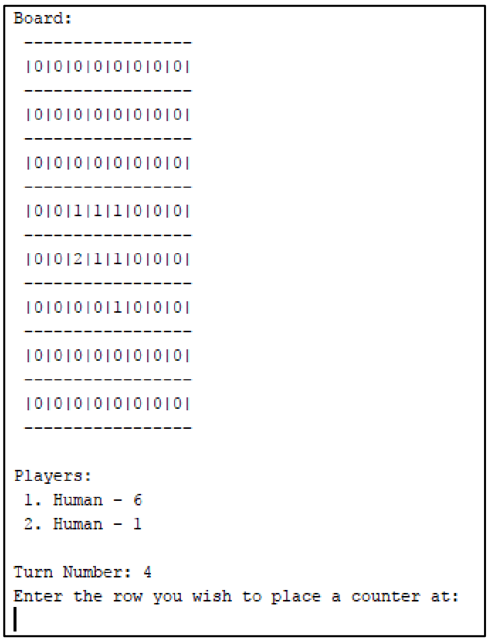
*Figure 5: The code used to determine how many countered can be flipped in a specified direction.*

One major change from the design stage was that the legal moves function now provides a grid of legal move locations, rather than a list of coordinates. I chose to do this to save space, as a 2D array of Boolean values has a smaller representation in memory than a list of Point objects. However, this resulted in some AI code iterating over the Boolean array even after all legal moves were found, resulting in wasted time. This way of returning legal moves from the GameState is something I would change in the future, but works perfectly fine for the current implementation.

Once the legal moves could be calculated, I was able to complete the playMove function, thus allowing counters to be played onto the board of the GameState. I waited until the legal moves method was complete so that I could implement an important check within the playMove function; to ensure that only legal moves can be played, the playMove function checks the coordinates passed to the function to determine if they matchup with a legally allowable Othello move. Though this check seems simple and even unnecessary, it acts as a full check to ensure the game logic is implemented properly, as there is no other way of generating game states other than through this method, and thus no other way to violate the game's rules. Since an exception is thrown by the method if an illegal move is passed to it, it can detect any errors in game logic understanding or AI behaviour, and has been helpful in ensuring the AI behaviour is correctly implemented. I chose to implement this check instead of simply trusting the behaviour of my game implementation as it allows for the accuracy of the ruleset to be easily evaluated, either by examining the legal moves provided in test game states, or by simulating games to ensure they can be played.

The Othello class was also created early on in development. This class contains a main method that allows for it to be executed, which runs all of the additional code needed to store and manipulate GameStates to create a full Othello game. It received incremental updates over time as the number of classes grew (as discussed in Section 4.3.2), and as more optional featured were implemented, but at the beginning of the project it could create and run a basic Othello game between two human opponents, while following the rules of the game accurately.

To view the GameState during development while the GUI was not in place, I overwrote Java's default toString method for the class to display the board layout in the console. The Othello class then simply printed out the GameState at each iteration of the game loop. Though the GUI was later implemented and became the primary viewpoint used for the remainder of development, the console output is still available to use in the final program, should the GUI be disabled.



```
Board:
 -----------------
 |0|0|0|0|0|0|0|0|
 -----------------
 |0|0|0|0|0|0|0|0|
 -----------------
 |0|0|0|0|0|0|0|0|
 -----------------
 |0|0|1|1|1|0|0|0|
 -----------------
 |0|0|2|1|1|0|0|0|
 -----------------
 |0|0|0|0|1|0|0|0|
 -----------------
 |0|0|0|0|0|0|0|0|
 -----------------
 |0|0|0|0|0|0|0|0|
 -----------------

Players:
 1. Human - 6
 2. Human - 1

Turn Number: 4
Enter the row you wish to place a counter at:
|
```

*Figure 6: The text version of the Othello program's UI.*

At this time, I also implemented the Player class, and created the HumanPlayer subclass for it. By using Java's Scanner class, the HumanPlayer can request the user to type in coordinates in the console to select where to place their counter. This allowed for full games of Othello to be played using just the GameState, Othello and HumanPlayer classes.

### 4.2.2 – Implementing Caching

Since each GameState produces a lot of immutable data, I realised during development of the Monte Carlo Tree Search decider that I could very easily cache important information within the object, such as the array of legal move locations or player scores, to save computation time.

```
/**
 * Internal method for determining the cache.
 */
private void cacheSetup() {
   this.legalMovesCache = new boolean[2][this.boardSize][this.boardSize];
   this.hasLegalMovesCache = new boolean[2];
   computeLegalMoves();
   this.scoreCache = new int[2];
   computeScores();
}
```

*Figure 7: The main caching code, which is run at the object's construction time.*

The GameState caches the legal moves of each player, whether each player can play a legal move or not, whether the game is over, and the player's scores. As calculating these values require iterating over the whole board, they are determined upon construction to speed up functions that require this data, at the expense of increasing construction time of the object (though this is still quicker than calculating the values after construction).

When this was initially added to the class, the data was correctly calculated and returned from their corresponding functions. However, upon implementing the MCTS algorithm into a decider, any player that used said decider would return illegal moves to the main Othello program, as detected by the GameState's playMove method. When debugging the issue, I noticed the following odd behaviour:

- Illegal moves would only be returned after the first turn; the first move of the game would always be legal.
- Printing out the grid of legal moves would display the legal move grid for the previous game state rather than the current one.
- The GUI would also show incorrect spaces that the players could play on, even for non-MCTS players.

Eventually I determined the root of the problem to be in the playMove function, specifically in the section of code after the legal move check, as it wasn't allowing the GameState's caches to be calculated correctly.

```
// Creates a copy of the current GameState to return.
GameState tempState = new GameState(this);

// Places the counter the player want to play.
tempState.placeCounter(id, move);

// Processes the flipping of counters from this point.
for (int linesRow = 0; linesRow < 3; ++linesRow) {
   for (int linesCol = 0; linesCol < 3; ++linesCol) {
      int lineLength = lines[linesRow][linesCol];
      int localRow = row;
      int localCol = col;
      while (lineLength > 0) {
         localRow += (linesRow - 1);
         localCol += (linesCol - 1);
         tempState.placeCounter(id, new Point(localRow, localCol));
         --lineLength;
      }
   }
}

// Increment state turn number and return the GameState
tempState.incTurnNumber();
return tempState;
```

*Figure 8: The initial code for the playMove function.*

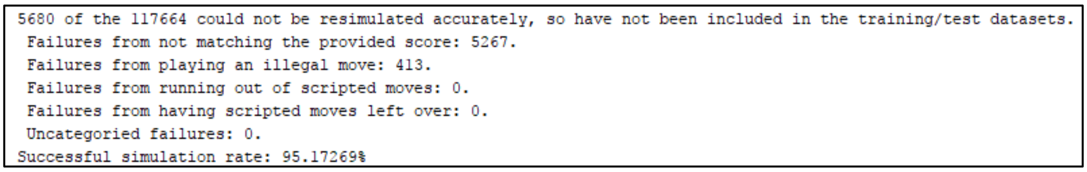
This segment of code is run inside the playMove function. The code clones the current state, edits the game board to reflect the move to be made, then returns the clone. I chose to carry out the game state manipulation this way because of experiences I have had with programs that carry out the change within the existing object (see Section 3.3). However, since the GameState's caches are only calculated at object construction, all the code beyond the first line shown above invalidates these caches. To solve this, I simply added a line to re-clone the temporary game state before returning the object, so that the caches are recalculated based on the newly-flipped counters.

This error was the hardest to debug and solve, as by the time the issues arose I had completed the Game and UI classes along with most of the AI classes, so the amount of code I had to potentially search was vast. Thankfully, I was able to track down the erroneous class and function by examining the exception thrown by the playMove method, then printing out data at key points at runtime and determining where the data could have been incorrectly handled, which led me to the incorrect caches.

### 4.2.3 – Testing the Ruleset and Functionality

As features were added to the GameState class, I made sure to test them by using the text UI, and ensured that the values returned during a simple game of Othello were correct, such and determining if legal and illegal moves were detected, if scores were accurately calculated, etc. There were no major errors discovered during this time.

Though I was confident from this developmental testing I had carried out that I had created an accurate representation of the Othello ruleset, I wanted to confirm this by simulating the tournament-level Othello games[5] using the GameState class. If most of the 100,000 games could be simulated without error, it would be more than enough evidence to show that my Othello implementation was faithful to the official Othello ruleset.



```
5680 of the 117664 could not be resimulated accurately, so have not been included in the training/test datasets.
 Failures from not matching the provided score: 5267.
 Failures from playing an illegal move: 413.
 Failures from running out of scripted moves: 0.
 Failures from having scripted moves left over: 0.
 Uncategoried failures: 0.
Successful simulation rate: 95.17269%
```

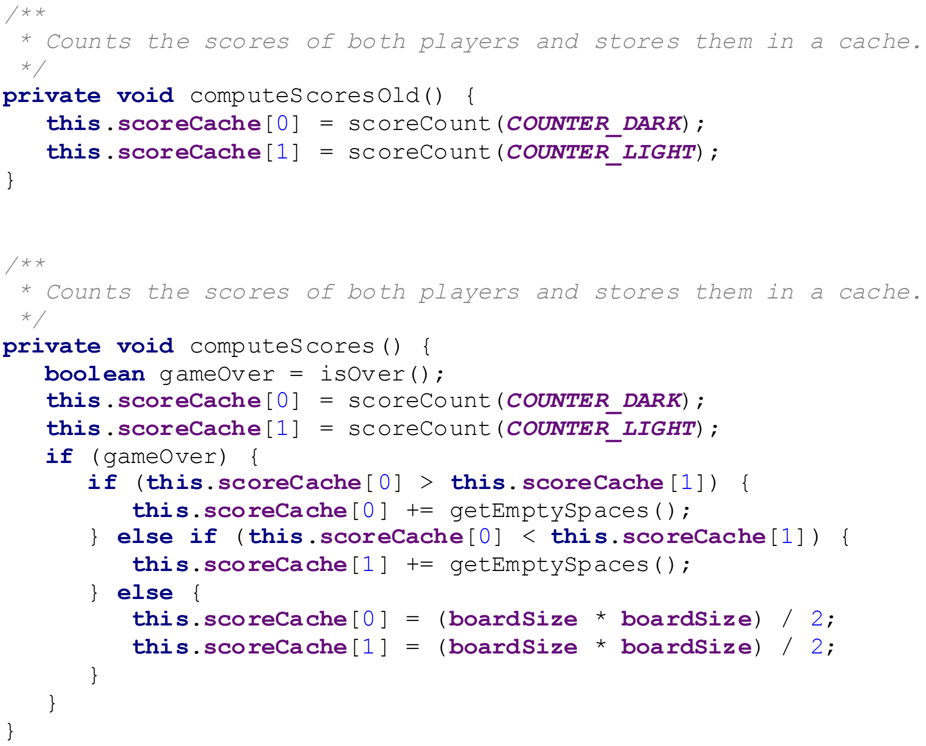
*Figure 9: Results from the first simulation of the tournament games.*

By using the NeuralNetDataHandler class[6] to run through the Othello game data set, I soon discovered that most of the games were able to be fully re-simulated, with a 95% success rate. However, I was surprised to see so many games fail to run due to providing an incorrect score. When examining the dataset of Othello games, I noticed that any tournament games that ended in a draw automatically changed the score to 32-32, while games that ended prior to the board being filled then added the number of empty spaces to the winning player's score. (i.e. a game ending at 59-1 would become 63-1.)

---

[5] These are the games I had gathered for the Deep Learning portion of the project, see Section 4.5.1.

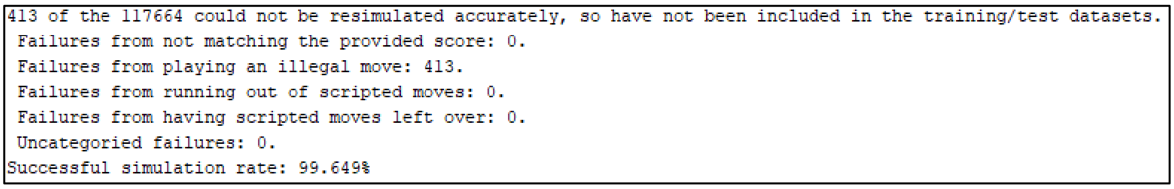
[6] A class created during the Deep Learning portion of the project that validates scripts of completed Othello games by re-simulating them. See Section 4.5.1 for more.



```
/**
 * Counts the scores of both players and stores them in a cache.
 */
private void computeScoresOld() {
    this.scoreCache[0] = scoreCount(COUNTER_DARK);
    this.scoreCache[1] = scoreCount(COUNTER_LIGHT);
}


/**
 * Counts the scores of both players and stores them in a cache.
 */
private void computeScores() {
    boolean gameOver = isOver();
    this.scoreCache[0] = scoreCount(COUNTER_DARK);
    this.scoreCache[1] = scoreCount(COUNTER_LIGHT);
    if (gameOver) {
        if (this.scoreCache[0] > this.scoreCache[1]) {
            this.scoreCache[0] += getEmptySpaces();
        } else if (this.scoreCache[0] < this.scoreCache[1]) {
            this.scoreCache[1] += getEmptySpaces();
        } else {
            this.scoreCache[0] = (boardSize * boardSize) / 2;
            this.scoreCache[1] = (boardSize * boardSize) / 2;
        }
    }
}
```

*Figure 10: The old and new versions of the score calculation code. The scoreCount() method simply sums the counters of a given colour on the board.*



```
413 of the 117664 could not be resimulated accurately, so have not been included in the training/test datasets.
 Failures from not matching the provided score: 0.
 Failures from playing an illegal move: 413.
 Failures from running out of scripted moves: 0.
 Failures from having scripted moves left over: 0.
 Uncategoried failures: 0.
Successful simulation rate: 99.649%
```

*Figure 11: Results from the second simulation of the tournament games.*

I quickly added new code to correctly determine the score in these cases, and re-ran the NeuralNetDataHandler code, which returned a >99% success rate with no errors due to incorrect scoring. Thankfully this scoring error was caught before the Deep Learning work was carried out, so the resulting ANNs were not affected by the old score calculation system.

As for the remaining 413 games, I discovered an article by Patrick Lea that discussed the tournament games dataset:

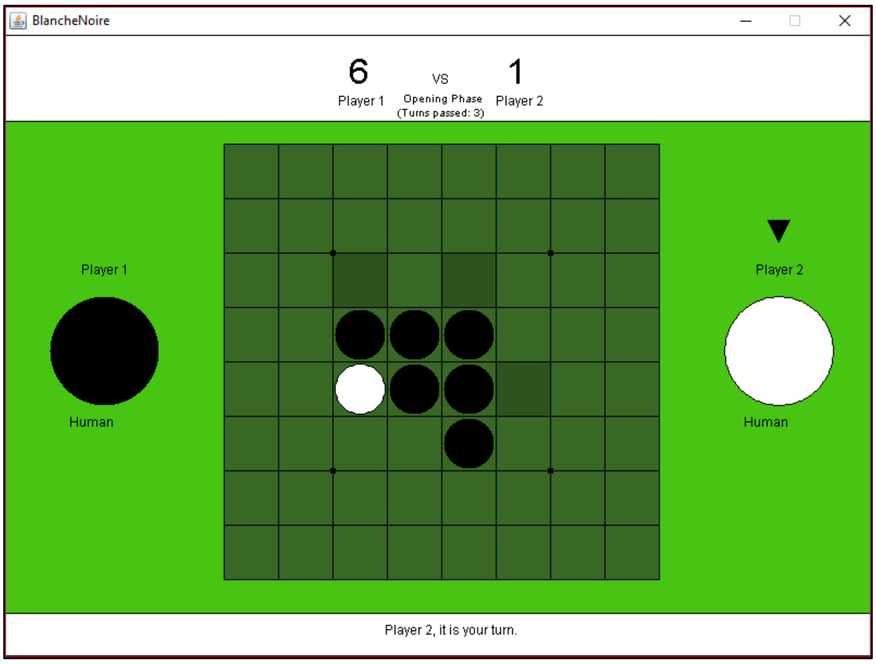
> “I still haven't figured out what it means for Black's score when the game ends *before* neither player can play a piece. The score seems inconsistent. This only occurs in 406 of 107,473 games. (Not a big deal.)” – Patrick Lea [18]

It seems that some games end before an official end to the game occurs, possibly due to the opponent forfeiting. Though my re-simulations have more than 406 broken games, I believe this is due to the fact I am using a more up to date database of games, that may contain slightly more of these erroneous games.

