Development of an Artificial Neural Network to Play Othello

Submitted May 2009, in partial fulfilment of the conditions of the award of the degree BSc (Hons) Computer Science

Alexander J. Pinkney
(ajp06u)

Supervisor: Prof. Graham Kendall

School of Computer Science and Information Technology
University of Nottingham

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

Signature: _____

Date: ____ / ____ / _____

Abstract

The aims of this dissertation are twofold: firstly: to program an Othello engine which can be played by two human players, which will provide information to the players such as whose go it is, the current state of the board, and where legal moves can currently be made; and, secondly: to develop an Othello playing algorithm which has no prior knowledge of any tactics or strategies, which is capable of consistently beating human players and other artificial intelligences.

The former is not necessarily a prerequisite for the latter, but will certainly be helpful for observing matches, and to act as referee, ensuring that the rules are enforced; for example, only legal moves should be allowed, and the board state should be correctly altered after a move has been made.

It is proposed that the latter is to be developed by evolving a population of artificial neural networks which repeatedly play Othello matches against one another. After each network has played every other a certain number of times, the poorest players (determined by numbers of wins, losses and draws) will then be 'killed off' and the strongest players duplicated and slightly modified, and so on, in effect, recursively generating better and better players, until such time as the system reaches a plateau.

Contents

1 Introduction

1.1 Othello

Othello is a two-player game which has existed in its current incarnation since the 1970s, and was popularised in Japan. The game was heavily based on *Reversi*, a game developed in the late 19th century by two Englishmen.[1]

The game is played on an 8×8 board, which is usually green. All playing pieces are identical and reversible, with one side black, and the other white. Pieces are usually referred to as *black* or *white*, according to the side showing face-up — this convention is adhered to throughout this report. The starting board configuration is shown in figure 1, below.

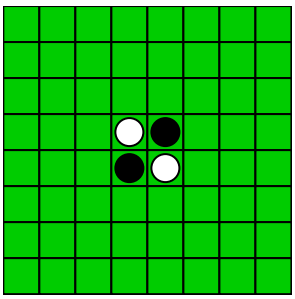

Figure 1: Initial Othello board configuration

On a player's go, they must place one piece of their colour on the board, in a square which encloses one or more pieces of the opponent's colour between one or more of their own, horizontally, vertically, or diagonally. All these enclosed pieces are then 'captured', whereby they are turned over to show the opposite colour. If a player cannot make a legal move, they simply miss their go. The game ends when there are no legal moves available for either player; this can be when the board is filled, when one colour is completely eliminated, or simply that no empty square can be used to capture pieces of either colour. The winner of the game is the player with the most pieces of their colour on the board at the end of the game.

[1] The only two notable differences between Reversi and Othello is that Reversi had no starting board configuration, i.e. the players could place their first pieces wherever they chose, and that each player had only a fixed number of pieces each, as opposed to the pool of pieces used in Othello. If one player ran out, the remaining player was entitled to play the remainder of their pieces.

1.2 Motivation

Until fairly recently, many board game playing "artificial intelligence" systems[2] were artificial only in the sense that their decision making process was not taking place within a human brain. Any tactics and strategies needed to be hard-coded by the programmer. Clearly, reliance on known strategies means that an AI can only be as strong as the strongest human player — likewise it is not exhibiting its own intelligence: it merely blindly follows the rules which its programmer has set down for it. Other AIs had been developed which learned on their own, however in the majority of cases, these AIs were told by their programmer which features of the game were important, and these in turn were based on human-developed strategies.

Human knowledge has an unfortunate tendency to go hand-in-hand with human error, and absolute reliance upon it is not always advisable. This was exemplified beautifully in August 1992 in a checkers[3] match between checkers world champion Dr. Marion Tinsley and Chinook, a computer checkers player. In the fifth game played, Chinook made a move based on one "taken from one of the major books on the checkers openings"[12] and then went on to lose. The author of the book later admitted that the inclusion of that particular move was erroneous[12], leading to the conclusion that even experts should not be considered to be infallible.

Rather than relying on existing knowledge of a game, perhaps every position in a game can be analysed to determine absolutely which move is the best to make. Many simple games can be 'solved' (by determining whether any given position leads to a win, lose, or draw) by completely expanding their respective game trees; that is, every possible game state which can be reached is examined, and assigned as a winning, losing, or drawing position. Assuming the game had been solved, writing an AI to play it would be trivial — the game could be won simply by moving into a 'win' position on every turn, or a 'draw' position, should 'win' not be available. In an ideal world, the game tree of Othello would be completely expanded as part of this dissertation, to allow the creation of a 'perfect' AI. Unfortunately, the game tree complexity of Othello has been estimated at approximately 10^{58}[1]. Even with the (extremely) optimistic target of reaching one board state per nanosecond, expanding the entire game tree would take in the order of 10^{41} years. The age of the universe is currently estimated as being around 14 billion years. To put things into con-

[2] hereafter referred to as *AI*s
[3] generally known as *draughts* in the UK

text, expanding the entire game tree of Othello at would take in the region of 10,000,000,000,000,000,000,000,000,000,000 *times the current age of the universe*. Clearly this approach is unfeasible.

One way to develop a computer-based AI which has the potential to beat even the best humans is by developing an algorithm which 'learns' on its own without any human input, beyond an indication of what is or is not a legal move, and whether or not a completed game has been won, lost, or drawn. An algorithm which performs in this manner may discover as-yet unknown tactics, particularly if these involve counterintuitive aspects. Human players tend to learn by copying the strategy of more experienced players, however this does not often lead to the discovery of novel modes of play. An AI which learns only on the basis of its win/loss ratio and does not rely on mimicking other players therefore has an inherent advantage as it can develop its own strategies and tactics.

Historically, the use of self-learning artificial neural networks as board game AIs has always been overshadowed by those AIs which are taught entirely by their programmer. This is perhaps due to the large amount of processor time needed for a neural network to evolve to a sufficient level — in fact in [9], Arthur Samuel states that he specifically avoided the use of a neural network approach for his checkers program, citing the impracticality of this approach given the technological resources available at the time. Since Samuel wrote this article, computing power has increased significantly, and its cost dropped dramatically, allowing development of a neural network to be possible on a home PC and within a reasonable timescale. What may have taken months only a few years ago may now be achieved in a matter of days.

