Computer Science Honours
Research Proposal

Augmenting Chess Evaluation Functions with Artificial Neural Networks

UNIVERSITY OF THE WITWATERSRAND
SCIENTIA ET LABORE
JOHANNESBURG

Steven James
475531

Supervised by: Prof Christian Omlin

19 July 2013

Abstract

Over the past 50 years, much work has been done in the field of computer chess to the extent that humans are no longer able to compete against the best chess programs. The large majority of this research is focused on expanding the game tree. Through a combination of improved hardware and search techniques, chess programs of today are able to analyse millions of positions per second. Despite the strength of these programs, many bemoan the fact that chess programs display no real intelligence — they are simply efficient searching machines. Furthermore, top programs do not possess the ability for self-improvement.

Artificial neural networks (ANNs) are one paradigm for creating self-improving agents. This paper proposes constructing a pseudo-intelligent chess program that implements an ANN. The program will be evolved using a genetic algorithm through a series of tournaments, with the final result being compared with a traditional, non-learning chess program.

Declaration

I, Steven James (student number: 475531), am a student registered for the the degree of BSc. Computer Science (Hons) in the academic year 2013.

I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that the work submitted for assessment for the above degree is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this is not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature: [illegible] Date: 19/07/13

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The notion of an automated chess-playing machine is even older than the advent of the computer itself. Standage [2002] recounts the story of a fake chess-playing automaton constructed during the 18th century, while Leonardo Torres y Quevedo developed a machine capable of playing a certain chess endgame as early as 1912 [Montfort 2005]. It is thus unsurprising that there exists a plethora of literature on the subject of computer chess. This can perhaps be attributed to the game itself. Not only is chess one of the most well-known board games in the world, but the complexity of the game strikes a fine balance. As Shannon [1950] claims, the problem of programming a computer to play chess is not too hard as to be intractable, nor too easy so as to be trivial.

Although programming a computer to play chess falls under the category of "artificial intelligence", some lament the fact that modern chess engines do not exhibit any form of intelligence. Fogel *et al.* [2004] observe that improvements in the strength of chess engines have been as a result of advances in hardware and software optimisations as opposed to advances in the field of artificial intelligence. It could be argued, however, that there is no need to implement such an approach — top chess programs have long since surpassed humans' abilities.

That is not to say that there have been no attempts at creating pseudo-intelligent chess programs — Fogel *et al.* [2004] is but one example of a chess engine that exhibits self-learning characteristics. The focus of self-learning chess engines is not in learning how to play a legal game of chess; rather, such an engine is given the rules of chess and must learn to play at a high skill level. This can be further distilled to a single action — evaluating a chess position and determining whether it is advantageous to the program. Within the framework of a chess engine, this action is the responsibility of the evaluation function.

The evaluation function is core to this research topic and the notion of a self-evolving program. Since there are no hard-and-fast rules for determining how good a chess position is, heuristics are required. Known tactical features are often incorporated into the evaluation function. These features are known as the parameters of the evaluation function. Weighting these parameters according to their importance is often also done by hand, although some have used genetic algorithms to optimise these weights, with varying degrees of success [Aksenov 2004] [David-Tabibi *et al.* 2011].

Another approach involves using an artificial neural network (ANN) to evaluate a position. The ANN need not replace the entire evaluation function — such a choice is unlikely to be successful (refer to Section 4.4 for an example). Rather, the ANN is used in conjunction with certain parameters, as put forth by Fogel *et al.* [2004] — in actual fact, their evaluation function implemented three ANNs.

Although the abovementioned approaches are currently inferior to the refined evaluation functions of today, such avenues should not be ignored. As previously implied, an optimal evaluation function is as yet unknown. The use of ANNs in evaluation functions may therefore be able to capture some intangible quality of a position that the defined heuristics cannot. Furthermore, a chess engine that is able to improve its play without human intervention is a desirable goal.

The focus of this research topic revolves around the aforementioned approaches. In particular, an evaluation function similar to the one used by Fogel *et al.* [2004] is proposed. By using a genetic algorithm to evolve the evaluation function (and notably its neural network), it can be determined whether genetic algorithms and neural networks can indeed be used to evaluate chess positions with any success. Such an approach also allows for comparison with traditional, deterministic chess engines. This can easily be achieved by pitting the evolved program against an engine that uses the same evaluation function, *sans* neural network. After a significant number of games, it can be concluded whether there is any significant difference between the two. Comparison with a commercial chess engine can be done in a similar fashion. Note that the focus of this research is not in designing a chess program capable of competing with a commercial engine — this is just not feasible given the timeframe. The focus here lies solely on the evaluation function.