## 4.3 – UI Implementation

Relevant packages: Main, UI

### 4.3.1 – Final Othello GUI



Player 2, it is your turn.

*Figure 12: A screenshot of the GUI being used to run a game between human players. The game state is identical to the one seen in Figure 5.*

After finishing the GameState implementation, I was able to quickly implement the GUI I wanted to for the program, thanks to Java's built-in packages for GUI creation. The final layout of the GUI is almost identical to the initial draft (see Section 3.4), because as I was implementing the various panels that make up the game window, I felt that there was no need to complicate the GUI any further; the Othello board can be easily seen, while important information such as the turn number and the player scores are also clearly visible.

There are some important changes made from the initial design[7]; spaces on the Othello board where a player can make a move are coloured in a slightly darker green than the other spaces, while a small triangle is displayed over the player who is currently taking their turn. Additionally, the number of turns that have passed is shown at the top of the screen, along with the phase of the game.[8] Finally, I titled the program "BlancheNoire", after the colours used for the counters in Othello.

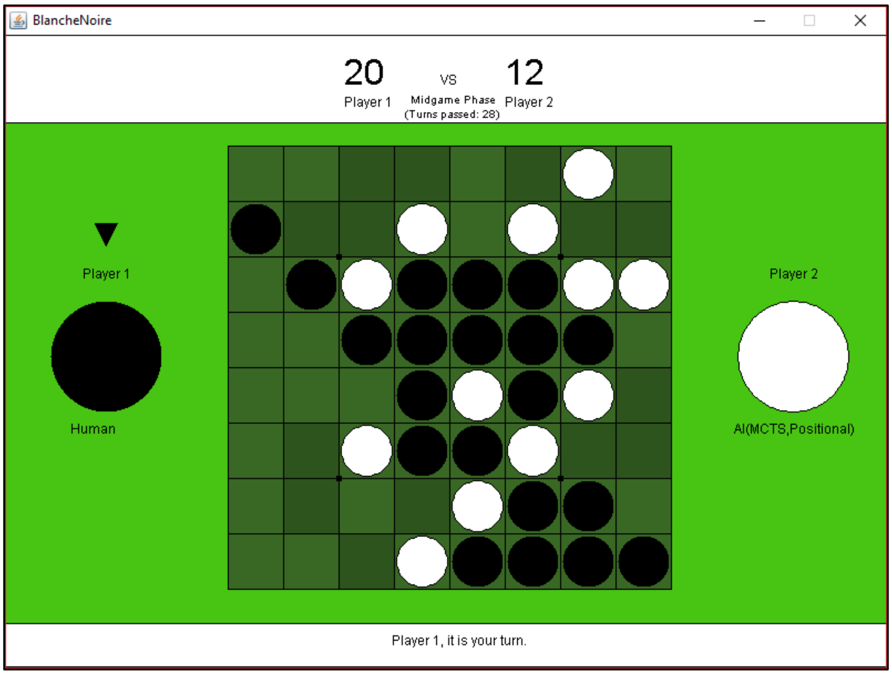
Internally, the only change from the design stage is that an extra Frame class called OthelloFrame was created to hold the three Panel objects. This allowed for functions such as the drawing and updating methods for each panel to be tied together into one method. Adding this class also allowed for easier implementation of the mouse input functions, which allow human players to click where they want to play a counter.

I encountered no major problems when creating the classes used by the GUI, due to the usability and accessibility of Java's interface classes.

---

[7] The initial design referred to in this section is found in Section 3.4.1.

[8] The phases shown are identical to the phases defined by Ted Landau in *Othello: Brief and Basic*; see Section 2.1.2 for information on phases.



Player 1, it is your turn.

Figure 13: Another screenshot of the Othello GUI in use, this time with a game being played by a human and an AI.

### 4.3.2 – Command Line Interfacing

One aspect that became more complex than I expected was the Othello program's ability to accept command line arguments to alter the behaviour of the game. As development progressed, more features needed to be changeable by a user at the launch of the program.

Since the AI players need to be composed from Decider and Evaluator classes, being able to change these classes – along with additional arguments that affect their behaviour – was a necessity. Furthermore, I wanted to provide the ability for other options to be changed, such as the visibility of the GUI, or whether games are archived to a text file upon completion. To do this, I added a section of code in the Othello class to parse through the arguments passed to the main method and store each of them, as long as they followed the format (-x y), so that an argument with the label x had the value y associated with it. Then, I added a check for each changeable value in the program, and altered the associated variables the corresponding argument label had been passed to the program.

The main problem with this system is that it requires arguments to follow a strict format, otherwise erroneous data can be passed to the program. I have validation checks in place to ensure that data is of the correct type and range, and if an argument cannot be accepted then the user will be notified. Additionally, player arguments are passed to the PlayerFactory class, which has numerous checks in place to ensure the correct Human or AI player is created.

Thanks to the flexibility of the program, performing tests and evaluation work on the system and its functionality has been easy to do. Though altering the program could be made easier, the current command line argument system is serviceable. A full list of the program's arguments is provided in Appendix D, and is also available in a text file alongside the main code of the program.

## 4.4 – AI Programming

Relevant packages: Deciders, Evaluators, Players

### 4.4.1 – Initial AI Setup and Testing

Creating the AIPlayer class was simple enough, as it was an extension of the Player class, that took two additional arguments for a Decider and Evaluator on construction. These would then be used when the Player's *decide* function was called without needing a different function call, so that the interface of an AI player would match that of all other players.

Before implementing any specific algorithms, I created abstract classes to define the interface that each Decider and Evaluator would have to use, so that no additional code needed to be built into the AIPlayer class to handle different classes and algorithms.

I started the Decider creation process by implementing a Random decider, that would simply select a random move from the legal moves available and would play it; as this decider did not need an evaluator to make its decisions, it allowed me to brute force test the game state and the game's GUI prior to implementing the first evaluator. Due to the short amount of time taken for two AI players using the decider to complete a game, I was able to run many games to test how well the GameState and GUI were implemented.

Following the successful tests using the Random decider, I moved onto implementing the Minimax decider. This was an important stepping stone between the Random decider and the MCTS algorithm for a few reasons; not only did it allow me to learn how to create an AI that analyses GameStates by looking numerous moves into the future, but it also provides a more suitable comparison for the MCTS algorithm than the Random decider would have.

Despite my intentions to only create one Minimax decider, an interesting issue with the way AIPlayer's are defined led me to create two variants of the same algorithm. When created, an AIPlayer is provided with a maximum time that it can spend determining the move it wants to make. Though this time is more of a guideline than a necessary limiter, the AI does need to begin halting its calculations once the time is up. For a traditional minimax algorithm, this would mean halting the algorithm without evaluating some moves, thus leading to potentially worse gameplay. This could be solved by iteratively deepening the search depth, while storing the results for shallower depths to be used in the event that the AI is forced to halt.

To that end, I created the Fixed Minimax and Iterative Minimax deciders, so that I could use the more appropriate one when necessary. The Fixed decider will immediately search to the maximum depth, but can be cut off and return poor evaluations of unexplored moves. On the other hand, the Iterative version will always return a somewhat accurate evaluation, but cannot search as far down a tree as the Fixed player can. The maximum depth either type of Minimax decider can search to is set to 6 moves into the future, but this can be changed via command line arguments. Though neither of these Deciders were necessary to create, the knowledge of Othello and Minimax AI behaviour I learned from developing them was incredibly useful for creating the MCTS decider.

To ensure that these Minimax deciders functioned correctly, I ran various tests to check that any AI players using a Minimax decider could competently play the game. During these tests, any players with an odd maximum depth would return poor move choices; by printing out debug information and examining the code, I determined that this was a result of the algorithm evaluating the game state with respect to whatever player was playing at the maximum depth. This resulted in players with oddnumbered max depths playing move that were beneficial to the opposing player instead of themselves. After fixing this error and running more tests, the operation of the Minimax deciders was completely fixed, and as a result I was confident that the implementation of these deciders was accurate.

Two evaluators were also created during this initial AI development stage; the Score evaluator provided a valuation on the game state depending on how much the player was beating their opponent in score, while the Positional evaluator uses a more complex calculation that assigns corner and edge pieces a higher value than other pieces on the board. The Score evaluator was mainly created as a placeholder, while the Positional evaluator represents an evaluation function based on the research I did into how humans determine the worth of a game state (see Section 2.1.2).

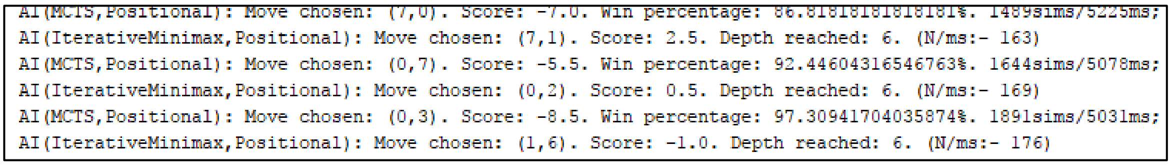
### 4.4.2 – Monte Carlo Tree Search

Once I had completed the Minimax deciders, I had all of the knowledge I needed to implement the Monte Carlo Tree Search algorithm as a decider class. The class was programmed to match the behaviour of the algorithm I had researched (see Section 2.4.2).

Initially, I created an inner class to be used to create the search tree while also storing the win/loss data, along with a simulation function for running playouts from a specified game state, and a method that created the whole search tree and ran the required simulations.

While creating these functions, I carried out tests to ensure that they functioned correctly, as it was easier to ensure the whole MCTS algorithm worked by first examining the functions it's made up of. As the simulation function can take an initial game state, apply legal moves to it without causing an error for playing illegal moves, and return the correct data based on the outcome, it was simple to confirm its functionality.

Since the algorithm also maintains a GameState representing the current tree node when creating and navigating the tree, the same errors that occur when illegal moves are played would be thrown when the tree incorrectly expanded. There were some initial tree navigation errors that I discovered and fixed, from the tree not factoring occasions when a player would have to pass their go, to altering the backpropagation algorithm to correctly update all affected nodes, but after these alterations to the code, the tree was correctly constructed and evaluated. Thankfully I didn't have to resort to visualising the constructed tree to ensure the structure and logic were correct, as that would have used a considerable amount of development time.



```
AI(MCTS,Positional): Move chosen: (7,0). Score: -7.0. Win percentage: 86.81818181818181%. 1489sims/5225ms;
AI(IterativeMinimax,Positional): Move chosen: (7,1). Score: 2.5. Depth reached: 6. (N/ms:- 163)
AI(MCTS,Positional): Move chosen: (0,7). Score: -5.5. Win percentage: 92.44604316546763%. 1644sims/5078ms;
AI(IterativeMinimax,Positional): Move chosen: (0,2). Score: 0.5. Depth reached: 6. (N/ms:- 169)
AI(MCTS,Positional): Move chosen: (0,3). Score: -8.5. Win percentage: 97.30941704035874%. 1891sims/5031ms;
AI(IterativeMinimax,Positional): Move chosen: (1,6). Score: -1.0. Depth reached: 6. (N/ms:- 176)
```

*Figure 14: The output produced by two deciders for each move they select.*

The outcome of the algorithm can be seen when the AI output is displayed during the Othello program's operation. The win percentage displayed by the MCTS decider shows how often the provided move led to a victory in the simulations that were run. Initially, I had programmed the AI to select the move with the highest number of total simulations instead, as I believed that it was a better measurement of the most promising move. I soon found out through some tests I ran that selecting the move to play based on the win percentage results in a more proficient AI.

However, at this stage I discovered that the MCTS performs poorly if it cannot run enough simulations to gather the necessary data, which can occur on weaker computers. To combat this, I implemented some optimisations to improve search times, such as the previously mentioned data caching in the GameState class (Section 4.2.2), which helped to improve the decider's ability to run on slower machines.

One I had completed the initial implementation, I also realised that there were numerous values in the MCTS algorithm that could be changed to affect the algorithm, such as the simulation type. I allowed additional arguments to be provided to the MCTS' constructor to alter its behaviour, along with enabling users to set these values via the command line argument system. This adds to the customisability of the AI system, while also allowing for different variants of the AI to be tested against each other, which was beneficial in evaluating the final results of the project. The following MCTS settings can be changed:

- Simulation Type – The types of simulations that the MCTS uses to construct its search tree can be changed to use heavy simulations (simulations use the move chosen by minimax from each state to determine how the game would play out), or can be left as light simulations (simulations select a random move from the current state to play the game).[9]
- Maximum Number of Simulations – The number of simulations used by a MCTS decider can be capped, to decrease the amount of time required for the decider to make a move.
- Random Move Chance – To add a small element of randomness to heavy simulations, a value between 0 and 1 can be used to determine how often the simulation should ignore the move chosen by minimax and use an “imperfect” random move.
- Minimax Quality – The two arguments for minimax depth and the time to spend minimaxing are only used when the MCTS decider is set to use heavy simulations. They can be changed to increase the accuracy of the produced simulations, and the expense of the number of them produced, or vice versa.

Detailed information on how these arguments change the operation of the program can be found in Appendix D.

Overall, programming the MCTS was quite a challenge due to the amount of debugging I carried out to ensure that the system was working correctly, from printing out all the legal moves and their victory percentages to ensure that the right move was being selected, to running the AI with test GameStates to check how it determined what move to play in various situations. The result of this process is a competent decider algorithm that can extensively search the game space, and provide the optimal move to play without a need for the game's rules to be explicitly encoded to it.

---

[9] Pseudocode for the functionality of these types of simulations can be found in Section 3.5.3.

## 4.5 – Deep Neural Network Development

Relevant packages: Learning, Players, Games, Util

### 4.5.1 – Utilising the Tournament Dataset

An important ingredient for any Deep Learning process is to acquire enough data for the ANN to analyse and train itself with. As Othello is not a very popular board game, I was initially worried about my chances of finding a suitable dataset, but I eventually discovered the French Federation of Othello website[10], which keeps a large database of over 100,000 Othello games from various tournaments that is free for public use, which was more than enough data for creating an ANN.

However, once I downloaded the games I noticed they were stored in a file format I had never encountered before. The .wtb file format that the games came in could only be opened using Win-Test[11], which is a piece of software used to log tournament results. By downloading the necessary files and a free copy of the Win-Test software, I was able to open the game databases for each year of competition.

However, Win-Test did not provide any tools for extracting the sequences of moves and final scores from the database, which led to me having to manually copy each tournament table into a text editor, strip away the unnecessary data, and save the resulting list of games. Additionally, I had to find an older version of Win-Test for some database files, as they were not supported by the newer version. Furthermore, the trial version of Win-Test encourages you to purchase the full version by randomly *terminating the program* at any point after half an hour of use. Thankfully, I only needed to use the software for a short time, and I eventually converted the database tables into a usable data format.

The next step was to load these games into a program and reformat them into data for the deep learning process. I began by creating a new class called the NeuralNetDataHandler for this purpose, and wrote some file reading and writing commands into a class called FileTools, as I expected to make many calls to the file system over the data formatting process.

The NeuralNetDataHandler begins by converting the data extracted from the database into a standard Othello game format I have created, which is comprised of a list of the coordinates of the moves made, followed by the final scores of the games, and the dimensions of the game board. I chose not to provide any information about which player carried out each move, as I felt that my implementation of the game logic should be able to correctly identify the player without the information.

Once the games are converted into the standard format, they are loaded into GameScript objects for validation. The GameScript class was created to allow any archived game written in the format mentioned above to be examined and re-simulated. It stores the list of moves made, and can create a GameState from any point in the game. Its primary use in the NeuralNetDataHandler code is to simulate each game from the tournament database to ensure that the GameState logic has been implemented correctly, and to remove any erroneous games.[12]

Once the games are validated, they are split into training and testing sets. The training set of games is used when constructing the ANN, while the testing set is used after the construction to evaluate the network's accuracy.