It is for the above reasons that the author has decided to create an AI to play Othello using a self-learning artificial neural network. Othello was chosen in particular, since its rules are simple, the branching game tree is reasonably small, and every game is finished after at most 60 moves. Chess, by comparison, is exceptionally complex, especially when taking into account the lack of homogeneity amongst its playing pieces; chess's game tree complexity has been estimated around 10^{123}[1]. Checkers was considered, but discounted since its rules are more complex than those of Othello, which would be likely to steer the focus of this dissertation away from creating an effective neural network-based player, and toward simply implementing the rules of checkers. The game tree of checkers is also complicated in the latter stages of the game, once pieces become kings and are able to retrace their steps. Othello was thought to provide a good balance between game tree complexity (i.e. not

trivial, but not too complex) and rule complexity.

1.3 Description

1.3.1 Othello Game Framework

An Othello game framework is to be developed which supports an interface for two players to compete. The framework should act as referee, ensuring only legal moves may be made, and should alter the board state accordingly after every move is made. It should also store various information regarding the state of a game, such as which player's turn it is and the current state of the board. The framework should prompt players to make a move when it is their turn, providing the player a list of legal moves from which to pick. For the purposes of convenience, there should also be a way of graphically describing the current state of a game, so that a human player may either observe or partake in the playing of a game.

1.3.2 Artificial Neural Network Population

It is intended that all neural networks will play one another twice (alternating which has the first move) and in addition, all will play several times against a 'random mover', which is intended to act as a control sample, so the progress of each generation of neural networks can be monitored. Once all these matches have been played, the networks will be ordered according to their total wins, losses and draws, against one another (i.e. the games versus the random player will not be taken into account for this) and the poorest players eliminated.

It is intended to store a sample (if not all) of every generation of neural networks so that the progress of their evolution can easily be followed and monitored. A mutation algorithm will need to be derived to actually allow the networks to evolve — this will need to be carefully crafted such that neither too much modification is made nor too little. It is possible that some sort of 'cross breeding' algorithm will be devised to combine the features of the best playing networks, and it is likely that this will be combined with a small amount of random mutation to ensure continual change.

In addition to a strong evaluation function, various other techniques exist which can strengthen the performance of a game-playing AI. For example, expanding the game tree by the depth of even a few moves can provide a considerable advantage. Methods of reducing the needed expansion of the game tree, such as $\alpha\beta$-pruning can speed up this process, as the subtrees can be rather large (as

noted in section 1.2).

The use of an opening book and an endgame database is not intended for inclusion within this project. Using these (in the opinion of the author) rather defeats the object of developing a purely positional evaluation function. The emphasis of this project is on the development of an Othello player which learns by itself, rather than simply the development of an expert player.

2 Related Work

2.1 Literature Review

In this section, a selection of related works are reviewed and their relevance to this project discussed. Most original work in this field is related to checkers, but some Othello players are also discussed.

2.1.1 Arthur Samuel's Checkers Player

The first notable self-learning game playing AI was developed in the 1950s by Arthur Samuel (see [9]), whose program was developed specifically to play checkers. His work in this field was groundbreaking — previous artificial intelligences required the parameters of their evaluation function to be painstakingly tweaked to perfection by hand. Samuel's checkers player was unique in that it adjusted its own parameters as it gained experience of more and more games.

Samuel identified several features of a checkers game which he regarded as being correlated with a win, such as the piece ratio, or the number of moves available to either player. Whilst his method was revolutionary, Samuel had unwittingly imparted his perspective of a checkers game on his program; the very fact that he had identified these parameters to his program meant that its learning method was inherently based on how the game was already played by humans.

Samuel was aware of the shortcomings of his solution, however he cited the inefficiency of neural networks as his reason for producing "the equivalent of a highly organised network which has been designed to learn only specific things" instead.

2.1.2 Chinook

Chinook was another checkers program, developed in the late 1980s and early 1990s. This relied on "deep search, a good evaluation function, and endgame databases"[10] to play, and also used an 'opening book', which contained opening moves taken from checkers literature which were considered to be strong.

Similarly, the program's evaluation function was trained using documented games played by grandmasters, and thus Chinook mimicked their playing style. Whilst this certainly ensured that Chinook could beat the majority of players, it struggled when it came to playing the very top checkers players of the time. Whilst it did not often lose a game against very strong players, it similarly did not often win, and many games remained draws. Chinook was beaten by very narrow margins by both Dr. Marion Tinsley (regarded as the greatest checkers player who ever lived, who reputedly lost a total of 5 games in the period 1950–1992[11]) and Don Lafferty, the world's second best player at the time.[11]

As mentioned in section 1.2, Chinook famously lost a match against Tinsley in 1992 due to an erroneous set of opening moves stored in Chinook's opening book. In [10], Schaeffer states that it was well known by the developers that Chinook's opening book was its weak point, and that work was being carried out to remedy this.

2.1.3 IAGO

IAGO was developed in the early 1980s, and was the first notable program which played Othello at a non-trivial level. Its evaluation function was constructed in a similar way to that of pre-Samuel AIs (see section 2.1.1) in that it was entirely hand crafted, and had no self-learning abilities[2, 8].

2.1.4 BILL

BILL was the first program to beat IAGO — in fact, it did so in spectacular fashion, beating IAGO in every match played, with only 20% of the thinking time.[2] Unlike IAGO, in addition to its evaluation function, BILL had a built-in learning function: it stored every game position it encountered, and assigned them as winning or losing, according to the outcome of the game. Using these, it learned to recognise patterns of winning and losing positions, and altered its evaluation function accordingly. One drawback of using this method was that no non-expert games were used to train BILL, and therefore against a novice player BILL did not have such an edge.

2.1.5 Logistello

Logistello was released in 1994[2], and is currently one of the strongest Othello-playing programs. Whilst it is self-learning, it does not identify features which it considers to be important — these are still specified by its programmer. However, what Logistello does do, is to decide how *combinations* of features are important — that is, unlike BILL, which treats every feature separately. Additionally, Logistello uses both an endgame database to ensure a perfect endgame, and an opening book to sway the odds into its favour from the outset.

2.1.6 Anaconda

Anaconda is another checkers player, developed in the late 1990s; its evaluation function was developed by evolving a population of artificial neural networks. It was provided with no knowledge of how to play checkers, besides being supplied with the rules, and evolved to the extent where it had identified its own features which it considered to be correlated to a win. After 840 generations, Anaconda had evolved to a level considered to be *excellent*[3].