The results of the above tests can be somewhat predicted. Evolving an ANN using a genetic algorithm is a viable means of creating an evaluation function, as it has already been achieved by Fogel *et al.* [2004]. Moreover, it is likely that an evaluation function with an ANN is superior to the same function without one, since the extra information captured by the ANN is probably beneficial to the evaluation function's accuracy. However, it is unlikely to better a commercial program's evaluation function, since it has many parameters, each of which has been refined and improved over a period of time. Naturally, these claims are tentative and remain to be tested.

Having briefly introduced the problem area, the remainder of this proposal is structured as follows. Chapter 2 reviews the basic concepts of computer chess, beginning with an explanation of the rules of the game. A brief history of the field of computer chess is then provided, as is a basic framework for a chess program (Section 2.4). Also discussed in this chapter are artificial neural networks and genetic algorithms. Section 2.5 explains the idea behind these two concepts and describes attempts to apply them to board games — chess in particular (Section 2.5.4).

Chapter 3 focuses on the research methods to be used. The research hypothesis is formally presented in Section 3.2, followed by an outline of the three distinct phases necessary to accept or reject the hypothesis (Section 3.3). These comprise an implementation, training and testing phase. The chapter concludes by providing motivation for the above phases and their relevance to the hypothesis.

Chapter 4 provides an outline for a research plan. The chapter emphasises important deliverables and provides a timetable for completing these deliverables (Sections 4.2 and 4.3). The chapter concludes by noting potential issues that may arise over the course of conducting the research and provides solutions to these problems (Section 4.4).

Lastly, Chapter 5 provides concluding remarks and a summary of the entire proposal.

Chapter 2

Background and Related Work

2.1 Introduction

The previous chapter outlined some of the deficiencies in modern chess programs. This chapter investigates the background necessary to construct a self-improving chess engine. For the sake of completeness, Section 2.2 provides an explanation of the rules and basic mechanics of chess. Readers familiar with the game of chess may wish to omit this section. Section 2.3 provides some background and highlights important ideas in the field of computer chess. Certain of these techniques are further expounded upon in Section 2.4. Of great importance to this research topic is the notion of an evaluation function, which is detailed in Section 2.4.2. Finally, Section 2.5 provides an overview of genetic algorithms (GAs) and artificial neural networks (ANNs). This includes an explanation of both GAs and ANNs, as well as their applications in the game of checkers (Section 2.5.3) and, more importantly, chess (Section 2.5.4).

2.2 Basic Chess Concepts

The following section explains the basic rules of chess as well as terminology used throughout this document. The rules are adapted from the official handbook of the *Fédération Internationale des Échecs* [FIDE 2008] — the international governing body of chess.

2.2.1 Introduction

Chess is a turn-based game played on an 8x8 board with squares of alternating colours. Two competitors contest the game — the player who controls the light-coloured pieces is termed white, with the player controlling the dark-coloured pieces named black. The game begins from the position illustrated in Figure 2.1, with white always the first to play.

The chess board is divided into horizontal rows and vertical columns. Horizontal rows are termed *ranks* and are labelled **1 − 8**, while vertical columns are known as *files* and labelled **a − h**. An individual square can be uniquely identified by specifying its rank and file. In Figure 2.1, for instance, the bottom-left corner is **a1**, while the top-right corner is **h8**.

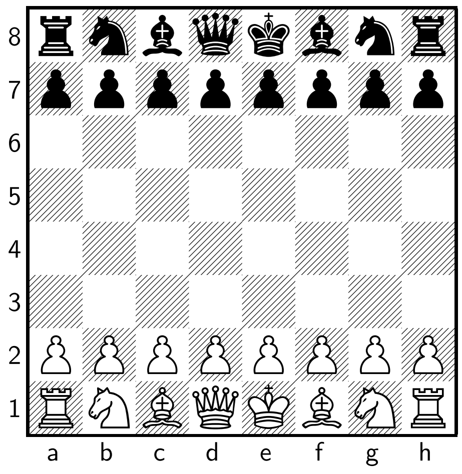


Figure 2.1: Starting position of a chess game

As evident in the above diagram, both white and black begin with 16 pieces. These pieces make up each player's *material* and are listed below:

White	**Pieces**	**Black**
♔	1 King	♚
♕	1 Queen	♛
♖	2 Rooks	♜
♗	2 Bishops	♝
♘	2 Knights	♞
♙	8 Pawns	♟

Table 2.1: List of pieces with which both sides begin

2.2.2 Piece Movement

The rules that govern the manner in which these pieces move are described below. Note that no piece may move to a square already occupied by a piece of the same colour. If the square to which a piece moves is occupied by a piece of a different colour, the latter piece is replaced by the former. This is known as a capture.