---

[10] Available at: http://www.ffothello.org/informatique/la-base-wthor/

[11] Official website: http://www.win-test.com/

[12] For more details on the validation of the rules in the GameState class, see Section 4.2.3.

The final step of the NeuralNetDataHandler is to iterate through each game in the training/testing data sets, split them into the GameStates that they are composed of, and create a binary representation of them. The representation I chose for the GameStates is a string of 128 bits, with a 1 representing the presence of a counter, while a 0 represents an empty space; the first 64 bits are used to show the presence of dark counters, while the last 64 bits are used for light counter representation. The deep learning process requires data to be in this format as it tells it which nodes in the input layer of the ANN should be activated or not.

I also chose to invert the board state to create additional data for the learning process, as any Othello game state can be inverted (i.e. dark counters are inverted to light counters, and vice versa) to create another legal Othello state.

When writing the data to the final csv files, the program must also provide a label to each binary representation of a game state to allow it to be used to train or test an ANN. However, as I wanted to explore what type of labelling was more effective, I created two different label formats, which are referred to by the NeuralNetDataHandler as Data Format 2 (DF2) and Data Format 3 (DF3).[13] DF2 labels each piece of game state data with a 1 if the player controlling the dark counters won the game the state belongs to, or a 0 if they did not. DF3 tracks the number of times each state appears, and stores the win/loss ratio for the dark counter player in each case. The win percentages for each state that appeared are then calculated, and the label applied to each state is based on the bin that the win percentage fell into; the number of bins created in this process is equal to the ceiling value of the average number of times a state appears in the data set, plus 1.

The only problem I encountered with creating the necessary data was that my computer struggled to hold all the games in memory during the process; though I was able to increase Java's maximum heap size to solve the problem, I also implemented various measures to minimise the amount of data used by the process by clearing unnecessary objects out of memory with manual calls to Java's garbage collector.

The NeuralNetDataHandler class can be run from the command line, and requires various arguments to specify the datasets it uses, the functions carried out, etc. It should be run before constructing an ANN so that the data is adequately prepared.

### 4.5.2 – Creating an Artificial Neural Network

With the DL4J framework installed[14] and the data formatted, I could begin work on creating a neural network construction class. As it is completely infeasible to build and train an ANN at launch of an Othello game, an additional class was needed train and test the ANN, along with saving the network to a file for later use. The NeuralNetFactory class was created to serve this purpose; the class allows for the sources of training and testing data, the network structure to use, and operation types to be altered with command line arguments, and it saved finished ANNs in a ZIP file, along with a statistics file for each network.

The ANN construction process begins with the training data set being loaded in from the provided training directory; the data set is split into numerous CSV files due to the number of records, so a

---

[13] Data Format 1 will not be discussed in this report, as it was ultimately unsuccessful at producing the correct label data.

[14] See Section 3.2 for more information on the DL4J framework.

temporary merged file of all of them is created. This file is then loaded into a DL4J DataSetIterator, which allows it to be used by the neural network during training.

After this, based on one of the program arguments, a neural network configuration is selected; a configuration consists of a series of settings for the different attributes of a network, from individual values such as the learning rate and the optimisation algorithm, to the layout and types of hidden layers. This is then used to create a full neural network in memory, that is then given the training data to learn from. The training continues until the required number of epochs (the number of passes through the training data) have been carried out.

After the training phase, the testing data is loaded in and applied to the network. This stage tests how often the created neural network can accurately predict the labels of the testing data, given the knowledge it has learnt from the training data. The statistics from the testing process are saved to a text file after the process, which provides details on how many times the network returned correct and incorrect readings, how far off the correct answer it was each time, etc.

With the training and testing complete, all that remains is to write the network to the ZIP file, which can be done via a built-in function for the ANNs produced by DL4J's framework. This storage method allows for easy saving and loading of the networks, with no change to the internal structure after re-loading a saved network.

### 4.5.3 – Combining an ANN with the DeepLearning Evaluator

```
// Get output from the ANN.
INDArray result = net.output(game.toINDArray(p.getPlayerID(),
                              game.getOpposingPlayer(p).getPlayerID()), false);
int numLabels = result.shape()[1];
if (numLabels > 1) {
   // Creates a composite score by multiplying the label by the probability that it
is that label.
   float weightedScore = 0;
   for (int i = 1; i < numLabels; ++i) {
      weightedScore += (result.getFloat(0, i) * i);
   }
   return (weightedScore * (100/(numLabels-1)));
} else {
   return (result.getFloat(0, 0) * (100));
}
```

*Figure 15: The code used to query the ANN and return a score from the Evaluator's evaluate() function. The function returns a score between 0 and 100 based on its confidence that the player can win from the game state.*

The finalised ANNs can be used by the DeepLearningEvaluator class, which as the name suggests can be used as an evaluator for an AI player. The class is instantiated by providing it with the file path of a saved ANN file, which is then loaded into the DeepLearningEvaluator object. Any calls to the evaluate function of the DeepLearningEvaluator convert the provided GameState to a ND4J array[15], which is passed to the ANN to evaluate.

Once the ANN finishes its evaluation and returns its values, the Deep Learning (DL) evaluator then determines how to return the result, by using the segment of code shown in Figure 15. The DL4J ANNs can return multiple scalar values to show how likely it is that the provided data belongs to a specified class if a classification network is used, so the DL evaluator then computes a composite score from each of the classification values. The evaluator also supports regression networks, which only return

---

[15] The ND4J (N-Dimensional Arrays for Java) package is part of the DL4J framework.

one scalar value from their ANN operation; as a result, this value can be directly returned from the DL evaluator.

### 4.5.4 – The Mass Network Production Phase

With the NeuralNetFactory complete, I could move into the experimenting phase; I quickly set out to make various neural networks using different parameters and data to determine how to create the most effective ANN. However, I soon discovered that the main challenge in creating intelligent ANNs arises from two places; choosing the data format to use, and determining the best structure for the neural network to use.

As previously mentioned, I created two reliable datasets using the NeuralNetDataHandler, which solves the first of the problems I faced. On the other hand, DL4J offers a massive library of features to create and alter the neural network structure, and as a result I had to quickly learn how to create an effective network structure for the data I had; the network had to have 128 input nodes for the 128-bit GameState representation, and had to have output nodes equal to the number of unique labels in the data. Every other value was left to me to decide, but with my lack of knowledge in this area, this proved quite difficult.

The final networks produced from the network structures I created are likely far from optimal, but I was able to create network structures I am confident in thanks to an article written by a developer called Erik Bernhardsson about his experience with developing a neural network for playing chess, which is somewhat similar to my project. In it, he discussed the various layouts used when creating his network, including a "3 layer, deep 2048 units wide artificial neural network" [19], which he used as the number of links between the various layers allows for better accuracy in the network's evaluations of the provided game states. I have adapted a similar network structure to provide the ANN with thousands of additional links between nodes, that should in theory improve the accuracy of said networks. I chose to utilise a similar network structure to the one Bernhardsson discussed due to the similarities between the applications of the network, and because it appeared that a wider network provided more opportunities to finely tune the network. I have also used smaller networks to test how the decision process is effected by network width and depth.

Finally, I also had to decide which of the major deep learning approaches I was going to use; classification or regression. Both can produce results that the DeepLearningEvaluator can use, and regression seemed to be the best fit because it produced a scalar value, similarly to the other Evaluator classes. However, because the data I had gathered was better suited to a classification-based network, and that I was unable to create any regression-based networks due to difficulties with using the associated DL4J libraries, I shifted my focus to only producing classification networks.

Despite the setback of not being able to create regression ANNs, the results of the network creation process have been promising; ANNs can now easily be created for use with the Othello program by any user, and are correctly loaded into and used by the Evaluator class. The speed of said Evaluator is also similar to existing evaluation classes, which allows it to be used alongside Minimax and MCTS deciders without drastically impacting their runtimes.

# Section 5 – Evaluating the AI's Potential

In this section, I will be discussing the results of the various tests I carried out on my Othello-playing AI to determine the extent of its game playing capabilities. As I have already confirmed the accuracy of my Othello implementation (Section 4.2.3), and that the AI decision algorithms are implemented accurately (Section 4.4), the following experiments will focus solely on evaluating the game-playing ability of the various AI classes and ANNs I have created.

For all the experiments run in this section, I used my Othello's programs arguments system to automatically run multiple games with the same settings, allowing for easy collection of the necessary data. To remove any bias from the data because of one player always starting first in each game, I also enabled the alternation setting of the program, which swaps the player that takes their go first in each game.

The hardware and software of the computer I used for these evaluations is as follows:

- Processor: Intel Core i5-4690K @ 3.5GHz
- Graphics Card: NVidia GeForce GTX 970
- RAM: 16GB of DDR3 1666MHz RAM
- OS: Windows 10
- Java version: v1.8.0_131
- DL4J version: v0.8.0

## 5.1 – AI vs AI Evaluation

To begin the evaluation process, I first decided to evaluate the decider classes against each other to discover how capable their decision abilities were. This meant running experiments using the following Deciders:

- RandomDecider
- FixedMinimaxDecider
- IterativeMinimaxDecider
- MonteCarloTreeSearchDecider

The Positional evaluation function was used in these experiments, to ensure each AI used the same game state evaluation, so that the performance of the Decider was the deciding factor between the AI players.

### 5.1.1 – Experiment 1: Random vs Fixed Minimax

As a preliminary experiment, I compared the performance of the intelligent Fixed Minimax algorithm against the basic Random move selection algorithm.

**Results:**

| **Experiment 1: Random vs Minimax** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | Random | FixedMinimax |
| Evaluator | | | | Positional | Positional |
| Victories | | | | 4 | **45** |
| Win Percent | | | | 8% | **90%** |
| Total Games | 50 | Draws | 1 | | |

**Notes:** The FixedMinimaxDecider's maximum search depth was set to the default of 6.

As expected, the Fixed Minimax player was able to outplay the Random move player in 90% of games; the Random decider lacks any structure or reasoning to its chosen moves, and likely only outplayed the Minimax player in the other games due to sheer luck.

### 5.1.2 – Experiment 2: Fixed Minimax vs Iterative Minimax

To evaluate the performance of both Minimax deciders, I carried out two different experiments to determine how the behaviour of the algorithm changed as the search depth was increased.

**Results:**

| **Experiment 2A: Fixed Minimax vs Iterative Minimax (Depth of 8)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | FixedMinimax | IterativeMinimax |
| Evaluator | | | | Positional | Positional |
| Victories | | | | **29** | 10 |
| Win Percent | | | | **72.5%** | 25% |
| Total Games | 40 | Draws | 1 | | |

**Notes:** Maximum search depth for the FixedMinimax and IterativeMinimax deciders was set to 8.

In this test, the Fixed variant of the Minimax algorithm won 72.5% of the games, while the Iterative variant won only 25% of the time. I chose the depth of 8 as it allows for the deciders to examine many moves, while also ensuring they complete their minimaxing operations within the 5 second time limit that AI player must choose a move in.[16]

**Results:**

| **Experiment 2B: Fixed Minimax vs Iterative Minimax (Depth of 12)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | FixedMinimax | IterativeMinimax |
| Evaluator | | | | Positional | Positional |
| Victories | | | | **32** | 8 |
| Win Percent | | | | **80%** | 20% |
| Total Games | 40 | Draws | 0 | | |

**Notes:** Maximum search depth for the FixedMinimax and IterativeMinimax deciders was set to 12.

---

[16] For more information on the operation of the Minimax algorithms, see Section 4.4.1.

For the other main test on the Minimax deciders, I set the maximum depth of the deciders to 12, so that they would most likely be unable to complete their minimaxing within the time limit, due to the number of game states to search. Due to this time cut off, I expected the FixedMinimax to make many poor decisions leading to it losing numerous games. However, in the end the Fixed variant of the Minimax decider won 80% of the games, thus achieving a larger win margin than the smaller depth test.

As the IterativeMinimax algorithm repeats many of its minimax calculations when increasing its search depth (since it lacks a transposition table for storing prior results), it wastes a lot of its potential search time. On the other hand, the FixedMinimax algorithm searches directly to the maximum depth when examining a move; even though it will not be able to evaluate every possible move, the ones that are evaluated will be more informed about the future game state than any evaluation the IterativeMinimax algorithm could produce. Additionally, as the end of the game approaches, the maximum searchable depth decreases, and since the Fixed algorithm searches through states depth first, it will take advantage of this smaller search space to improve its calculations before the Iterative algorithm can. The endgame phase of Othello is where many of the most important moves are made, so it’s likely that the Fixed decider’s improved foresight at the game’s end allowed it to improve its victory chances.

To analyse the effect of changing the depth at lower values, I ran more tests with varying depth values, and graphed the results:

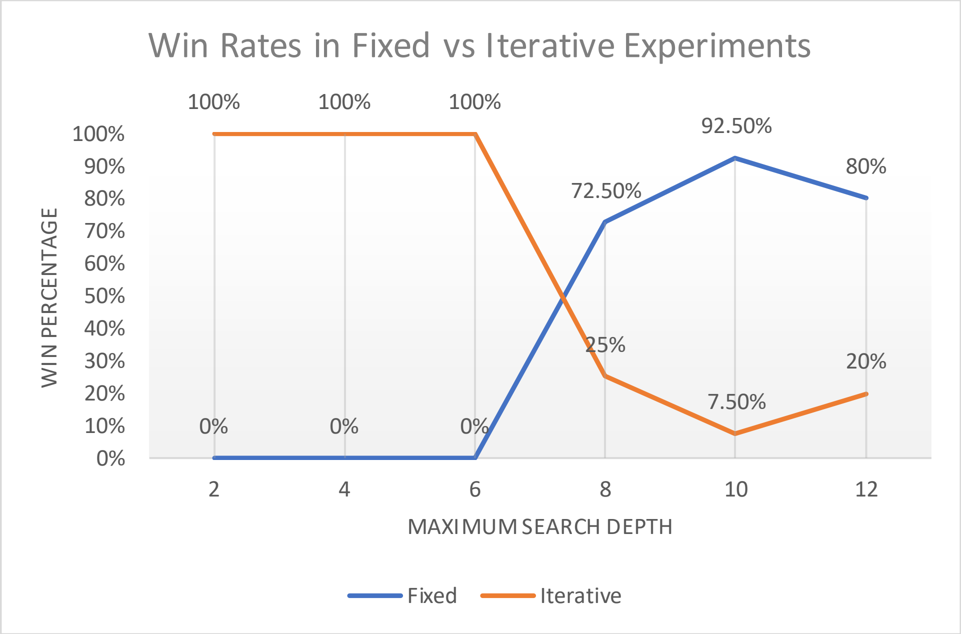


*Figure 16: A graph showing the win rates of the Fixed and Iterative minimax deciders as the maximum search depth varies.*

Overall, the IterativeMinimax decider is more effective at lower depths, likely because the deciders’ evaluations are not affected by the time cut off whatsoever, plus since the minimax players have no randomness to them, the same games are played repeatedly, resulting to the 100% win rate for the Iterative decider. However, as the maximum search depth is increased, the Iterative algorithm’s performance decreases, and the Fixed variant is able to defeat it more and more often.

For future tests requiring a comparison with a Minimax algorithm, I decided to use the FixedMinimax decider, since the experiments required giving the Minimax decider a deep search depth to compete with the other deciders.

### 5.1.3 – Experiment 3: Fixed Minimax vs Monte Carlo Tree Search

The final AI type to evaluate was the Monte Carlo Tree Search decider; I conducted two tests by playing it against an AI player using the FixedMinimax decider. For each test, I set the MCTS decider to use a different method of running the simulations that are uses to build its search tree, to evaluate how that impacted its performance.