2.1.7 Moriarty and Miikkulainen

Moriarty and Miikkulainen[7] developed an Othello player in a similar manner to Anaconda, and in a similar manner to the way in which this project is intended to be developed, in that no strategies or tactics were provided to the program. The program was unusual in that it did not expand the game tree at all, beyond looking at the currently legal moves, and despite this, it independently developed an "advanced mobility strategy"[7], which Moriarty and Miikkulainen state was discovered only once, and not, like many strategies, discovered in several places independently. They go on to state that the rediscovery of this method was "a significant demonstration of the potential power of neuro-evolution"[7] and that mobility strategies are notoriously difficult to learn, particularly as they involve counterintuitive aspects, such as minimising your own piece count during the middle-game. This, however, restricts the legal moves available to the opponent, thus providing an advantage as they are forced to make 'bad' moves.

It is stated in this paper that their population of neural networks was evolved initially against a random moving opponent, and then later against players af-

forded with $\alpha\beta$ search capability. The neural network population then "evolved to exploit their initial material disadvantage and discovered the mobility strategy".

Evolving a position evaluation function in a manner akin to Moriarty and Miikkulainen's would be an ideal aim for this project.

2.2 This Project in Context

It is intended to follow the same route as Anaconda and Moriarty and Miikkulainen, in that the population of neural networks will not be provided with any prior knowledge of Othello. Unlike Logistello, an opening book and endgame database will not be used, as these will not help the program achieve 'intelligence', however some game tree evaluation is likely, but certainly nothing on the scale of Chinook.

3 Design

3.1 Representing a Game of Othello

3.1.1 Game Framework

The first major milestone in this project is the implementation of a computerised Othello framework. As stated in section 1.3.1, this will need to be made aware of the rules of Othello; i.e. it must be able to recognise legal moves, and to alter the board state correctly when a player has made their move.

The most base element of an Othello game is the board itself — intuitively, therefore, this is the best place to start.

3.1.2 The Othello Board

An Othello board is made up of 64 separate spaces, each of which may be in one of three states: occupied by a white piece, occupied by a black piece, or empty. It was decided to represent the board as an array of 64 variables. Initially, this array was intended to be two-dimensional (8×8), as this closely parallels a real board, and allows co-ordinates of a board space to correspond directly with array indices. After some deliberation, however, it was decided to use a one-dimensional array, since the current board state is likely to be used by many processes, and in some cases will be required to be modified. Java provides an `Array.clone()` method; however this only works correctly with one-dimensional arrays[4] (although it would be trivial to implement such a method). Also, the input layer of the neural networks (see section 3.1.5) is made up of a one-dimensional array of 64 values. In the interests of simplicity, therefore, a one-dimensional array was chosen.

Since each position on a board has three possible states, it was decided to make the board state array up of `int`s, with `0` representing an empty position, `1` representing a position occupied by a black piece, and `-1` representing a position occupied by a white piece.

[4] Java's multidimensional arrays are not truly multidimensional — they are actually arrays of arrays, and therefore Array.Clone() only clones the first dimension.

3.1.3 The Player

A two-player game such as Othello clearly requires some way for a player (computer or human) to interact with the game, both to evaluate the board state, and to select a move to play. It is therefore envisioned that a player object will be defined, such that a game takes two player objects, and alternately requests each player object for a move, should a legal move for that player be available. Whether the player is human or computer, the game framework should ideally treat them indistinctly. The best way to do this is to define a generic player interface, which human and computer players can implement. A computer player, for instance, when prompted for a move, will need to use some form of *evaluation function* (see section 3.1.5) to select a move, however a human player will need to be prompted to enter the move they wish to make[5], using the keyboard or mouse. In either case, once the player has selected a move, an `int` corresponding to the chosen board position can be returned, and the board state updated accordingly.

3.1.4 Moves

In order that the player need not distinguish legal moves from illegal, it is intended that when requesting a move from a player, they will be provided with a list of legal moves, in the form of an array of 64 boards. Entries in this array which are `null` represent that moving in the corresponding positions is illegal; entries containing a board object show what the new board state would be if the player were to move in the corresponding position. This has the advantage that the player does not need to be aware of the actual mechanics of the game, and also eliminates any risk of the player and the game framework differing in their opinions of what the updated board position should be[6]. Additionally, the player does not waste time needlessly evaluating illegal moves.

Various (slightly vague) mentions have been made so far regarding updating of the board state after a move has been made. Clearly, this is a pivotal element of the Othello game framework. Before the act of updating the board state following a given move can even be considered, the legality of the move must first be established. If the board position requested to be played in is not empty, the move is illegal. If it *is* empty, then each of the eight directions emanating from the position must be checked to see if pieces can be captured. The move is

[5] A human player effectively acts as their own evaluation function, however for the purposes of programming, it is easier to make a clear distinction between computer and human players.

[6] For instance, in the case of programmer error.

legal if at least one piece may be captured in at least one direction. Armed with information on which directions pieces may be captured in, the board state may now be updated, initially by placing a piece of the player's colour in the requisite position, and then by recursively 'flipping pieces' (i.e. negating their values in the board state array) in each valid direction, until another piece of the player's own colour is encountered.

3.1.5 Evaluation Functions

In the context of this project, an evaluation function simply takes a board state, and somehow converts it into a single numeric value, representing the 'goodness' of the state. A computer player has an evaluation function, and uses this to evaluate the result of each legal move it can make. The move yielding the highest[7] value is selected. In the case that two or more moves yield the same, highest value, the player simply picks one of these moves at random. Three different evaluation functions are intended to be developed: the random mover, the greedy mover, and the artificial neural network mover.

Implementing the random mover is trivial: since the player will choose randomly from equally weighted moves, the random mover simply needs to assign the same value regardless of the board state.

The greedy mover is defined as favouring board states in which its piece count is highest. Again, this is reasonably simple to implement — the greedy mover can just return the ratio of its pieces to its opponent's.

The artificial neural network mover is somewhat more complex, and is explained in detail in the following section.

3.2 Artificial Neural Network

Artificial neural networks are made up of a collection of nodes (or *neurons*) and links between them. Each node takes a numeric input, and provides a corresponding numeric output, based on its *threshold* and *activation function*. Outputs of nodes may be linked to the inputs of others; each link is directional, and has an associated *weight*. This weight is used to alter the value travelling along the link, most commonly by being used as a multiplier.

[7] The choice of a high value representing a good move is arbitrary — equally, the lowest value could be defined as 'good'.