**Results:**

| **Experiment 3A: Minimax vs Monte Carlo (Heavy Simulations)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | FixedMinimax | MCTS |
| Evaluator | | | | Positional | Positional |
| Victories | | | | 3 | **17** |
| Win Percent | | | | 15% | **85%** |
| Total Games | 20 | Draws | 0 | | |

**Notes:** Iterative Minimax decider had a maximum search depth of 8. MCTS decider was instructed to run heavy simulations (i.e. using minimax to determine moves in the simulations it ran) to a depth of three at each move. It also had a random simulation move chance of 0.01, a minimax time of 5, and a maximum simulation count of 10,000.[17]

This test had the MCTS decider use the more intelligent way of running simulations, as it used a minimax algorithm to predict how the games would progress within the simulations it ran.[18] The MCTS also takes an argument for a probability value, which determines how often a random move is chosen as the next step of the simulation instead of the optimal minimax-selected move; for this experiment, it was set to 0.01, so that some variance in the moves used occurred. The Minimax algorithm used within the simulation was also given a depth that allowed it to evaluate states a few moves into the future, so that it could compete with the MCTS algorithm.

The results of the experiment are quite clear; the MCTS decider defeated the Minimax decider in 85% of the games that were run, which was one of the largest victory margins I had seen at this point in the evaluation process. Clearly, the MCTS algorithms ability to choose moves based on their probability of victory can outwit the Minimax's logical approach to selecting moves.

**Results:**

| **Experiment 3B: Minimax vs Monte Carlo (Light Simulations)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | FixedMinimax | MCTS |
| Evaluator | | | | Positional | Positional |
| Victories | | | | 6 | **14** |
| Win Percent | | | | 30% | **70%** |
| Total Games | 20 | Draws | 0 | | |

---

[17] These values are arguments for constructing the MCTS decider. For more information, see Section 4.4.2 and Appendix D, Subsection 3.

[18] For more information on the types of simulations available, see Section 4.4.2 or Appendix D, Subsection 3.

**Notes:** Iterative Minimax decider had a maximum search depth of 8. MCTS decider was instructed to run light simulations (i.e. using random to determine moves in the simulations it ran). It was also given a maximum simulation number of 10,000.

Meanwhile, the other experiment I ran using this matchup used a MCTS decider with a less intelligent simulation method; each move in the simulations run by this MCTS algorithm used completely random moves instead of ones chosen via Minimaxing. This makes each individual simulation less accurate, as the randomly chosen moves may never be considered by an intelligent opponent, but allows for more simulations to be run, thus balancing out the lack of an informed move selection with a large sample size.

This variant of the Monte Carlo Tree Search was also able to handily defeat the Minimax decider, with a 70% success rate. Ultimately it appears that the lowered accuracy of the simulations does affect the MCTS algorithm's effectiveness, as the win margin in this test is notably smaller than the previous test.

### 5.1.4 – Experiment 4: Monte Carlo Tree Search Comparisons (Simulation Type)

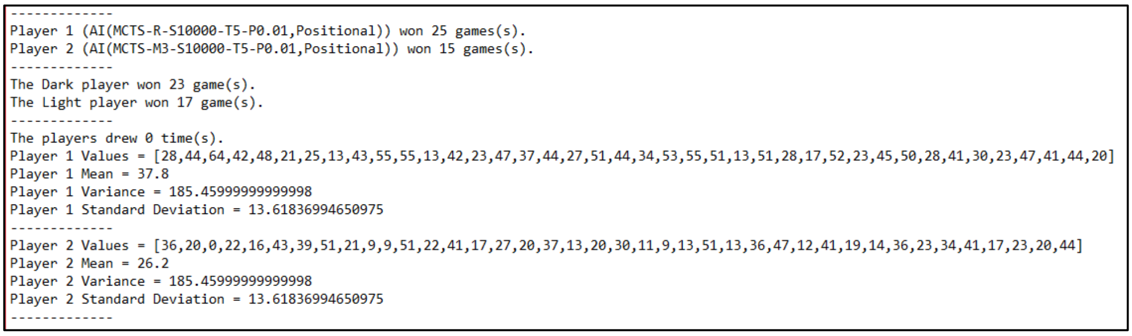
As a follow-up experiment to the previously experiment, I ran a test to evaluate which method of simulating games used by the MCTS results in the best performance against another MCTS decider.

**Results:**

<table>
<tr><th colspan="4">Experiment 4: MCTS Comparisons (Simulations Type)</th><th>Player 1</th><th>Player 2</th></tr>
<tr><td colspan="4">Decider</td><td>MCTS (Random)</td><td>MCTS (Minimax)</td></tr>
<tr><td colspan="4">Evaluator</td><td>Positional</td><td>Positional</td></tr>
<tr><td colspan="4">Victories</td><td>**25**</td><td>15</td></tr>
<tr><td colspan="4">Win Percent</td><td>**62.5%**</td><td>37.5%</td></tr>
<tr><td>Total Games</td><td>40</td><td>Draws</td><td>0</td><td></td><td></td></tr>
</table>

**Notes:** Player 1's MCTS decider was instructed to run heavy simulations (i.e. using minimax to determine moves in the simulations it ran) to a depth of three at each move. It also had a random simulation move chance of 0.01, a minimax time of 5, and a maximum simulation count of 10,000. Player 2's MCTS decider was instructed to run light simulations (i.e. using random to determine moves in the simulations it ran). It was also given a maximum simulation number of 10,000.

Surprisingly, the less intelligent approach of using light simulations to construct the search tree was able to win more games than the heavy simulation method. I had initially expected the heavy simulations to perform the best out of the two simulation types, but the results showed that it wasn't the case.



```
-------------
Player 1 (AI(MCTS-R-S10000-T5-P0.01,Positional)) won 25 games(s).
Player 2 (AI(MCTS-M3-S10000-T5-P0.01,Positional)) won 15 games(s).
-------------
The Dark player won 23 game(s).
The Light player won 17 game(s).
-------------
The players drew 0 time(s).
Player 1 Values = [28,44,64,42,48,21,25,13,43,55,55,13,42,23,47,37,44,27,51,44,34,53,55,51,13,51,28,17,52,23,45,50,28,41,30,23,47,41,44,20]
Player 1 Mean = 37.8
Player 1 Variance = 185.45999999999998
Player 1 Standard Deviation = 13.61836994650975
-------------
Player 2 Values = [36,20,0,22,16,43,39,51,21,9,9,51,22,41,17,27,20,37,13,20,30,11,9,13,51,13,36,47,12,41,19,14,36,23,34,41,17,23,20,44]
Player 2 Mean = 26.2
Player 2 Variance = 185.45999999999998
Player 2 Standard Deviation = 13.61836994650975
-------------
```

*Figure 17: The statistics file produced for Experiment 4. The file tracks the number of wins per player, the number of wins per counter colour, a score list, and other additional statistics.*

To explore this matter further, I examined the statistics that my Othello program produced once the games had been run. In this experiment, the difference between the two players' average scores was 11.6; though this shows that Player 1 did have a higher average score, the difference between the two players scores is smaller I expected, given the difference in number of games won. For example, in Experiment 1 the score difference was 35.9, while in Experiment 3B the difference was 16.7 – a much larger difference despite the two experiments sharing a similar win rate. This likely means that many of the games played by these two AI players were very close fought matches.

Despite this, it is clear that a MCTS decider using light simulations can handily defeat one that uses heavy simulations. I believe the reason for this is simple; as the light simulations use random moves to predict how a game will end, they will likely explore much more of the game state than the heavy simulations will, as the Minimax method will likely keep the heavy simulations exploration locked to similar routes to the end of the game, even as the tree expands. Thus, the light simulations better summarise all possible outcomes for that move, whereas the heavy simulations model the victory chances when the optimal route is followed. Additionally, the MCTS decider using the heavy simulations predicts the path that an optimal player will take; however, in this experiment the MCTS decider using light simulations would likely never take that route, resulting in the heavy simulation MCTS player being unable to accurately predict the opponents moves.

Overall, though the heavy simulation MCTS decider was beaten by a large margin, this is likely because the weaknesses of the heavy simulation approach are exploited by the light simulation approach, and both ways of running simulations still perform very well as AI deciders.

### 5.1.5 – Experiment 5: Monte Carlo Tree Search Comparisons (Varying Probability)

As mentioned previously in the report, one of my goals for the Othello system was to allow users to customise the level of difficulty of the AIs they play against. I decided to evaluate the random move chance of the MCTS algorithm next, as it appeared to be a good way to alter the AI's difficulty. With the results from this test, I would be able to assess the extent to which a user can customise the AI players.

**Results:**

| **Experiment 5A: MCTS Comparisons (Probability)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | MCTS (P = 0.00) | MCTS (P = 0.02) |
| Evaluator | | | | Positional | Positional |
| Victories | | | | 8 | **12** |
| Win Percent | | | | 40% | **60%** |
| Total Games | 20 | Draws | 0 | | |

| **Experiment 5B: MCTS Comparisons (Probability)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | MCTS (P = 0.00) | MCTS (P = 0.01) |
| Evaluator | | | | Positional | Positional |
| Victories | | | | 7 | **13** |
| Win Percent | | | | 35% | **65%** |
| Total Games | 20 | Draws | 0 | | |

| **Experiment 5C: MCTS Comparisons (Probability)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | MCTS (P = 0.01) | MCTS (P = 0.02) |
| Evaluator | | | | Positional | Positional |
| Victories | | | | **14** | 6 |
| Win Percent | | | | **70%** | 30% |
| Total Games | 20 | Draws | 0 | | |

| **Experiment 5D: MCTS Comparisons (Probability)** | | | | Player 1 | Player 2 |
|---|---|---|---|---|---|
| Decider | | | | MCTS (P = 0.01) | MCTS (P = 0.10) |
| Evaluator | | | | Positional | Positional |
| Victories | | | | 8 | **12** |
| Win Percent | | | | 40% | **60%** |
| Total Games | 20 | Draws | 0 | | |

**Notes:** Both players' MCTS deciders were given the same arguments, except for their random move chance; run heavy simulations to a depth of three, minimax time of 5, and a maximum simulation count of 10,000.

In this experiment, I used 4 different probability levels to view the effects that changing the random move chance had. Oddly enough, setting the probability to zero, which made the MCTS algorithm rely solely on minimaxing for its simulations, resulted in poor performance in comparison to the deciders that allowed for some randomness.

The reason for this is tied to the nature of the Minimax algorithm; as the minimaxing functions select the most optimal move (according to its evaluator), there will be no variance in the move selected from a game state, no matter how frequently the same state appears. This means many games can experience no variance in tactics, due to the fact there is no chance of different games playing out. As a result, when a MCTS decider has a small chance to play a random move instead of a minimaxed move during its heavy simulations, this allows for a larger variety of moves to be simulated, thus improving the algorithms prediction ability, and by extension its move decision ability.[19] As a result, it appears to be beneficial to the MCTS algorithm to introduce at least some randomness to the simulation process.

On the other hand, the difference that further increasing the random probability has on the results is difficult to determine; since increasing this chance alters the simulation process to be similar to light simulations, passing a larger value to the MCTS for this parameter can be beneficial if facing a very logical opponent, while keeping it on the lower end of the scale can provide more accurate simulations to defeat a less intelligent player. It seems that the optimal value for this parameter is different depending on the player that the MCTS algorithm is facing.

Regardless, these results do indicate that the overall difficulty of the AI player is affected by the change in random move chance, as expected.

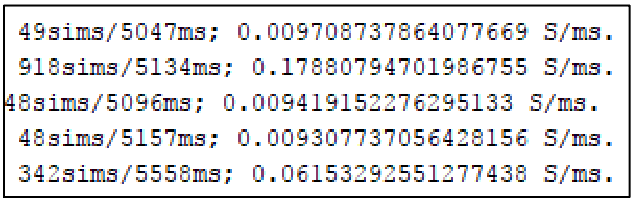
### 5.1.6 – Experiment 6: Investigating the Remaining MCTS parameters

At this point in the evaluation process, the only two MCTS parameters left untested were the heavy simulation minimax depth and the time per simulation minimax run. These two variables are closely related to how accurate the heavy simulations are, and how many of these simulations can be run in

---

[19] Further information on the random move chance's changes to simulation behaviour can be found in Section 4.4.2 or Appendix D.

the timeframe given to the decider.[20] As a result, I performed some developmental tests while creating the MCTS to determine a combination of the two variables to carry out the first set of experiments with.



```
 49sims/5047ms; 0.009708737864077669 S/ms.
 918sims/5134ms; 0.17880794701986755 S/ms.
48sims/5096ms; 0.009419152276295133 S/ms.
 48sims/5157ms; 0.009307737056428156 S/ms.
 342sims/5558ms; 0.06153292551277438 S/ms.
```

*Figure 18: A section of the typical console output of the MCTS algorithm while using heavy simulations. Each line shows the number of simulations per move chosen, the time taken to run all the simulations, and then the number of simulations per millisecond.*

To analyse how these variables affect the MCTS algorithm, I examined the detailed output of a typical Othello game between two AI players using MCTS deciders. With the minimax simulation depth of 3 and a time per minimax of 5ms, the above information is displayed; as these are the outputs of the first five moves between two MCTS deciders, the search space is at its largest, and at a minimum the algorithm can run approximately 48 simulations. Though this appears to be a low number, it is enough to provide an accurate prediction of the potential moves' impact, as at this point in the game there are typically few moves to choose from the starting states. As the game progresses, more simulations can be run, due to the decreasing number of possible states.

I examined the minimum number of simulations that the MCTS decider could run when different combinations for the heavy simulation search depth and time were used:

| **Experiment 6: Remaining MCTS Parameter Effectiveness** | | Minimax Search Depth | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 |
| Time per Minimax | T = 2ms | 1796 | 178 | 73 | 54 | 52 | 50 |
| | T = 5ms | 1536 | 175 | 48 | 24 | 24 | 23 |
| | T = 10ms | 1542 | 159 | 39 | 15 | 12 | 11 |

---

[20] Details on how these arguments affect the MCTS algorithm can be found in Appendix D.

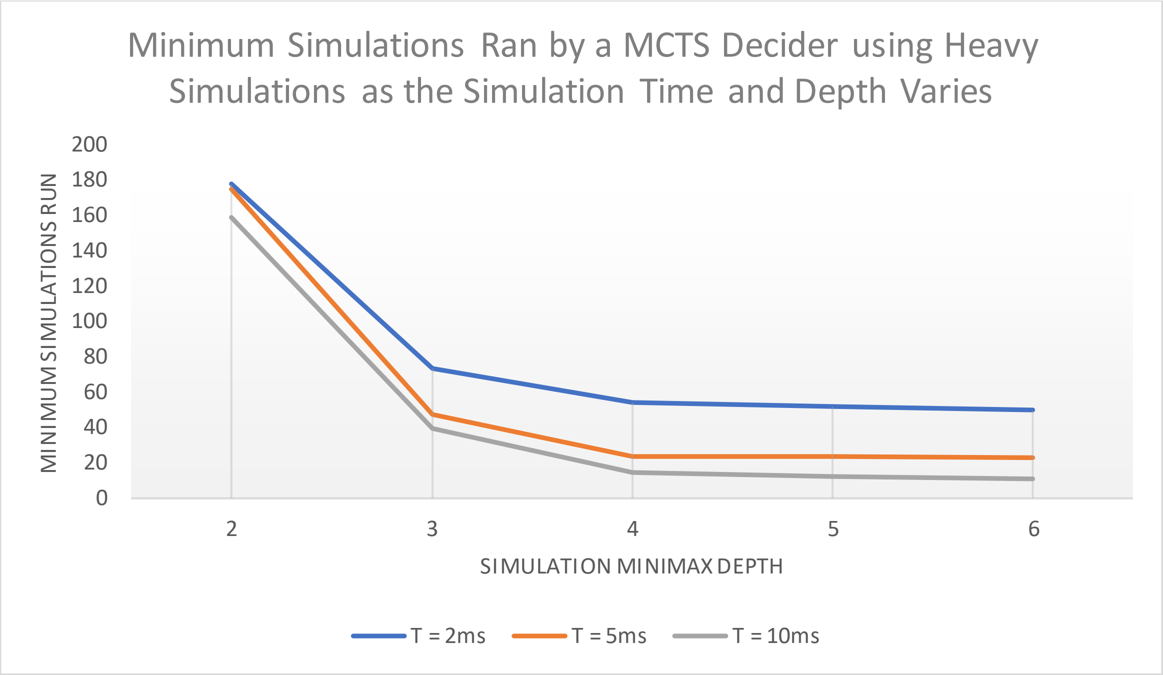


*Figure 19: A graph showing the effect that varying the search depth and time allowed for heavy simulations has on the minimum number of simulations a MCTS decider can run.*

My findings show that the higher the amount of time provided for the MCTS decider to run the minimaxing within the simulations, the less total simulations are run. Similarly, less simulations can be run when the simulation minimaxing is instructed to search deeper.

The logical conclusion from this would be to only allow heavy simulations to minimax to a shallow depth; however, as the depth of the minimax search affects the quality of the MCTS' predictions, the depth should be set as high as the computer's hardware can manage while still producing a useable number of simulations for the MCTS algorithm to evaluate.

```
if (current.isOver() || depth == 0 ||
System.currentTimeMillis() - startTimestamp >= timeLimit) {

   return e.evaluate(current, playerToEvaluate);

}
```

*Figure 20: Section of code from the minimax algorithm that determines what to do when the time to minimax is exceeded. This code is used by both Minimax deciders, and by the heavy simulation system of the MCTS decider.*

Additionally, the minimaxing code is set to immediately return an evaluation of a node once the time limit has been reached, rather than recursively searching deeper. This means that even though using a shorter amount of time produces more simulations and thus more data, it is also likely that the simulations' view of how the games will progress will be weakened.

Overall, a balance between these two variable needs to be struck to ensure that enough data is produced for the MCTS decider while also keeping this data accurate and useful.

### 5.1.7 –Summary

Overall, the outcomes of the tests carried out in this section met my expectations; given the way each decider functions, I expected the MCTS to be the most competent, followed by the Minimax deciders. The lack of many surprising results also suggests the deciders' algorithms are working as intended.

However, I did not expect to go as in depth as I did when examining the MCTS various parameters and simulation modes. I had already observed some trends that the decider exhibited when developing it (hence why I used the preset values for some MCTS arguments), but the outcome of some experiments warranted further exploration of the effects the arguments had on the performance of the decider.

With all the data I have collected on the operation of the MCTS algorithm I have implemented, I can say that I am very pleased with its overall performance against other AI players in Othello. As it appeared to be the most competent decision algorithm of the ones that I researched, I had high hopes for its performance, which it was able to fulfil.

However, these tests only examined the performance of the deciders against other AI players. For the next stage of experiments, I decided to evaluate the performance of the AI classes against a different kind of opponent.

## 5.2 –AI vs Human Evaluation

The following section contains information on the tests I ran to compare the AI's capabilities against human players. To generate the data I needed, I found 6 volunteers (including myself) to play 4 games against each AI iteration. The volunteers were provided with the Othello ruleset to learn the rules, and the list of 21 Principles from Othello: Brief and Basic[21], but otherwise had no additional information. The GUI was enabled for each test, but the console output was disabled so that no player could view information on how the AI was thinking or playing. This ensured a series of bias-free games were played.

### 5.2.1 – Experiment 7: Human Player vs Minimax Player

The first Human tests I ran involved having a player compete against an AI opponent using the FixedMinimax decider.

**Results:**

<table>
<tr><th colspan="4">Experiment 7: Human vs Minimax</th><th>Player 1</th><th>Player 2</th></tr>
<tr><td colspan="4">Decider</td><td>Human</td><td>FixedMinimax</td></tr>
<tr><td colspan="4">Evaluator</td><td>-</td><td>Positional</td></tr>
<tr><td colspan="4">Victories</td><td>4</td><td><b>20</b></td></tr>
<tr><td colspan="4">Win Percent</td><td>16.7%</td><td><b>83.3%</b></td></tr>
<tr><td>Total Games</td><td>24</td><td>Draws</td><td>0</td><td></td><td></td></tr>
</table>

**Notes:** The FixedMinimax decider was given a maximum depth of 8 to search to.

Most of the games ran in this experiment resulted in a victory for the AI player; the Human players found it difficult to overcome the AI's ability to rapidly flip counters and reduce the number of moves the player could make. I expect the high number of Human losses is because the FixedMinimax decider was given a large maximum depth, resulting in the AI being difficult to play against.

### 5.2.2 – Experiment 8: Human Player vs Monte Carlo Tree Search Player

I also decided to have the players attempt to defeat the MCTS algorithm as well, since the approach to selecting moves that MCTS decider uses is quite different from the one the Minimax players use. As a result, players could have found it easier or harder to play against an AI using a MCTS decider.

**Results:**

<table>
<tr><th colspan="4">Experiment 8A: Human vs Monte Carlo (Hard)</th><th>Player 1</th><th>Player 2</th></tr>
<tr><td colspan="4">Decider</td><td>Human</td><td>MCTS (Minimax)</td></tr>
<tr><td colspan="4">Evaluator</td><td>-</td><td>Positional</td></tr>
<tr><td colspan="4">Victories</td><td>3</td><td><b>21</b></td></tr>
<tr><td colspan="4">Win Percent</td><td>12.5%</td><td><b>87.5%</b></td></tr>
<tr><td>Total Games</td><td>24</td><td>Draws</td><td>0</td><td></td><td></td></tr>
</table>

---

[21] These 21 Principles can be found in Appendix B.

**Notes:** The AI player's MCTS decider was provided with the following arguments; run heavy simulations to a depth of three, minimax time of 5, with a random move probability of 0.01, and a maximum simulation count of 10,000.

The results of this experiment are similar to the Humans versus Minimax results; the AI player was able to defeat the human players in a large majority of the games that were run. One volunteer commented that even though they lost all of their games, they felt that this AI was easier to play against than the AI player used in the previous test. I assume that this is because the MCTS algorithm focuses on the most likely way to win, which is comparable to how a human views the board. In comparison, the Minimax deciders view the future game states more mathematically and factually, which can feel robotic to human players.

The comments given by the volunteers gave me the idea of testing a light simulation MCTS player against humans, as it changed enough of the MCTS player's decision process to warrant its own examination.

**Results:**

<table>
<tr><th colspan="4"><b>Experiment 8B: Human vs Monte Carlo (Easy)</b></th><th>Player 1</th><th>Player 2</th></tr>
<tr><td colspan="4">Decider</td><td>Human</td><td>MCTS (Random)</td></tr>
<tr><td colspan="4">Evaluator</td><td>-</td><td>Positional</td></tr>
<tr><td colspan="4">Victories</td><td>7</td><td><b>16</b></td></tr>
<tr><td colspan="4">Win Percent</td><td>29.2%</td><td><b>66.7%</b></td></tr>
<tr><td>Total Games</td><td>24</td><td>Draws</td><td>1</td><td></td><td></td></tr>
</table>

**Notes:** The MCTS decider was instructed to use light simulations to create its search tree, and was told to run a maximum of 1000 simulations.

By running another test using a MCTS decider with a reduced number of light simulations, the difficulty of the AI was notably changed; each volunteer defeated the AI at least once during their play time. This shows that altering the difficulty of the AI through the already provided arguments is viable. However, additional arguments to change the difficulty could be useful for the MCTS algorithm, as it does naturally perform well, even when limited to a small number of simulations.

### 5.2.3 –Summary

At the outset of the project, I was unsure if the AI programs created for this project would be able to defeat a competent human player, due to how complex the strategy of Othello can become. However, I am glad to see that the AI players have surpassed my expectations, as I was only able to defeat each one once in the experiments shown above, despite my knowledge of the game and of AI's implementation. The AI players were able to adapt to any situation the human player put them in, and were often in full control of the game.

One such example of the control the MCTS decider demonstrated was during a game where I had been in the lead with around 8 moves left to make, when the MCTS player determined that, because of the layout of the board and the fact that I had no legal moves to make if it played in a specific order, that there was a way for it to play in all 8 of the remaining spaces on the board. As a result, I watched it take eight consecutive turns, after which my score had decreased from 50 points to 6 points. I was amazed by how capable it was at manipulating me into losing the game so easily.

## 5.3 – Artificial Neural Network Performance Analysis

The final evaluation tasks I carried out were all focused on the deep learning side of the project. By the time I had reached this stage, I had created a set of promising neural networks, so that I could compare their performance against one another. My goal wasn't necessarily to determine which ANN could win the most games, but rather to determine how the data and network structure used to create the ANNs affected their ability to evaluate Othello states.

Each DeepLearningEvaluator used one of the ANNs to calculate the value of the game states it was given; the ANNs ability to help an AI player win by returning accurate valuations of the game state was measured in the following experiments. The configurations used by each of the ANNs that feature in the experiments are shown below.

| ANN Label | Data Format Used | Network Layout Used |
|---|---|---|
| DF2-1 | Data Format 2 | 1 |
| DF2-2 | Data Format 2 | 2 |
| DF3-1 | Data Format 3 | 1 |
| DF3-2 | Data Format 3 | 2 |
| DF3-3 | Data Format 3 | 3 |

*Figure 21: The data formats and neural network layouts used by each ANN.*

| Data Format | Description |
|---|---|
| Data Format 2 | Label represents win or loss for the evaluated player. |
| Data Format 3 | Label is the percentage of times the evaluated player won from this state. |

*Figure 22: The descriptions for each data format.*

| Layout Number | 1 | 2 | 3 |
|---|---|---|---|
| Type | Classification | Classification | Classification |
| Learning Rate | 0.01 | 0.01 | 0.01 |
| Optimisation Algorithm | Stochastic Gradient Descent | Stochastic Gradient Descent | Stochastic Gradient Descent |
| Updater | Nesterov's | Nesterov's | Nesterov's |
| Updater Momentum | 0.9 | 0.9 | 0.9 |
| Number of Inputs | 128 | 128 | 128 |
| Hidden Layers | 1 | 3 | 3 |
| Nodes per Hidden Layer | 100 | 128 | 256 |
| Hidden Node Activation Function | ReLU | ReLU | ReLU |
| Number of Outputs | * | * | * |
| Loss Function | Negative Log Likelihood | Negative Log Likelihood | Mean Square Error |

* = Depends on format of training and testing data

*Figure 23: The parameters used by each neural network layout.*

| ANN Label | Accuracy | Precision | Recall | F1 Score |
|---|---|---|---|---|
| DF2-1 | 0.7100 | 0.7090 | 0.7035 | 0.7062 |
| DF2-2 | 0.7206 | 0.7199 | 0.7142 | 0.7171 |
| DF3-1 | 0.7090 | 0.6105 | 0.4797 | 0.5372 |
| DF3-2 | 0.7067 | 0.5831 | 0.4869 | 0.5307 |
| DF3-3 | 0.7001 | 0.6131 | 0.4741 | 0.5347 |

*Figure 24: The statistics produced by the learning process for each ANN.*

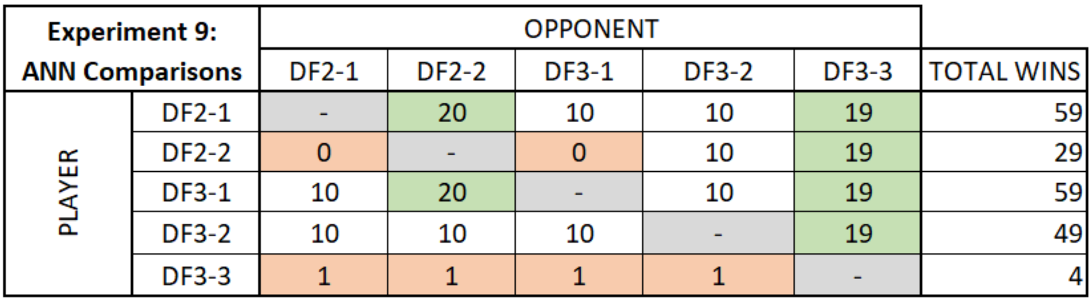
As you can see above, the maximum number of layers in the range of ANNs is 3; I decided to keep the number of layers low to improve response times when evaluating a game state, as introducing many layers would slow down the program by a large degree.

It is also worth noting that every ANN was created using 100% of the training data; the methods used to create Data Format 3 rely on creating numerous bins to categorise each game state, with the number of bins being decided by the average amount of times any game state appeared throughout the tournament data. As this average was surprisingly low for the data, and the number of data points and bins were smaller than expected, I decided to use as much of the data as possible, to ensure that the ANN had enough data to study and train. With regards to Data Format 2, due to the way its records were produced, there were less of them to be processed, so I chose to use all the available training data to ensure it had enough data points to examine as well.

### 5.3.1 – Experiment 9: Comparing the Potential of various ANNs

For the sake of comparison between the created ANNs, I ran numerous games between AI players using DeepLearningEvaluators fitted with one of the five networks, and examined how frequently the players would defeat each other.

**Results:**



| Experiment 9: ANN Comparisons | | OPPONENT | | | | | |
|---|---|---|---|---|---|---|---|
| | | DF2-1 | DF2-2 | DF3-1 | DF3-2 | DF3-3 | TOTAL WINS |
| PLAYER | DF2-1 | - | 20 | 10 | 10 | 19 | 59 |
| | DF2-2 | 0 | - | 0 | 10 | 19 | 29 |
| | DF3-1 | 10 | 20 | - | 10 | 19 | 59 |
| | DF3-2 | 10 | 10 | 10 | - | 19 | 49 |
| | DF3-3 | 1 | 1 | 1 | 1 | - | 4 |

*Figure 25: Table of results from comparing various ANNs' performance against each other. To find out how many times X won against Y, find the cell where row X and column Y intersect.*

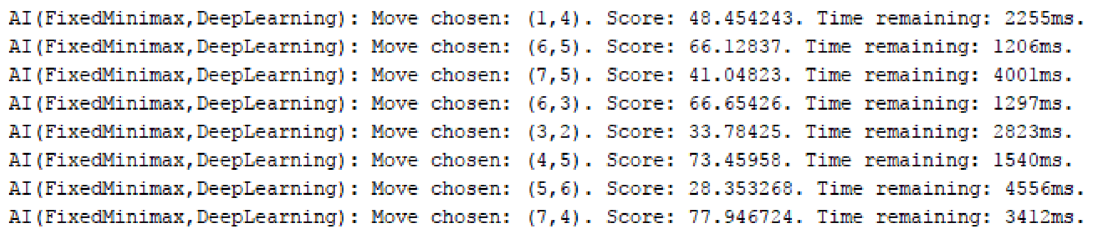
**Total Games Run:** 200 (20 per unique ANN combination)

**Notes:** All DeepLearningEvaluators were paired with a FixedMinimaxDecider set to search to a maximum depth of 5.

These initial results seem to indicate that ANNs DF2-1, DF3-1 and DF3-2 were the strongest ANNs out of the group. One reason I wanted to explore how these matchups played out was related to how the DeepLearningEvaluator interprets the returned data; the ANN within the evaluator returns an array of the likelihood that the provided data belongs to each known class. The evaluator then regresses this data into one scalar value between 0 and 100 to use as the GameState's score.[22] Due to the different data formats, I expected that this regression process could have caused an imbalance of victories for one type of format.

---

[22] See Section 4.5.3 for more information on how the evaluator does this.



```
AI(FixedMinimax,DeepLearning): Move chosen: (1,4). Score: 48.454243. Time remaining: 2255ms.
AI(FixedMinimax,DeepLearning): Move chosen: (6,5). Score: 66.12837. Time remaining: 1206ms.
AI(FixedMinimax,DeepLearning): Move chosen: (7,5). Score: 41.04823. Time remaining: 4001ms.
AI(FixedMinimax,DeepLearning): Move chosen: (6,3). Score: 66.65426. Time remaining: 1297ms.
AI(FixedMinimax,DeepLearning): Move chosen: (3,2). Score: 33.78425. Time remaining: 2823ms.
AI(FixedMinimax,DeepLearning): Move chosen: (4,5). Score: 73.45958. Time remaining: 1540ms.
AI(FixedMinimax,DeepLearning): Move chosen: (5,6). Score: 28.353268. Time remaining: 4556ms.
AI(FixedMinimax,DeepLearning): Move chosen: (7,4). Score: 77.946724. Time remaining: 3412ms.
```

*Figure 26: Output from one of the tests, showing one player's score increasing and the other's decreasing as the game swings in the favour of one player.*

When I examined the AI readouts that showed the evaluation scores the ANNs and DeepLearningEvaluators were returning, they resembled the same change over time that other evaluator scores do; one neural network could tell it was slowly gaining control over the game, while another could tell it was moving into less advantageous game states. Additionally, even though two players may have ANNs that were made with different data formats, the scores produced by each DeepLearningEvaluator all exhibited the uniform increase and decrease as seen above, regardless of how the initial data was formatted.