3.2.1 Activation Functions

These networks can be used to represent complex, nonlinear functions, due to the nonlinear nature of their activation functions. The most commonly used activation functions are the sigmoid function,

$$f(x) = \frac{1}{1 + e^{-x}},$$

and the hyperbolic tangent function,

$$f(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

The graphs of these functions are shown in figures 2 and 3.

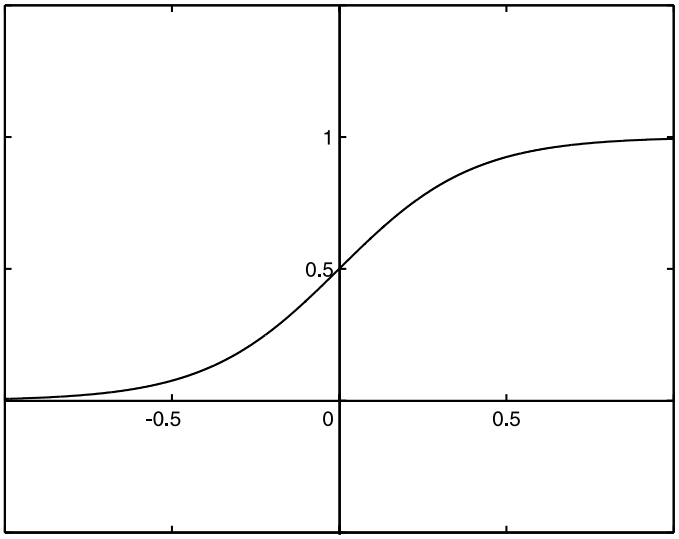

Figure 2: Sigmoid function

Since the inputs to the neural network will be made up entirely of 0, 1, and -1 (values taken directly from the board state array) it was decided that the hyperbolic tangent function would be used. This is due to the fact that it is an odd function[8], and thus rotationally symmetric about the origin, paralleling the inputs.

[8] An *odd function* is defined as follows: for a function $f(x)$, f is odd if $-f(x) = f(-x)$ for all x in the domain of f.

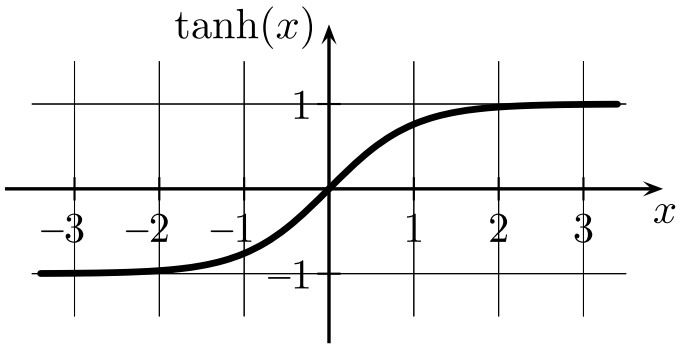

Figure 3: Hyperbolic tangent function

3.2.2 Design

The form of artificial neural network intended to be used in this project is known as a *multilayer perceptron*, which is a type of feedforward network[9]. Multilayer perceptrons are composed of an input layer of nodes, one or more hidden layers, and an output layer. Each of these layers is fully connected to the successive layer — that is to say, there is a link from *every* node in one layer to *every* node in the next.

The universal approximation theorem [5] states that any continuous function that maps an interval of real numbers to some output interval of real numbers can be modeled arbitrarily closely by a multilayer perceptron with just one hidden layer. Thus it was decided to design the network as follows:

- Input layer, comprising of 64 nodes, fully connected to:
- Hidden layer, comprising of 42 nodes, fully connected to:
- Output layer, comprising of one node.

The size of the hidden layer was chosen on the basis of a rule of thumb that it should be approximately $\frac{2}{3}$ the size of the input layer. The output layer consists of only one node, since only one single numeric value is required as output. The resulting neural network resembles that shown in figure 4.

As shown in figure 4, with 64 nodes in the input layer and 42 in the hidden layer, a total of 2688 edges (links) are required to fully connect layers 1 and 2. These are represented by 2688 weights: one per edge. Similarly, the 42

[9] Feedforward here refers to the acyclic nature of the layout of the neural net.

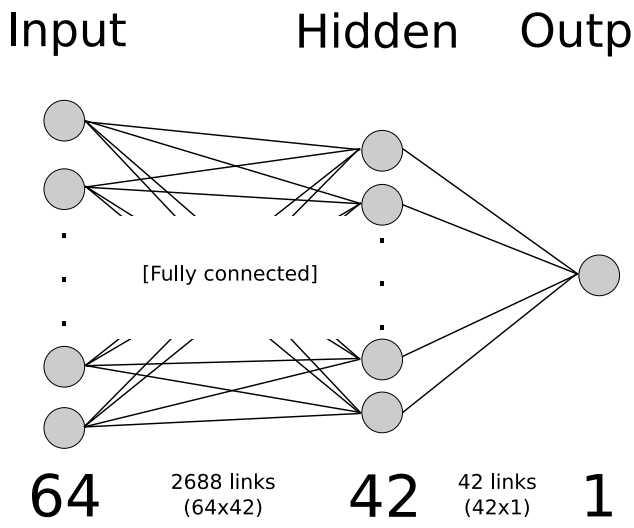

Figure 4: Artificial Neural Network Design

graph edges between layers 2 and 3 are represented by 42 weights. In addition to these weights, each of the 42 nodes of the hidden layer requires a threshold value; this effectively scales the function along the x-axis, providing a smoother or sharper curve. The weights and thresholds are applied as follows.

The output of each node in the input layer is the same value as is input. Since the input values are discrete, an activation function is superfluous, and simply applying a weight along each output edge performs adequately.

The output of each node in the hidden layer is defined as

$$output = \Psi(\varphi, t)$$

where

$$\Psi(\varphi, t) = \frac{e^{t \cdot \varphi} - 1}{e^{t \cdot \varphi} + 1}$$

and

$$\varphi = \sum_{i=0}^{n-1} x_i w_i \,.$$

Here, Ψ is the activation function, φ is the weighted sum of all n (64) inputs (x is the input value and w is the weight associated with the corresponding edge), and t is the threshold value associated with the node.

The output of the node in the final layer is simply the sum of its weighted inputs. Since there is only one node in the layer, it was decided that there was

no need to apply an activation function. The output can therefore be defined as

$$output = \sum_{i=0}^{n-1} x_i w_i$$

where x is the input value, and w the weight corresponding to the associated edge, with n inputs (in this case 42).

These values of weights and thresholds will be stored in three arrays of `doubles`: one holding the weights between layers one and two, one holding the threshold values for layer two, and one holding the weights between layers two and three.

3.2.3 A *Good* Player?

The precise values of the weights and thresholds required to make a good player is unknown — indeed, achieving this is one of the main focuses of this dissertation. Initially, there is no better method of setting these values than simply assigning random numbers. The intention, therefore, is to do so, assigning random numbers from a Gaussian distribution, with mean zero and standard deviation 1. Using a Gaussian distribution theoretically allows for any number, however, statistically, the vast majority of values produced[10] will be within three standard deviations of the mean. Discovering which configurations of weights and thresholds make an effective player will effectively be left to chance, as discussed in the following section.

3.3 The Evolution Process

It is intended to maintain a population of 20–30 neural networks as described in the previous section. In the first instance, all the weights and thresholds will be randomised using the aforementioned Gaussian distribution random number generator, resulting in a population of players whose ability is likely to vary from reasonably good, to reasonably bad. This random population may be considered as the first generation of many in the evolutionary process.

[10] Approx. 99.7%

3.3.1 Ranking System

Once this population of neural networks has been established, there must be some method whereby it can be decided which are better at playing Othello than others. It has been decided that a form of tournament system will be used, in which each neural network plays each other twice, having the first move in one instance, and the second in the other. Each network will receive a score according to the output of each game — plus one point in the case of a win, and minus one in the case of a loss. No points will be added or subtracted in the case of a draw. After these games, the networks will be ranked according to their scores, and the bottom-scoring half of the population will be eliminated.

3.3.2 Mutation Algorithm

To compensate for this elimination (and indeed, to actually perform the process of evolution) each network in the top-scoring half of the population will be duplicated; this duplicate replacing one of the eliminated nodes. To allow the population to 'evolve' slightly every generation, each of these duplicates will have each of its weights and thresholds altered by a random amount, in the following manner.

$$X_{n+1} = X_n + \mu \cdot \Gamma(0, 1)$$

where n represents the current generation, X is the value to be mutated (either a weight or a threshold), μ is the mutation factor, and $\Gamma(0, 1)$ is a function which produces a random number from a Gaussian distribution with mean zero and standard deviation 1.

The inclusion of μ, the mutation factor is to control the rate of mutation. Java's `Random` object only produces Gaussian values with a mean of zero and standard deviation of one. This on its own would elicit far too high a rate of mutation[11], therefore the mutation factor is used to (in effect) reduce the standard deviation of the Gaussian function, producing a narrower bell curve. It was suggested that a mutation factor of 0.05 would be a sensible value to use.

[11] A rate of mutation which is too small would lead to *very slow* evolution; conversely too large a rate may lead to values which have little relation to their previous incarnations, effectively just producing a random number. The mutation factor must be chosen carefully, striking a balance between speed and accuracy.

3.3.3 Benchmarking

In order that it can be inferred that the population of neural networks is actually evolving into a better player, some form of benchmark test must be used. Since the neural networks evolve by playing against each other, it is difficult to tell simply from these games that . For this reason, it was decided that the top-scoring neural network in each generation would play 1000 games versus a random moving opponent (see section 3.1.5), using a similar scoring method to that of the tournament system. Theoretically, therefore, a 'better' neural network should win proportionately more games versus a random mover thana 'worse' one. These scores will be output to a .csv file[12] so that the progress of each generation of neural networks can be tracked and plotted to a graph.

3.4 Game Tree Expansion

Whilst not intended for use during the evolutionary stages of the project, expansion of the game tree is an important feature of many board game playing AIs. Use of lookahead, even by only a few moves, offers considerable advantage, especially when used in conjunction with minmax evaluation, whereby the player may seek to minimise their maximum possible loss. The efficiency of the minmax evaluation can be improved by employing $\alpha\beta$ pruning. This allows the expansion of certain subtrees to be halted, if it can be proved that the move is worse than one previously examined; in many cases, this will vastly decrease the space complexity of the game tree produced, and will, in the worst-case scenario, produce the same game tree as a minmax evaluation.

An $\alpha\beta$ search tree algorithm will be produced, which will enable any computer player to use lookahead to an arbitrary depth[13], which can be used in the later stages of the project, for single player games, once a neural network which can play Othello to a reasonable level has been developed.

[12] A **c**omma **s**eparated **v**ariable file, readable by many common spreadsheet packages.
[13] There will, however, clearly be constraints based on reasonable use of time and memory.

4 Implementation

4.1 Programming Language

The only language in which the author currently feels comfortable programming such a large project as this is Java. In an ideal world, C++ would have been used[14], due to the various speed and efficiency advantages it offers. Nevertheless, Java *was* the language chosen for this project — its strong object-oriented nature does lend itself to the construction of well designed programs, and its inherent cross platform nature means that almost any code can run on any machine with a JVM installed.

4.2 Class Overview

Figure 5 displays the basic relationships between each Java class. The main method is contained within the `Othello` class. Each other class represents a concrete implementation of an object, with the exception of `OthelloPlayer` and `EvaluationFunction`, which are interfaces to allow different implementations of different types of players and evaluation functions respectively. It was intended to produce a more detailed discussion of each class for inclusion in this section, however the code is quite well commented, with reasonably intuitively named methods and variables, and in most instances, classes are structured as described in the design section. Suffice to say, what follows is a brief overview of each class and its function within the project.

4.2.1 Othello.java

This class contains the main method. Various command line arguments allow different functions to be performed. For instance, running `java Othello sp` initiates a single player game.

[14] In an ideal world, C++ would have been taught as the staple language of the first year Computer Science course, and Java offered as an optional second year module, as opposed to the other way around... however, this is probably not the place for such musings!