This not only confirmed that the DeepLearningEvaluator was correctly interpreting the ANNs values, and that all of the ANNs did have a good understanding of the qualities that indicate a beneficial game state for the player, but also that ANNs could be used to create intelligent evaluation functions. However, the real test of their evaluation ability would be against an existing evaluation function.

### 5.3.2 – Experiment 10: Positional Evaluator vs Deep Learning Evaluator

I wanted to observe how a DeepLearningEvaluator could compare to the PositionalEvaluator – which was the most advanced hard-coded evaluation function in my Othello program – in both a Minimax match and a MCTS match. I chose to use all five ANNs to see how each of their performances varied against opponents that do not utilise deep learning information.

**Results:**

| **Experiment 10A: Positional vs DL (Minimax)** | | | |
|---|---|---|---|
| Positional Player | | Deep Learning Player | |
| Positional | 10 | 10 | DF2-1 |
| Positional | 20 | 0 | DF2-2 |
| Positional | 10 | 10 | DF3-1 |
| Positional | 10 | 10 | DF3-2 |
| Positional | 20 | 0 | DF3-3 |

| Difference between the two players' scores (DL - Pos) |
|---|
| -4 |
| -64 |
| -3 |
| -14.9 |
| -64 |

**Total Games Run:** 100 (20 per ANN)

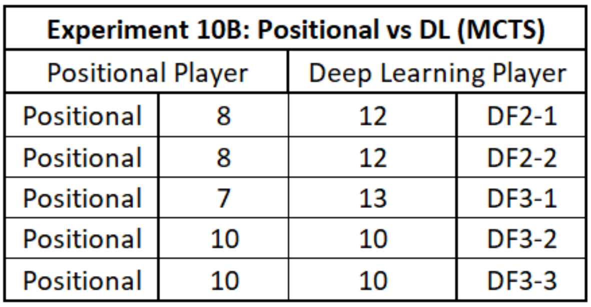
**Notes:** Both Fixed Minimax deciders were set to search to a maximum depth of 5.

The left-hand table displays the number of times the player equipped with the specified evaluator was able to win. The same three promising ANNs from the previous experiment were able to compete competently against the Positional evaluator, while the lacklustre performance of ANNs DF2-2 and DF3-3 continued. As the main Othello program of my project is able to measure the average score each player achieved over the series of games run, the right-hand table is used to show the difference between the scores of the Deep Learning player and the Positional player. Though some ANNs were able to draw in overall games won, all networks led their player to have a lower average score than the player equipped with a Positional evaluator.

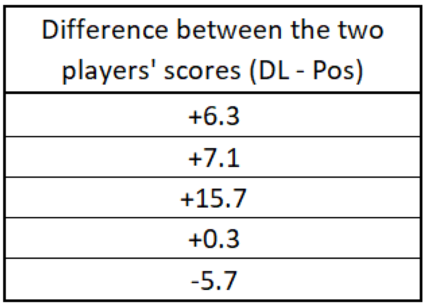
Notably, in the sets of games that ended in a draw, it was the player who was controlling the dark counters that won; this means that for these games, whoever placed the first counter was the winner.

I believed that reason that each player's score was a factor of 10 was related to a lack of randomisation and variety in the Minimax algorithm, hence I decided to also run the same tests using the MCTS decider.

**Results:**



| Experiment 10B: Positional vs DL (MCTS) | | | |
|---|---|---|---|
| Positional Player | | Deep Learning Player | |
| Positional | 8 | 12 | DF2-1 |
| Positional | 8 | 12 | DF2-2 |
| Positional | 7 | 13 | DF3-1 |
| Positional | 10 | 10 | DF3-2 |
| Positional | 10 | 10 | DF3-3 |



| Difference between the two players' scores (DL - Pos) |
|---|
| +6.3 |
| +7.1 |
| +15.7 |
| +0.3 |
| -5.7 |

**Total Games Run:** 100 (20 per ANN)

**Notes:** Both MCTS deciders were given a maximum simulation count of 500, instructed to carry out heavy simulations to a minimax depth of 2, given a random move probability of 0.01, and given 5 seconds to run each simulation minimax.

These results show notable improvements in the performance of the DeepLearningEvaluators; though the number of victories that each evaluator achieved varied, the difference between the average scores was very consistent. As a result, I believe that the score difference is the best way to determine the difference in skill level of the DeepLearningEvaluators and the PositionalEvaluator:

- **DF2-1** – The first of the ANNs to be able to win more games than its opponent, and had a positive difference between its average score and the opponents.
- **DF2-2** – A similar case to DF2-1, but somewhat more promising due to the greater difference between the player's scores.
- **DF3-1** – The most successful of the ANNs, in both number of games won and ability to maximise the score of the player in each game, resulting in the higher score difference between it and the opposing player.
- **DF3-2** – A less successful result, but this ANN was still able to gain a higher score than their opponent on average.
- **DF3-3** – Though this network resulted in a negative score difference, it was still capable of winning an equal number of games with the Positional evaluator.

Overall, the ANN-backed DeepLearningEvaluator was able to beat out the Positional evaluator in numerous games, with some wins being decided by a sizeable margin. This data demonstrates the effectiveness of using an ANN for powering an AI's evaluation function, even against an evaluation function that factors in numerous aspects of the game state to produce its results.

### 5.3.3 – Experiment 11: Altering the Epoch Count

An important factor in generating an ANN is choosing the number of epochs to run in the training process. An epoch refers to one pass-through the training set to allow the ANN to learn the features of the data. The previously examined ANNs used 20 epochs during their construction, which has produced excellent results; however, I was interested in examining the effect of lowering this number to 10 epochs, as it was possible some of the 20 epoch ANNs would have been "over-trained" as a result of being created with a higher than necessary epoch count.

In the following results tables, ANNs that were constructed using with 10 epochs have "-10" appended to their label (e.g. DF2-1 has become DF2-1-10). Likewise, the ANNs used in Experiments 9 and 10 will now be referred to with a "-20" at the end of their label.

**Results:**

| **Experiment 11A: ANN Comparisons (10 epochs)** | | OPPONENT | | | | | |
|---|---|---|---|---|---|---|---|
| | | DF2-1-10 | DF2-2-10 | DF3-1-10 | DF3-2-10 | DF3-3-10 | TOTAL WINS |
| PLAYER | DF2-1-10 | - | 20 | 10 | 0 | 20 | 50 |
| | DF2-2-10 | 0 | - | 0 | 0 | 5 | 5 |
| | DF3-1-10 | 10 | 20 | - | 10 | 19 | 59 |
| | DF3-2-10 | 20 | 20 | 10 | - | 20 | 70 |
| | DF3-3-10 | 0 | 15 | 1 | 0 | - | 16 |

**Total Games Run:** 200 (20 per unique ANN combination)

**Notes:** All DeepLearningEvaluators were paired with Fixed Minimax decider set to search to a maximum depth of 5.

I collected data about which ANNs could win against each other in the same way as Experiment 9, but the results were quite different; the version of DF3-2 made using 10 epochs was the ANN with the most victories in these games, while many matchups that were draws when the 20 epoch networks were tested became definitive victories or losses. To further evaluate the ANNs, I used the same method as Experiment 10 to gain information on how these ANNs can compete against the evaluation performance of the PositionalEvaluator.

**Results:**

| **Experiment 11B: Positional vs DL (Minimax, 10 epochs)** | | | |
|---|---|---|---|
| Positional Player | | Deep Learning Player | |
| Positional | 0 | 20 | DF2-1-10 |
| Positional | 0 | 20 | DF2-2-10 |
| Positional | 10 | 10 | DF3-1-10 |
| Positional | 3 | 17 | DF3-2-10 |
| Positional | 10 | 10 | DF3-3-10 |

| Difference between the two players' scores (DL - Pos) |
|---|
| +42 |
| +36 |
| -10 |
| +15.6 |
| -21 |

| Experiment 11C: Positional vs DL (MCTS, 10 epochs) | | | |
|---|---|---|---|
| Positional Player | | Deep Learning Player | |
| Positional | 6 | 14 | DF2-1-10 |
| Positional | 7 | 13 | DF2-2-10 |
| Positional | 8 | 12 | DF3-1-10 |
| Positional | 5 | 15 | DF3-2-10 |
| Positional | 10 | 10 | DF3-3-10 |

| Difference between the two players' scores (DL - Pos) |
|---|
| +13.5 |
| +5.6 |
| +8 |
| +11.8 |
| +2.7 |

**Total Games Run:** 200 (20 per ANN per table)

**Notes:** In the tests run in 11B, both Fixed Minimax deciders were set to search to a maximum depth of 5. In the 11C tests, both MCTS deciders were given a maximum simulation count of 500, instructed to carry out heavy simulations to a minimax depth of 2, given a random move probability of 0.01, and given 5 seconds to run each simulation minimax.

Surprisingly, all of the 10 epoch ANNs were able to soundly defeat or tie with the PositionalEvaluator in both the Minimax and MCTS tests. Overall, when comparing these results to the ones from Experiment 10, it appears that networks DF2-1 and DF3-2 have benefited the most from the epoch change, while the effectiveness of networks DF2-2 and DF3-1 have decreased. The correct values to use for constructing an ANN are often hard to determine, due to the amount of domain-specific knowledge needed about the data, followed by the variety of possible network configurations.

The results of the above experiment have shown that fewer epochs is not suitable for all types of networks. To examine how much the networks' performances have changed because of the epoch change, I ran one last experiment to determine how well the 10 epoch ANNs performed against their 20 epoch counterparts.

**Results:**

| Experiment 11D: ANN-10 vs ANN-20 Evaluation | | | |
|---|---|---|---|
| ANN-10 Player | | ANN-20 Player | |
| DF2-1-10 | 23 | 17 | DF2-1-20 |
| DF2-2-10 | 20 | 20 | DF2-2-20 |
| DF3-1-10 | 18 | 22 | DF3-1-20 |
| DF3-2-10 | 18 | 22 | DF3-2-20 |

| Difference between the two players' scores (ANN-10 - ANN-20) |
|---|
| +5.2 |
| -0.2 |
| -3.7 |
| -3.1 |

**Total Games Run:** 160 (40 per matchup)

**Notes:** The MCTS decider was used alongside the DeepLearningEvaluator with the given ANN to run these tests. The MCTS used heavy simulations to a depth of 2, with a max of 500 simulations, 5 seconds to minimax, and a random move chance of 0.01. The DF3-3 ANNs were omitted due to their prior poor performance.

The results from this round are difficult to decipher initially, due to the small margins of victory between the players using each ANN. However, once you consider the size of each network, a clearer image begins to emerge:

- **DF2-1** – Uses the smallest data set and the network with the least number of nodes in it. As a result, using a smaller number of epochs is beneficial to its learning process.

- **DF2-2** – Still uses the smaller of the two data formats, but uses a much larger network, so more epochs are needed for it to fully understand the training data.
- **DF3-1** – Learns from the larger Data Format 3 while using the smaller of the network configurations. The 10-epoch version of the network clearly hasn't had enough time to learn the data, as its performance is worse than its 20-epoch variant.
- **DF3-2** – Though the DF3-2-10 ANN initially showed promise during the Minimax and MCTS comparisons, it appears that it couldn't compare to its 20-epoch counterpart, which is unsurprising since the DF3-2 networks use the largest data set and a network configuration with a large number of nodes and links.

Overall, it is clear that all of the combinations of Data Formats and network configurations have potential to become intelligent Othello evaluation functions, but care is needed to ensure the network is trained to an appropriate degree. It seems that DF2-1 benefits from 10 epochs the most, while DF2-2, DF3-1 and DF3-2 are more suited to an epoch count in the range of 15 to 20.

As for DF3-3, it clearly needs to be given a lot more time to learn from the training data during the ANN creation process, as its network configuration means it has more than double the number of nodes of the second biggest layout. In hindsight, I think a better approach would have been to increase the number of layers in the network, instead of increasing the number of nodes per layer as I did to create the configuration for DF3-3.

### 5.3.4 – Summary

I was glad to see that all the work I had completed for the Deep Learning section of the project has resulted in competent ANNs whose evaluations can compete with and even defeat the best hard-coded evaluator I created.

Though there is room for improving the network configurations by utilising full regression networks and tailoring the parameters of each ANN to better suit its application, I am pleased with the current success of the DeepLearningEvaluator and the ANNs that power it.

As for the Data Formats that I created, I believe that both are suitable for creating an ANN with, as they both capture the necessary information that should be learned by a network.

## 5.4 – Evaluating the Accuracy and Suitability of the Implementation

During all of the games run for the experiments throughout Section 5, there were no exceptions thrown by the program. As a result of the existing validation in the program, I can derive the following results:

1. The Othello game logic is once again confirmed to be correctly implemented. An Illegal Argument Exception would have arisen from the playMove method of GameState if there had been a violation of logic[23]. This also confirms that the GameState provides the AI with the correct logic as well.
2. The AI players are once again confirmed to be functioning as intended, as if they had provided illegal moves or null references, the playMove method would have also detected this. As a result, this confirms that the deciders and evaluators are correctly carrying out their tasks and are returning legal moves to play.

These facts, combined with the GameState validation carried out in Section 4.2.3, and the various AI behaviour checks carried out in Section 4.4, ultimately prove that all of the implemented game logic and AI algorithms are in working order.

In the end, I think my approach to implementing the Othello game logic and the AI systems was a good choice, as the customisability of the AI allows for a varied number of games to be played, while the Othello game is fully playable by either AI or Human players, which are two goals I wanted to achieve.

## 5.5 – Discussing the Flaws in the Implementation of Deep Learning

Personally, I am pleased with the fact that I was able to implement the DeepLearningEvaluator as expected; its performance as an evaluator is capable of exceeding that of any other evaluator class currently in the Othello program.

I feel that my approach to the Deep Learning process was also successful in producing a suitable number of ANNs to examine and analyse. Plus, despite my troubles with learning the DL4J framework, I feel that its usage within the project benefitted the final result, due to the amount of methods and classes it provided to help create the neural networks.

However, I feel that it is important to discuss the aspects of the Deep Learning process that did not turn out as successfully. Though I will elaborate on how these problems could be fixed in the next section, this section will highlight the effect that each one had on the outcome of the project.

1. **The Lack of Regression-based Networks**
   At the start of the project, I planned to investigate regression networks alongside the classification networks that were eventually created. However, a combination of technical difficulties with DL4J and the tournament data's ability to be classified easily resulted in a focus on classification networks.
   Overall, this has very little of an impact on the final system, due to the DeepLearningEvaluator's built-in regression, but the ability to compare an actual regression network to the evaluator's pseudo-regression would have been interesting.

---

[23] See Section 4.2.1 for information on how the playMove function upholds the legality of playable moves.

2. **The Lack of Variance in Evaluated Networks**
   For my evaluation on the ANNs, I used a variety of networks, but the configurations of these networks were not particularly varied, as most of them shared similar parameters. For many of these parameters, the similarity made sense, such as for the networks' learning rates or input node count. However, other parameters could have been varied more to create a better set of ANNs to work with, as definitive conclusions were difficult to identify in my ANN experiments, due to the similarity between the networks.
   The main cause for this was my inexperience with determining what network layouts and parameters would benefit the final ANNs. It is an aspect that I could definitely improve if I had more time to spend on it.

3. **Limiting the Deep Learning applications to an Evaluator**
   During my research, I overlooked the possibility of integrating Deep Learning routines into a decider class to augment their operation. There was the opportunity for altering the UCT algorithm – which determines what nodes the MCTS expands when constructing its game tree – to utilise deep learning methods as a way of improving its selection process. Rather than using a data set of archived games, the Deep Learning UCT algorithm could have used previous game simulations to determine what types of moves are the best to explore, which ensures that the most likely moves to benefit the player are adequately explored within the AI's decision time limit.
   Though I did miss the opportunity to implement this, I feel that I also would not have had the time to implement it alongside the other ANN work I have done. Additionally, due to the large amount of simulations and states the current version of the MCTS algorithm can examine within the time limit enforced by my Othello program, I believe that there is not an immediate need to optimise its selection process further.