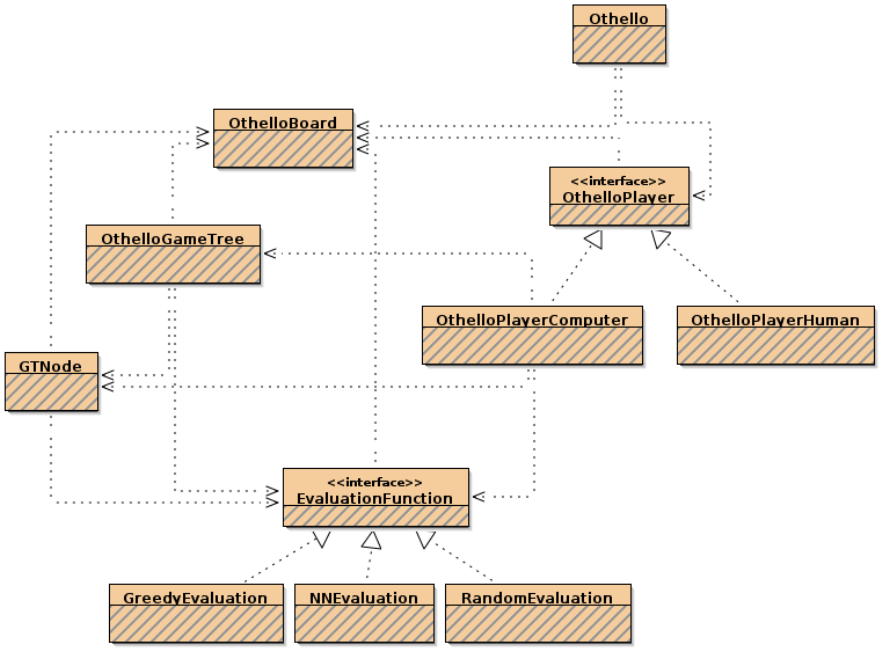

Figure 5: Class Diagram

4.2.2 OthelloPlayer.java

This is an interface implemented by OthelloPlayerComputer and OthelloPlayerHuman. It defines the `int getMove(OthelloBoard[] moves, int colour)` method, which is implemented separately in the human and computer player objects.

4.2.3 OthelloPlayerComputer.java

The constructor is of this object takes an EvaluationFunction object, and an integer value defining its game tree search depth beyond the next move. `getMove()` utilises the evaluation function on an $\alpha\beta$ pruned game tree expanded to the requisite depth.

4.2.4 OthelloPlayerHuman.java

The constructor of this object takes a String as input, which can be used to set the player's name, for display during games. `getMove()` requests input from

the user via the `UserInput` object. (This was provided for use in first year during the G51PRG module, and it has been assumed it is OK to use here.)

4.2.5 EvaluationFunction.java

This is an interface implemented by the various evaluation function subclasses (GreedyEvaluation, RandomEvaluation, and NNEvaluation). It defines the `double evaluate (OthelloBoard board, int colour)` method, which is implemented separately by each subclass.

4.2.6 GreedyEvaluation.java

This implements the `evaluate()` method defined in the Evaluation function interface by simply returning the ratio of friendly to enemy pieces, thus favouring positions with greater proportions of 'friendly' pieces.

4.2.7 RandomEvaluation.java

This implements the `evaluate()` method defined in the EvaluationFunction interface by returning the value `1` regardless of the board state. The OthelloPlayerComputer class, which makes use of evaluation functions, randomly picks from the best available moves, in the case that two or more moves have the same, highest heuristic value.

4.2.8 NNEvaluation.java

This implements the `evaluate()` method defined in the Evaluation function interface in the manner described in section 3.2.

NNEvaluation also has a method named dumpToDisk() which, when called, writes the threshold and weight values to disk, for the purposes of backup.

4.2.9 OthelloBoard.java

This is the class which contains most of the implementations of the rules of Othello (with the exception of maintaining state of which player's turn it is,

which is performed by the playGame() method in Othello.java). It is this class which states whether a given move by a given player is legal, provides a list of legal moves for a given player, provides information on the state of the game (such as a given player's piece count), and actually performs the act of updating the board state when a move is chosen by a player. The current board state may be output to the console in the form of an ascii board (by calling the method `printTextBoard()`), as shown in figure 6.

```
   0 1 2 3 4 5 6 7
 + - - - - - - - - +
0|     *           |
1|       *         |
2|   0 0 0 0 0     |
3|       * 0       |
4|     * * *       |
5|       *         |
6|       *         |
7|                 |
 + - - - - - - - - +
```

Figure 6: Sample of ascii board representation

4.2.10 OthelloGameTree.java and GTNode.java

These are the classes responsible for creating and expanding game trees. OthelloGameTree.java creates trees from `GTNode` objects, which form a doubly-linked[15] tree. `GTNodes` contain an `OthelloBoard` object, on which the expansion of their children is based, and state to define the node as being 'Max' or 'Min' for the purposes of $\alpha\beta$ pruning.

The two notable methods in OthelloGameTree.java are `expandwithAB(GTNode node, int colour, int depth, EvaluationFunction ef)` which (intuitively) expands a GTNode with $\alpha\beta$ pruning to a certain depth, using a certain evaluation function, from the point of view of a certain colour of piece (i.e. a player); and, `printXML()` which prints the structure of a game tree as XML to the console, by traversing it in a depth-first manner. Attributes such as whether a node is min or max, and the assigned heuristic value of the node may be included. With a bit of trickery using XSLT, Troff and pic (as taught in the G52DOC module) this XML can be converted into a visual representation of the game tree. A sample trees is shown in figure 7.

[15] i.e. traversable in either direction

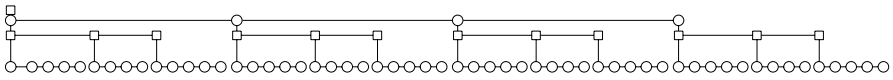

Figure 7: Game tree of Othello, expanded from initial board configuration by a depth of 3

4.3 Timekeeping and General Comments

The programming of the initial Othello framework actually took a surprisingly small amount of time — indeed, getting to the stage where a two player game could be played only took a matter of a few hours. The introduction of the two simple evaluation functions, `GreedyEvaluation` and `RandomEvaluation`, allowing single player games versus the computer (or for the computer players to play each other or themselves) took only a short time more.

Methods of game tree expansion took a little longer. Firstly, a method was written which expanded the game tree (using depth-first search), and then retrospectively applied a given evaluation function to the leaf nodes, and propagated these values up the tree in a Minmax fashion. Whilst this did work, it did not lend itself well to the implementation of $\alpha\beta$ pruning, since this requires that leaf nodes are evaluated and their values propagated upwards whilst the tree is still being expanded. Perfecting this method, and checking it was working correctly was reasonably time consuming: in the order of a few weeks.

Developing the neural network-based evaluation function also turned out to be a lot simpler than anticipated. Once it had been established that the problem essentially boiled down to enumerating the hidden layer (for each node in the hidden layer, summing all the weighted inputs, and then applying the activation function to this number), then enumerating the output layer (in a similar manner, using this time the enumerated values of the hidden layer) and then returning this value, the whole evaluation function was developed remarkably quickly. This did seem somewhat anticlimactic — it had been anticipated that developing the neural network evaluation function would be a long and arduous process. It turned out that the arduousness was yet to come.

The element which took longest was undoubtedly the actual stage of evolution. This was not only due to the computationally-intensive nature of the process, but also because there were a fair few false starts in which no noticeable evolution was observed. It took several weeks, a lot of head scratching, and a fair amount of swearing at the computer to finally iron out all the problems (see section 4.4.1) but once this had been done, significant evolution was observed

after the first few hundred generations.

4.4 Testing

The modularity of the system design lent itself well to continuous testing as each component was programmed. It was reasonably easy to ensure that every component behaved as expected, and without any side effects, during the development process. There was one major bug present in the version of the program which had been, up to that time, considered to be 'final'. It is covered in detail in the next section.

4.4.1 Problems and their Solutions

The one major problem encountered was initially only discovered after the 'evolution' phase had been in progress for several hundred generations, without any notable signs of progress[16].

After being completely unable to fathom what was going wrong (*every* method was checked repeatedly, and no major errors, logic or otherwise, were found) it was decided to make a copy of all the code, and to modify this copy to play Noughts and Crosses instead of Othello, in the hope that this simpler game would make the bug more obvious. Noughts and Crosses's game tree complexity is in the order of 10^5, as opposed to Othello's estimated 10^58. Its game tree can therefore be expanded totally with relative ease.

Converting the main classes to Noughts and Crosses was a reasonably straightforward process. A new evaluation function was created, called 'Perfect Evaluation' which relied on total expansion of the game tree, assigning higher values to games won after fewer moves, and lower values to games lost after fewer moves[17]. This was played against in single player mode — the expectation was that the computer player utilising the `PerfectEvaluation` function would always win or draw. In fact, the opposite was true.

Whilst playing the game, the heuristic value that the `PerfectEvaluation` assigned to each state was displayed on the screen. In many cases, these values appeared to be incorrect, for instance, no positions ever appeared to be

[16] For a description of measures of progress, see section 3.3.3.

[17] A winning configuration was assigned the value $1 + S_e$, and a losing configuration $-(1 + S_e)$, where S_e is the number of empty spaces. A draw was assigned the value zero.

assigned a negative value, even when moving in this position would have resulted in a loss for the computer. Tracing the logic through the program, it was discovered that the slightly convoluted method by which an evaluation function is informed from which player's perspective it should evaluate the board state was buggy, resulting in a win from either player being considered 'good'. This was fixed simply by the insertion of a couple of minus signs.

After this, the computer played reasonably well, blocking wins from the human player, up until the point where it had the opportunity to make a winning move itself, where in every case, it appeared to make every possible attempt to avoid winning. Whilst initially, this was quite gratifying, it rather spoiled the effectiveness of the evaluation function. This transpired to be a bug in the $\alpha\beta$ search tree expansion. When the method was written, it was assumed that the head of the tree would always be a 'Max' node, thus the non-recursive method (which calls the recursive method) always assigned the value of the maximum child to the head of the tree. Unfortunately, this is not how the game tree search algorithm was being used. Rather than creating one game tree (whose head *would* be a 'Max' node) the code creates as many game trees as there are legal moves — one for each possible move, where the head of each tree represents the state of the board after the move has been made. The maximum value of the head of each of these trees is then used to determine which move to make, implying that the head of each tree should be a 'Min' node. Since every `Node` object stores its status as 'Max' or 'Min' in an instance variable, it was trivial to amend the non-recursive $\alpha\beta$ method to actually check whether the head of the tree is 'Max' or 'Min', and to assign the maximum or minimum value accordingly.

Subsequently, the `PerfectEvaluation` did indeed perform perfectly. Each pair of corresponding Othello and Noughts and Crosses classes were compared with `diff` on the School of CS's Unix machines, and the above described logic bugs were resolved in the Othello classes. Once the neural network evolution process was set to work, results were reasonably quick to be seen.

4.5 Results

Figure 8 shows a graph displaying the evolutionary progress of the population of artificial neural networks, measured by the benchmarking technique described in section 3.3.3. This shows the number of games won from 1000 played between the best performing neural network from each generation, and

a random moving opponent. The neural network player improves markedly from approximately generation 80, reaching a plateau at around generation 300. Evolution was stopped just after the 800^{th} generation, as no significant evolutionary progress had been made for approximately 500 generations. The dip in the graph towards the end is probably just a local minimum, as occurred around generations 400 and 700. It should probably be noted that although the neural network player evolves to win significantly more games versus the random player, it does not appear to be any better at avoiding draws, which tend to make up 30–40 of the games each generation.

Whilst the graph does appear to show significant evolutionary progress, it should be pointed out that since the opponent was a random mover, it can be assumed that approximately half the moves made were *good*, and half *bad*. Unfortunately, no pre-developed expert Othello playing evaluation function was available to use as a benchmark. These results are perhaps then nothing to become *too* excited about. Sections 5.1 and 5.2 offer some thoughts on the further improvement of play.