Despite the fact that these areas of Deep Learning were not fully explored by my project, I believe that the way I have applied ANNs in my project has still been very beneficial in improving the AI player's ability to comprehend and play Othello, which was the original goal of this project.

# Section 6 – Future Improvements for the System

Overall, I am very pleased with the quality of the Othello program, the AI algorithms implemented within, and the results of the evaluation stage. But as previously mentioned, I have noted some areas where the work I carried out could be reimplemented to a higher standard, or where functionality could be could be altered or improved as a result of conclusions derived from the Evaluation phase. I will discuss these points in this section.

## 6.1 – Improvements to Existing Features

One of the most glaring issues I see with the project at this time is the way caching has been implemented. Although analysing and storing where a player can carry out legal moves at object instantiation results in faster computation time for deciders that require said information, the additional time added to the construction of the GameState is problematic. If a state is created and the information it has cached is never used, then the time spent computing the cache is wasted.

Logically, the best way to improve this would to initialise the object with empty caches, then run the code to calculate these values when they are first requested. Once this first calculation has been completed, the result can be cached and used for future calls for the data. This ensures only needed information is cached instead of all possible information. I believe the additional construction time for GameState objects is currently holding back many of the program's algorithms, especially the MCTS algorithm, as thousands of GameStates are generated by deciders when they are determining what move to play. The current algorithms still function correctly, but this change to the GameState would speed up the run times of many of the program's classes and objects.

Additionally, my decision to change the legal move list to a legal move array was also a poor one, as many algorithms waste time iterating over the array looking for the locations where moves can be played, whereas a list could be iterated over without this worry. This change would be easy to carry out, as all of the data about the legal moves is already contained in the GameState class; only the interface and the algorithms that access it would need to be altered.

Moving away from optimisations, I would also like to see changes to the GUI of the program, as the one currently in place could be improved. Most of the volunteers for the Human vs AI analysis commented that they would prefer if the counter placing and flipping process was more animated, as the change happens quite rapidly in the current version of the program. Though I initially saw it as unnecessary since the AI would not rely on the GUI in anyway, I agree that improving the clarity of the moves made via animations would aid a human player's understanding of the game. Furthermore, even though the ability to enter command line arguments allows the program to be customised quite easily, I think many users would prefer a graphical options screen to edit these parameters. This screen could appear after launching the program, but prior to initiating the main Othello game.

As for improvements to be made to the Deep Learning facilities of the project, I think that a proper exploration of regression networks' potential use as part of the DeepLearningEvaluator class should be carried out, as discussed in Section 5.5. I was unable to fully create a regression network with DL4J's tools, but the final DeepLearningEvaluator class does act as a final regression layer for the classification networks that have been made. Nonetheless, I believe that creating an actual regression network will provide more accurate values due to the expanded number of tools available within the DL4J framework for regression operations.

Finally, there is always the possibility for iterating on the structure used to create ANNs to improve it; my lack of knowledge in the field has likely limited the potential of the current networks' ability due to their somewhat mismatched internal structures. With more time for the project, I would have liked to create many more network layouts to determine the best configuration of nodes and links for creating an ANN for evaluating Othello. From the Evaluation section, I have already identified increasing the number of layers of the ANNs as a potential area for improvement, but I am sure there are many others as well.

## 6.2 – New Features

If I was to continue the project from this stage, the next area of development I would explore would be to convert the project from an Othello-focused program to a general game playing program. I believe that the current GameState object could be refactored into an OthelloGameState class, and a new interface class could then be created to allow for GameState objects to be created for other games. This would lead to an interesting analysis on how easily the currently implemented AI algorithms could be re-applied to a general game playing environment, and how the performance of a general game-playing AI would compare to one built specifically for a game. Though this would be a very large undertaking, I feel it would be a necessary step if the project was to continue, as I have almost completely explored Othello game playing at this point.

I have already discussed improving the existing caching in the GameState class, but there is also the possibility of adding caching to the evaluators, as many states are likely to reappear in the evaluation process. I experimented with this concept during development, but I soon discovered that implementing an efficient caching system for all evaluators would require too much time and testing to ensure it was correctly working.

Concerning the performance of the DeepLearningEvaluator and the ANNs, I believe that the accuracy of the network could be improved with an additional amount of data; however, rather than source this from an archive of Othello games, I think that adding the ability for the ANNs to learn from the game states passed to it would be a much better option. DL4J's representation of ANNs allow for additional learning after the initial training stage, and adding this feature could allow the prediction ability of the DeepLearningEvaluator and the ANN it is using to improve just by playing against itself repeatedly. In my evaluation of the current set of ANNs, I believe that the performance of said networks was held back by gaps in the database of Othello games; adding self-learning to the DeepLearningEvaluator could solve this issue quite rapidly.

Additionally, I think exploring the potential application of integrating Deep Learning techniques into a MCTS decider to improve its search tree construction and evaluation abilities would be an excellent direction to take the project in next, as with the right implementation, it would result in a decider that needs less time to even better results than existing decider classes.

While on the topic of the ANNs, I would also recommend that tests be carried out comparing Human players to a DeepLearningEvaluator to evaluate how well humans can play against the ANNs' evaluation methods. I was unable to carry out this test, as the volunteers I used in other Human tests were unavailable during the final part of the project.

One last feature that could be added would be an open book for AI players[24]. I discussed the possibility of implementing a feature like this at the outset of the project, but as time passed I determined that the ANNs were ultimately a superior version of an open book, due to their generalised understanding of how to advance from any state in the game.

---

[24] See Section 2.2.4 for more detailed open book information.

# Section 7 – Conclusion

This project's aim was to explore how an AI player could be created or trained to play the board game Othello, via methods such as Minimax searching, Monte Carlo Tree Searching, and Deep Learning. To facilitate this AI analysis, the project would also include a programmatic recreation of the Othello game, along with a graphical interface to display it.

The project and all the work associated with it has now been completed; I was able to implement every key feature that had been stated at the project's outset, along with adding various other features to the final system.

The Othello game program created for the project utilises a fully-featured game state representation class that is capable of validating moves and states to ensure full satisfiability of the original board game's rules, while also providing numerous methods to allow AI agents to analyse the state to any degree necessary.

The Othello program itself has been equipped with a command line argument system to allow for multiple degrees of customisability, including AI difficulty adjustment, board size alteration, and statistical measurement output.

The GUI created to allow for viewing the Othello game state provides users with a way to play against another Human or AI opponent, and displays helpful notifications and relevant data to aid new players in learning how to play the game.

The AI system I have implemented allows for easy assembly of an AI player from a variety of different move decision algorithms and evaluation functions. Despite some doubts during development about the capability of the AI system versus a human player, my evaluation has shown that the performance of the AI's decider and evaluator classes is as expected; the Monte Carlo Tree Search is a particularly notable decider for its ability to analyse each move's probability of leading to a victory based on various simulations of the future of the game.

Finally, the Deep Learning research and experimentation I carried out has led to the ability to create ANNs via the project's deep learning classes. Once created, the ANNs can be loaded into their own evaluation function class to use with an AI player; these DeepLearningEvaluators have the ability to surpass the performance of any other evaluator in the program when given a correctly trained network to use.

I am very satisfied with the outcome of this project; though I am somewhat disappointed that I couldn't spend more time improving the quality of the ANNs, the ones that have been produced and analysed are still very capable at evaluating the game states they are given. I believe that the program I have created can be used as a tool to allow Othello novices to hone their skills against varying difficulties of AI opponents, while also providing useful facilities to enable AI performance analysis and neural network development.

Overall, the main goal of producing an AI capable of playing Othello was met; the Monte Carlo Tree Search decider and Deep Learning evaluator combine to produce a player capable of playing the board game Othello to a high degree of skill, and is capable of defeating both Human and other AI players.

# Section 8 – Personal Reflection

Overall, I think this project has progressed relatively smoothly; during the first 2 months, I stuck to the schedule I outlined in my Initial Plan, and I was even able to complete some implementation tasks ahead of time. This in turn allowed me to devote more time to the AI and Deep Learning implementation tasks, as I felt they would require a lot of effort to complete.

However, the final few weeks involved me frantically catching up on the work I had yet to complete, as I had required even more time than I had thought to implement both the MCTS algorithm and the Deep Learning system. Though this did not affect the quality of the work I carried out, it caused more stress for myself than necessary.

Considering how large of a task the Deep Learning system was, I should have had more milestones in place for it, such as "Complete Data Formatting class", "Complete Network Constructing class", etc. I feel that I would have had a better target to aim for if these kinds of milestones had been in place, rather than the single "AI uses Deep Learning" milestone at the end of the allotted time for the task. In the future, I will make sure to use milestones more wisely to keep my work schedule more balanced.

As I chose to create this project, I have always been very passionate about working on and improving it as much as I could. Whether it was designing the interface or implementing decider code, I was committed to producing the best possible deliverables for the project. However, I feel that this has resulted in the needless addition of some features.

For example, it was not necessary for me to add the Minimax decider before implementing the MCTS decider, since the latter was always going to be superior to the former. However, I do feel that using the extra time to implement the Minimax decider provided me with valuable experience that then aided the development of the MCTS decider. Additionally, providing this easier decider class allows for more customisation of the AI difficulty for users of the system.

Despite this, there are other examples of my passion for the project resulting in "feature creep", such as some unnecessary command line options, the multiple Minimax players, some of the statistics data in the main Othello class, and the game archiving system. Though I believe these features aid the final product's quality, it would have been best to take a step back, observe the project as a whole, and decide if the feature was truly necessary, rather than implementing it as soon as I thought of it. I will keep this approach in mind in future projects I work on.

Though the results of the Deep Learning work I carried out were successful, I think that my approach to it was flawed, and resulted in the evaluation process becoming tedious. After creating the code necessary to format the data and construct the networks, I began creating ANNs very quickly, by changing various values between iterations. At the time, I thought this was beneficial for the project, as I was able to see how the statistics for each ANN changed, but since I had not decided on any way to evaluate the networks to determine their worth to the project, much of the work carried out in this time was wasted.

It was only as I began to run out of time on my schedule that I began using the techniques shown in Section 5.3 for examining and comparing the ANNs. I would have benefited from this system being in place earlier, as I could have gathered much more data to support the evaluation of the networks. I hadn't done any work prior to this project that involved iterating on content repeatedly, so I didn't expect to need to evaluate each ANN so thoroughly to better inform the decisions for the next produced network, despite the system I laid out in Section 3.6.1. I will make sure to clearly define a work plan for this type of exercise should I come across it again in the future.

Despite these missteps towards the end of the project, much of the work I carried out during the first tasks of the project was done in a way that made the later work easier to complete. For example, when I was constructing the GameState class, I understood that despite how well I checked the functionality of the code, there would be times where inexplicable errors would crop up with the AI system. From a previous tast I had completed during my AI university module, I knew that debugging these kinds of errors would be a huge hassle, due to the web of classes that could be responsible for the error by that point in development.

To combat this, I added validation methods in the GameState class to ensure that any obvious errors would be caught and output with a clear error message specific to the project. For example, the *playMove* method will throw an exception if a null object or illegal move is passed to it; when this specific exception would arise, I knew that the culprit had to be the decider that last ran, as the *decide* method directly returns an object to the *playMove* method. With this type of validation in place, I was able to narrow down the search area to one method instantly, which saved me a lot of time over the course of the project. This kind of forward thinking and consideration for future development of the program is a skill I have honed over the course of this project, and I hope to put it to good use in my future endeavours.

# Glossary

## Othello Terms

- **Counter** – The piece that players use to play Othello. Can be two colours: light (white) or dark (black).
- **Board** – The 8 by 8 playing area that counters can be placed in, thus allowing for sixty moves to be played in a game at maximum, due to the game beginning with four counters already placed.
- **Flipping** – Process of turning a light counter to a dark counter, or vice versa.
- **Phases** – Othello is split into three phases of 20 turns each, known as the Opening, Midgame and Endgame phases. [3]
- **Bracketing** – Another name for placing two counters in a horizontal, vertical or diagonal line with each other, thus "bracketing" the counters of the opposite colour between these two counters. This results in the bracketed counters being flipped. [3]
- **Stable** – A counter is stable if it cannot be bracketed in any way in the current game state, and therefore cannot be flipped. [3]
- **Game State** – A single configuration of the game board and the counters on it.
- **Manoeuvrability** – A Measure of how many moves a player can play onto the current game state. The more moves a player can choose from, the more manoeuvrable they are said to be.

## Game Theory Terms

- **Perfect Information** – A Game Theory descriptor. A game with Perfect Information is one where any player can view all observable information at any given time. Opposite of Imperfect Information game.
- **Zero-sum Game** – A Game Theory descriptor. A Zero-Sum Game is any game where the gain observed from a move for one player is equal to the loss the other player receives for the same move.
- **Solved Game** – A Game Theory descriptor. A game is said to be solved if all possible outcomes from all possible states have been predicted by a computer.
- **Decision Tree** – A model for structuring the way game states can be connected. The nodes of the tree represent the various game states, while the edges represent the moves that transition one game state into another.
- **Pruning** – The process of removing game states from a decision tree if it is not beneficial to explore them.

## AI Terms

- **Minimax algorithm** – A type of game-playing algorithm that determines which moves to play based on how disadvantageous the move is for the opponent, while still being as advantageous as possible for itself.
- **Monte Carlo algorithm** – A category of algorithms that depend on repeatedly sampling their search space in a random manner to obtain the necessary results.
- **Monte Carlo Tree Search algorithm** – A type of game-playing algorithm that determines which moves to play by sampling the possible outcomes of each move via repeated simulations.

## Deep Learning Terms

- **Neural Network** – A collection of neurons and links capable of interpreting information passed to it. Requires training before it can be fully useable.
    - **Classification Network** – A neural network that returns a classification or other discrete value for the data it is given.
    - **Regression Network** – A neural network that returns a scalar value for the data it is given.
- **Neurons / Nodes** – The main objects that make up neural networks. Can output signals of varying strength depending on the strength of the input signals.
- **Links** – Connect the various nodes in the network and transmit signals between them. The signal strength depends on the weight of the link.
- **Training Data** – The data set that is provided to a new neural network to configure the weights of its links.
- **Testing Data** – The data set that is provided to a trained neural network to evaluate how well the weights of its links can produce the necessary values/classifications.
- **Epochs** – The number of times that the training data is passed through the neural network.

## Java Terms

- **Exception** – A type of object thrown by Java processes to signify that something has gone wrong. Typically halts the execution of the process when it happens.
- **Garbage Collector** – A process responsible for evaluating whether or not objects in memory are in use. Also clears these areas of memory so they can be returned to the heap.

## Project-specific Terms

- **Decider** – A class representing an algorithm for searching through the possible future game states and determining which moves a player should play. Examples include Minimax deciders and the MCTS decider.
- **Evaluator** – A class representing an implementation of an evaluation function, which AI players can use to determine the worth of a game state. Examples include the Positional evaluator and the Deep Learning evaluator.
- **Simulation** – A possible sequence of game states from a provided starting state to the end of the game. Used by the MCTS algorithm to inform itself about how a game may play out from a certain move or game state.
    - **Heavy Simulations** – Simulations that use an intelligent way (e.g. Minimax searching) of determining the moves a player would play in the future.
    - **Light Simulations** – Simulations that choose the future moves less intelligently or even randomly.