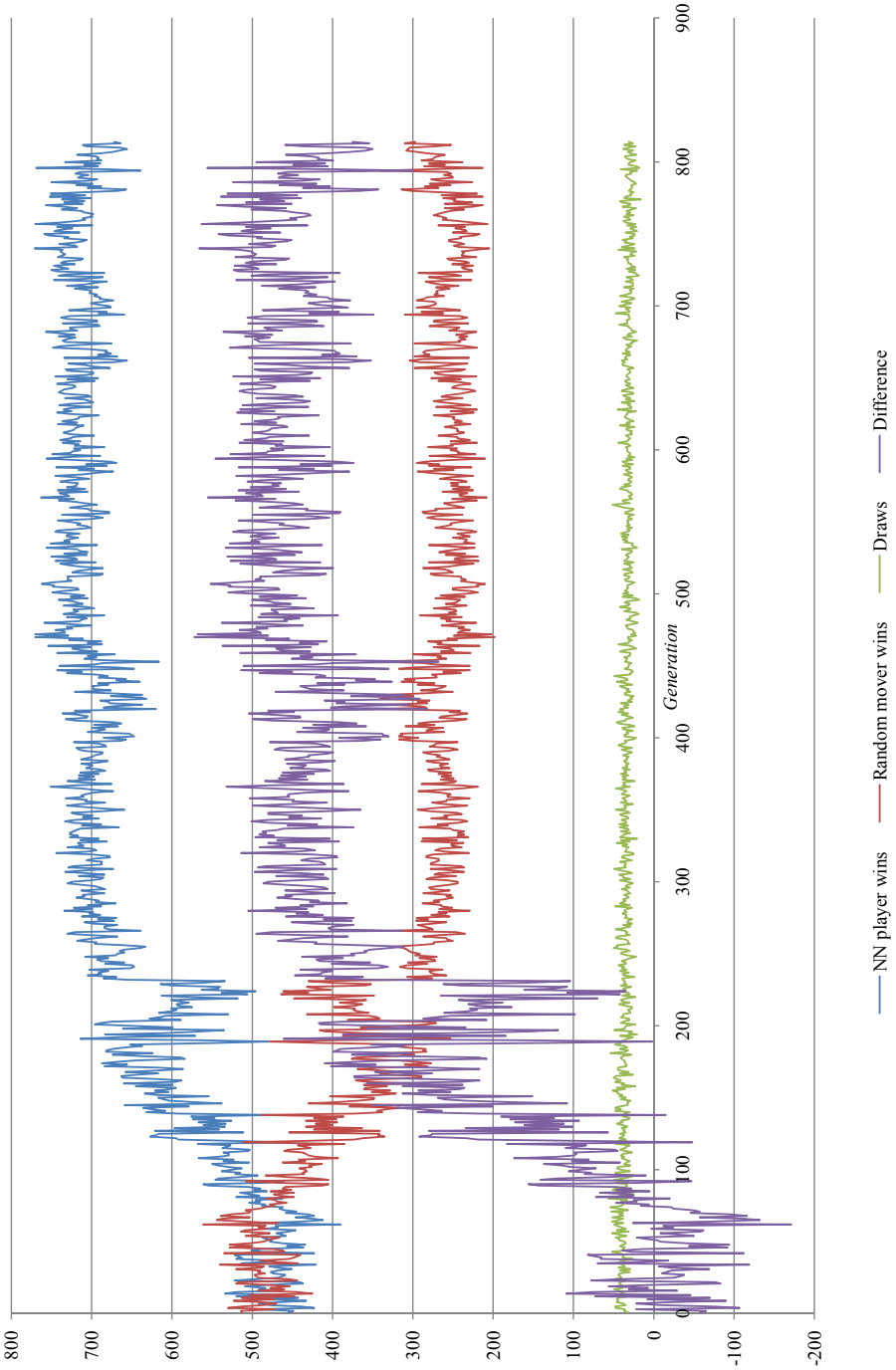

Figure 8: Graph of scores after 1000 neural network vs. random player matches, by generation

5 Evaluation

5.1 Conclusions

In the discussion section of [7], Moriarty and Miikkulainen state:

> Another question to explore in future research is whether networks can evolve significant play by playing each other. Co-evolving populations would preclude the need for a strong, searching opponent, which is typically the bottleneck in these simulations. Additionally, the networks should evolve more general game-playing strategies since the opponent's strategy is not constant. A difficult issue that must be addressed, however, is how to judge performance of one network relative to other networks in the population. Since the strength of the opponents will vary, a weak network may exhibit strong play simply because its opponents were subpar. Such fitness evaluation noise could be averaged out by playing each network against more opponents, however, this would greatly increase the time needed for evolution.

Just as Arthur Samuel in his 1959 paper ([9]) cited the high computational overhead of using artificial neural networks to evolve his checkers player, so in 1995 Moriarty and Miikkulainen state that playing their artificial neural networks against one another was considered too computationally expensive to be worth bothering with.

This project has shown that the method Moriarty and Miikkulainen considered and dismissed has been successful in evolving an Othello player. Clearly, evolution did occur to some extent, and judging by the way the graph in figure 8 levels off, this was to the optimal level achievable by the system configuration. In order to further evolve the neural network population, following in the path of Moriarty and Miikkulainen — evolving each neural network against a player afforded $\alpha\beta$ search capabilities — is considered as the logical next step to take.

5.2 Improvements and Possible Extensions

As stated above, the next obvious step to take in order to improve the neural network population's playing abilities is to evolve them against another player

which uses some form of game tree search. The evaluation function used by this player can simply be that of one of the neural networks evolved thus far. It would certainly be interesting to see just how far the Othello playing strategy can be evolved.

One feature of many programs is notably absent from this project — a graphical user interface. Whilst it would be nice to have the use of a GUI, it is not really a crucial element of the project, since games can be perfectly displayed and interacted with in the console in the form of an ASCII-based interface. Since the main focus of this project was to evolve an Othello playing evaluation function, and not to create an Othello game to be released to the masses, it was felt that the inclusion of a GUI would be given a low priority.

6 Bibliography

[1] Louis Victor Allis. Searching for solutions in games and artificial intelligence. Master's thesis, University of Limburg, 1994.

[2] Michael Buro. The evolution of strong othello programs. In *Entertainment Computing - Technology and Applications*, pages 81–88. Kluwer, 2003.

[3] Kumar Chellapilla and David B. Fogel. *Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program against Commercially Available Software*. Congress on Evolutionary Computation 2000 (CEC 2000), July 16-19, La Jolla Marriot Hotel, La Jolla, California, USA.

[4] Kumar Chellapilla and David B. Fogel. Evolution, neural networks, games, and intelligence. *Proceedings of the IEEE*, 87(9):1471–1496, 1999.

[5] Balázs Csanád Csáji. Approximation with artificial neural networks. Master's thesis, Faculty of Science, Eötvös Loránd University, Hungary, 2001.

[6] K. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36(1):1–26, 1988.

[7] David E. Moriarty and Risto Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(2):195–209, 1995.

[8] P. S. Rosenbloom. A world-championship-level othello program. *Artificial Intelligence*, 19(3):279–320.

[9] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, pages 211–229, July 1959.

[10] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2–3):273–290, 1992.

[11] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 50:189–226, 1996.

[12] Jonathan Schaeffer, Norman Treloar, Paul Lu, and Robert Lake. Man versus machine for the world checkers championship. *AI Magazine*, 14(2):28–35, 1993.