# Table of Abbreviations

| *Abbreviation* | *Long Name* |
|---|---|
| *AI* | Artificial Intelligence |
| *ANN* | Artificial Neural Network |
| *DF* | Data Format |
| *DL4J* | Deep Learning for Java |
| *GUI* | Graphical User Interface |
| *JAR* | Java Archive |
| *JVM* | Java Virtual Machine |
| *MC* | Monte Carlo |
| *MCTS* | Monte Carlo Tree Search |
| *ND4J* | N-Dimensional Arrays for Java |
| *UCB1* | Upper Confidence Bound 1 |
| *UCT* | UCB1 applied to trees |
| *UI* | User Interface |

# Appendices

## Appendix A – Example of Othello Game States

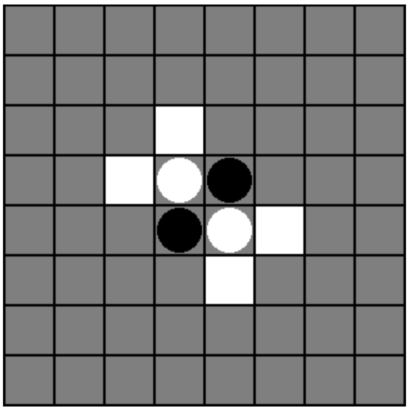


*Figure 27: An initial set up of the game; white squares are where the dark player can potentially place counters, (as they play first) while grey squares are where counters cannot be placed until the game state changes (or because counters already exist there)*

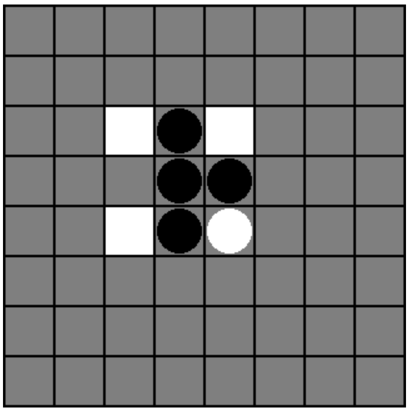


*Figure 28: A state following on from the initial state, where the dark player has placed a counter, resulting in a light counter being flipped to dark. The white grid squares show where the light player can place a counter.*

Example layouts used from https://en.wikipedia.org/wiki/Reversi

## Appendix B – Ted Landau's 21 Key Strategies and Tactics

1. Get More Stable Discs, Not Just More Discs – "…a player will take the move that flips the maximum number of discs. This has been clearly shown to be an inferior strategy." "The point therefore is not simply to acquire discs, but acquire discs that cannot be flipped…"

2. Not All Squares Were Created Equal – "A crucial idea that beginning players inevitably realise is the importance of the 4 corner squares on the board. A corner is important because it can never be flipped. That is, it is a *stable* disc." "In summary, the incorrect strategies place unwarranted value on flipping large number of discs even though they are not stable."

3. Control of the Game: Mobility Optimization and Dynamic Square Evaluation – "Your hope is to get your opponent to make a poor move that will allow you to win the game…. Your goal is to *force* him to make a poor move."

4. Good Moves and Bad Moves: Gaining Control – "… a good move can typically be defined in terms of the principles of gaining and maintaining control (with the ultimate aim of acquiring stable discs)." "… most expert players agree that the easiest way to get control of the game is by maintaining fewer discs that your opponent… This is referred to as *evaporation* strategy."

5. Good Moves and Bad Moves: Planning Ahead – "[Success] depends not merely on what is best for the current position, but what is likely to be best based on what the board will be like 2, 3, 4 or even more moves later."

6. Isolated C-square Traps – "… in this section we demonstrate a "trick" or "trap" that will force your opponent to concede a corner, no matter how many moves he might have available at the time."

7. Unbalanced Edges – "An unbalanced edge is defined as an edge occupied by five adjacent discs of the same colour immediately adjacent to a vacant corner." "Knowing whether or not to take an unbalanced edge, or to attack a currently existing one, are among the more difficult decisions in Othello."

8. Controlling the Main Diagonal: Stoner Traps and More – "Main diagonals can be so powerful, that a good player is constantly on guard for possibilities of gaining or losing control of them…"

9. Gaining and Losing Tempo – "A player is said to gain tempo when he achieves an advantage of timing by deriving one more viable move than his opponent from play within a limited area of the board and thereby forcing the opponent to initiate play elsewhere."

10. Odd and Even Regions of the Board – "… as the game develops, the board is frequently subdivided in separate regions of vacant squares. In this section we discuss a particularly noteworthy aspect of these regions… whether there are an odd or even number of vacant squares in the region." "… it is typically disadvantageous to move into an even-numbered region…"
11. Blocking Techniques: Access and Poisoned Moves – "Blocking techniques refer to ways to prevent your opponent from taking an otherwise good move." "A poison move is [a] type of

blocking move… you do not legally prevent the opponent from moving to a square; you simply make it undesirable."

12. The Semi-Forcing Move – "A semi-forced move is a move which is force not by the rules of the games, but rather by tactical considerations."

13. Mobility Reconsidered – "In some situations, there may be several safe moves that all flip about the same number of discs… How do you decide between them?"

14. The Opening: The Three Basic Variations – "… generalizations of strategy have such limited applicability in the opening, as to be practically useless."

15. The Opening: Quiet Moves – "Players frequently find themselves in the position that anything they do only make their positions worse. Their ideal move would be to pass, but they can't. So they attempt to "disturb" the board as little as possible… these types of moves are called *quiet moves*."

16. Midgame Strategy: Patterns – "In many cases, a global view of the midgame can assist in your decision making. This can be achieved by viewing the board position as one of several possible patterns."

17. The Endgame: The Final Count – "… clearly with only a few moves left, now is the time to go for broke and flip the most amount of discs each turn right? Well, it's almost right. The idea is not necessarily to flip the most amount of discs on each turn, but to flip the most amount of discs relative to your opponent."

18. The Endgame: Counting on More than Just Counting – "Probably the best overall advice for endgame play is "expect the unexpected" … Players need to be on guard for the hidden dangers and/or opportunities that may exist in a given position."

19. Edge Play: General Concepts and Initiating Edges – "As the end of the opening phase of Othello approaches, the players are inevitably presented with the decision as whether or not to occupy an edge square… These next two sections present some useful guide lines for dealing with this issue." "… how does a player decide which edge square move is the best to initiate on?"

20. Edge Play: Developing and Resolving Edges – "Is it a good idea to resolve an edge as soon as you can, or should you leave it unresolved for a while, or even let your opponent resolve it?"

21. Decisive Moves and Prioritising of Moves – "… at some point, a move is made after which, barring any major blunders, the outcome seems decided. This is called the decisive move." "Decisive moves are really a special case of a still more general concept: how to *read the position* and *prioritise* the possible moves."

All names and descriptions are taken from reference [3], *Othello: Brief and Basic* by Ted Landau.

*N.B. What Landau refers to as "discs" are the counters on the board in Othello.*

## Appendix C – The UCT Formula

The UCT formula can be used to determine the most promising node to select when navigating a Monte Carlo Search Tree. The formula is:

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(t)}{n_i}}$$

Where:

- $w_i$ is the number of wins when this node/move is chosen as the next move from the previous state.
- $n_i$ is the number of times this node/move has been chosen as the next move from the previous state, regardless of whether it resulted in a win or loss.
- c is the exploration parameter that effects how often moves that haven't been explored much are chosen. Typically, this is set to the square root of 2, but may change based on the application UCT is used in.
- t represents the total number of times the parent node of the current node has been chosen (i.e. that nodes value for $w_i$)

UCT information used from reference [14]

## Appendix D – BlancheNoire's Command Line Arguments

### Available Arguments

These arguments can be provided directly into the command line, in the format (-x y), where -x is one of the argument labels shown here, and y is a value matching the requested type for that argument.

| Argument Label | Type | Default | Description |
|---|---|---|---|
| **-player1** | String | Human | Defines the type of player to represent player 1. Human players require no extra arguments. |
| **-player2** | String | AI(Random,Score) | Defines the type of player to represent player 2. AI players must be given a decider and an evaluator. |
| **-useGUI** | Boolean | True | Toggles the graphical interface on or off. |
| **-showOutput** | Boolean | False | Tells the program whether or not to display the output the AI players produce in the console. |
| **-archiveGame** | Boolean | True | Determines whether or not the played games are written to an archive file upon completion. |
| **-alternate** | Boolean | False | If set to true, will swap which counters the players control in each game. Only affects runs where -runCount is greater than 1. |
| **-writeStats** | Boolean | False | Determines if the game should record the scores of each player to allow for statistical analysis. |
| **-moveDelay** | Integer | 100 | Number of milliseconds to wait before allowing the next player to move. Allows games state to be viewed before it is changed again. |
| **-AIRunTime** | Integer | 5000 | Number of milliseconds that an AI player is allowed to use to determine the move it wants to play. |
| **-runCount** | Integer | 1 | Number of times to repeat the execution of the Othello games. |
| **-boardSize** | Integer | 8 | The width and height of the Othello board in board spaces. |

### Available Deciders and Evaluators

This is a list of all available Deciders and Evaluators that can be used for Player arguments.

| Decider Name | Argument | Description |
|---|---|---|
| **RandomDecider** | Random | Uses random selection to determine what moves to play. |
| **FixedMinimaxDecider** | FixedMinimax | Searches to a fixed depth and determines what move to play via evaluation of possible moves. Execution can be halted by timeout, resulting in some states not being evaluated. |
| **IterativeMinimaxDecider** | IterativeMinimax | Alternate Minimax decider that iteratively searched to deeper depths until max depth is reached. On timeout, will always return a value for each state. |
| **MonteCarloTree SearchDecider** | MCTS | Chooses moves based on simulations of possible futures from each move. The returned move has the highest predicted victory chance. |

| Evaluator Name | Argument | Description |
|---|---|---|
| **ScoreEvaluator** | Score | Evaluates states based on the difference between the two player's scores. |
| **PositionalEvaluator** | Positional | Determines the worth of a state based on what counters they own and in what positions they are in. |
| **DeepLearningEvaluator** | DeepLearning | Returns the valuation of the games state based on the result of the evaluator's artificial neural network. Must be given the ANN to use as an additional argument. |

### Additional Decider/Evaluator Arguments

This table contains all additional arguments that can be added to Deciders or Evaluators to change their behaviour. There must be no spaces between the main Decider/Evaluator name, the argument labels, and their values.

| Arg Label | Name | Type | Used By | Default | Example | Description |
|---|---|---|---|---|---|---|
| **-D#** | Depth | Integer | MinimaxDeciders | 6 | FixedMinimax-D9 | Defines the max depth that a Minimax AI will search to. |
| **-R** | Use Random Simulations | String | MonteCarloTree SearchDecider | In Use | MCTS-R | Applies a Random Decider for a MCTS AI to use for simulating moves during its decision process. (i.e. sets the MCTS decider to run light simulations) |
| **-M#** | Use Minimax Simulations | String + Integer | MonteCarloTree SearchDecider | Not In Use | MCTS-M7 | Applies a FixedMinimaxDecider for a MCTS AI to use for simulating moves during its decision process. (i.e. |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | sets the MCTS decider to run heavy simulations) Takes a number to specify the Minimax decider's depth (Recommended range for this value is between 2 and 6 inclusively, depending on the power of the computer's CPU). |
| **-S#** | Simulation Maximum | Integer | MonteCarloTree SearchDecider | Infinite | MCTS-S25000 | Sets the max number of simulations a MCTS AI will carry out before halting. If not specified, no limit will be placed. |
| **-P#** | Random Move Probability | Double | MonteCarloTree SearchDecider | 0.10 | MCTS-P0.01 | Specifies the chance that a Random Decider is used to determine moves in the MCTS simulations instead of the provided decider (see -R and -M arguments). |
| **-T#** | Time per Minimax | Integer | MonteCarloTree SearchDecider | 10 | MCTS-T25 | Sets the time the internal decider can spend deciding a move. Larger times result in more accurate simulations, but less of them. |
| **-F(“”)** | DeepLearning ANN Filename | String | DeepLearning Evaluator | None, path is required | DeepLearning-F(ann/net.zip) | Used to specify the file to load the pretrained NN from for the DeepLearningEvaluator. This argument is mandatory. |

## Examples

| Goal | Command |
|---|---|
| **Set Player 1 to be a Human Player** | -player1 Human |
| **Set Player 2 to be a Fixed Minimax AI player with depth 6** | -player2 AI(FixedMinimax-D6,Positional) |
| **Set Player 2 to be an AI using MCTS and DL** | -player2 AI(MCTS-M6,DeepLearning-F(ann/othello_net_second_df3_1.0.zip)) |
| **Turn off the GUI** | -useGUI false |
| **Play 10 games with archiving on, but no move delay** | -runCount 10 -archiveGame true -moveDelay 0 |
| **Run 6 games between a Human and a MCTS AI, with the GUI and alternating players, but with no output, archiving or move delay** | -player1 Human -player2 AI(MCTS-M3-S10000-T5-P0.01,Positional) -useGUI true -showOutput false -archiveGame false -moveDelay 0 -runCount 6 -alternate true |

# References

[1] Albright, Daniel. 26th September 2016. *10 Examples of Artificial Intelligence You're Using in Daily Life* [Online]. Available at: http://beebom.com/examples-of-artificial-intelligence/. Last accessed 31st January 2017.

[2] Wikipedia. *Reversi* [Online]. Available at https://en.wikipedia.org/wiki/Reversi. Last accessed 3rd February 2017.

[3] Landau, Ted. 1987. *Othello: Brief and Basic*, revised edition. Milton Bradley Co. Also available at: http://www.tedlandau.com/files/Othello-B%26B.pdf. Last accessed 7th February 2017.

[4] *Othello Game Rules* [Online]. Available at: http://www.ultraboardgames.com/othello/game-rules.php. Last accessed 3rd February 2017.

[5] Hammond, Kris. 2015. *What is Artificial Intelligence?* [Online]. Available at: http://www.computerworld.com/article/2906336/emerging-technology/what-is-artificial-intelligence.html. Last accessed 7th February 2017.

[6] Wikipedia. *Computer Othello* [Online]. Available at: https://en.wikipedia.org/wiki/Computer_Othello. Last accessed 7th February 2017.

[7] Romano, Benedetto. 2nd January 2011. *Othello Programming* [Online]. Available at: http://www.romanobenedetto.it/index.htm. Last accessed 7th February 2017.

[8] Nijissen, J.A.M. 22nd June 2007. *Playing Othello Using Monte Carlo* [Online]. Available at: https://project.dke.maastrichtuniversity.nl/games/files/bsc/Nijssen_BSc-paper.pdf. Last accessed 7th February 2017.

[9] Wikipedia. *Game theory* [Online]. Available at: https://en.wikipedia.org/wiki/Game_theory. Last accessed 8th February 2017.

[10] Wikipedia. *Solved game* [Online]. Available at: https://en.wikipedia.org/wiki/Solved_game. Last accessed 8th February 2017.

[11] Wikipedia. *Monte Carlo method* [Online]. Available at: https://en.wikipedia.org/wiki/Monte_Carlo_method. Last accessed 9th February 2017.

[12] Frontline Solvers. *Monte Carlo Simulation* [Online]. Available at: http://www.solver.com/monte-carlo-simulation-overview. Last accessed 9th February 2017.

[13] Weisstein, Eric W. *Monte Carlo Method* [Online]. Available at: http://mathworld.wolfram.com/MonteCarloMethod.html. Last accessed 9th February 2017.

[14] Wikipedia. *Monte Carlo Tree Search* [Online]. Available at: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. Last accessed 10th February 2017.

[15] Copeland, Michael. 29th July 2016. *What's the Difference Between Artificial Intelligence, Machine Learning and Deep Learning?* [Online]. Available at: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/. Last accessed 10th February 2017.

[16] Wikipedia. *AlphaGo* [Online]. Available at: https://en.wikipedia.org/wiki/AlphaGo#Match_against_Lee_Sedol. Last accessed 10th February 2017.

[17] Wikipedia. *Minimax* [Online]. Available at: https://en.wikipedia.org/wiki/Minimax. Last accessed 14th February 2017.

[18] Lea, Patrick. 8th March 2012. *Computer Reversi, Part 1.5: The Thor Database* [Online]. Available at: http://ledpup.blogspot.co.uk/2012/03/computer-reversi-part-15-thor-database.html. Last accessed 25th April 2017.

[19] Bernhardsson, Erik. *Deep Learning for... chess* [Online]. Available at: https://erikbern.com/2014/11/29/deep-learning-for-chess.html. Last accessed 26th April 2017.