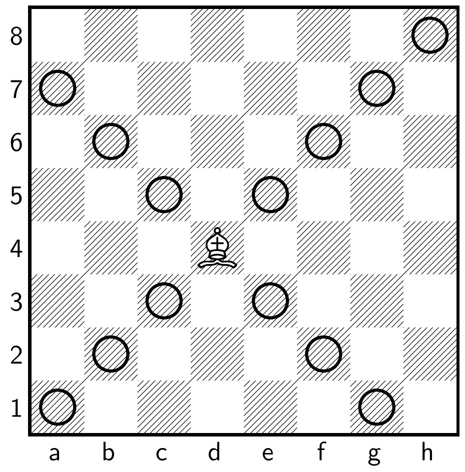
Sliding Pieces

The rook, bishop and queen together constitute what are informally known as sliding pieces. They may move any number of squares along a horizontal, file or rank (depending on the piece) until the edge of the board or an obstructing piece is reached. Sliding pieces may not move over any obstructing piece, regardless of colour.

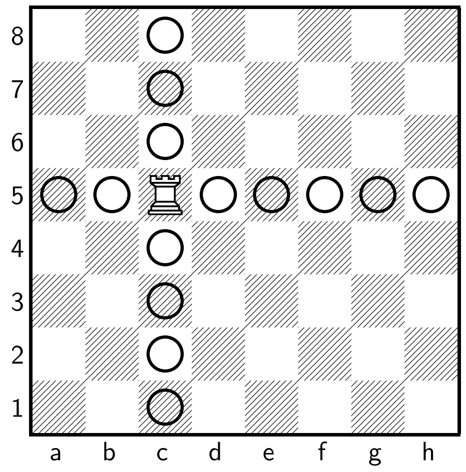
Bishop The bishop can move any number of squares along a diagonal on which it stands.

Rook The rook can move any number of squares along the file or rank on which it stands.

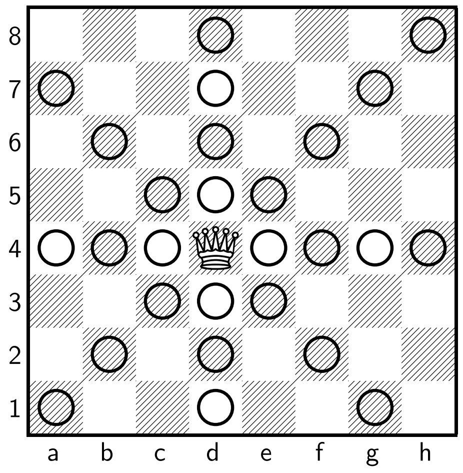
Queen The queen can move to any square along a diagonal, rank or file on which it stands.



(a) Movement of a bishop



(b) Movement of a rook



(c) Movement of a queen

Figure 2.2: Movement of sliding pieces

Non-Sliding Pieces

Unlike the above, the remaining pieces may only move a fixed number of squares.

Knight The knight moves to one of the nearest squares that does not lie on the same file, rank or diagonal as itself (Figure 2.3)

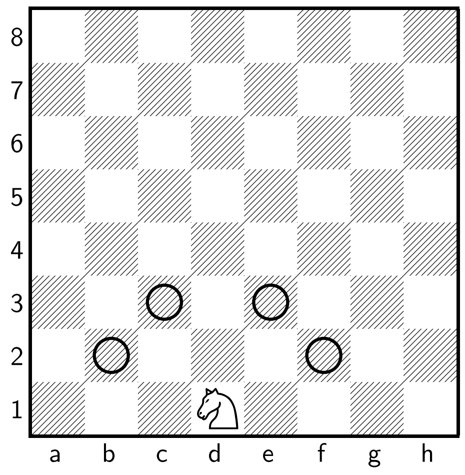
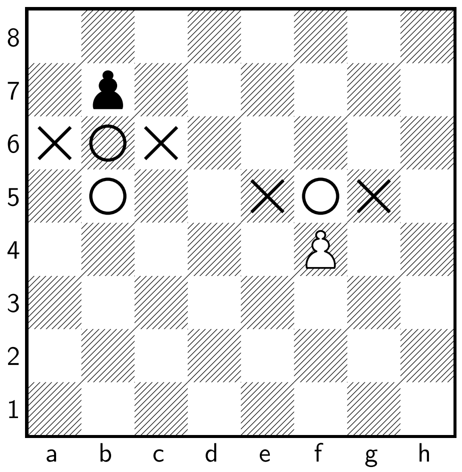


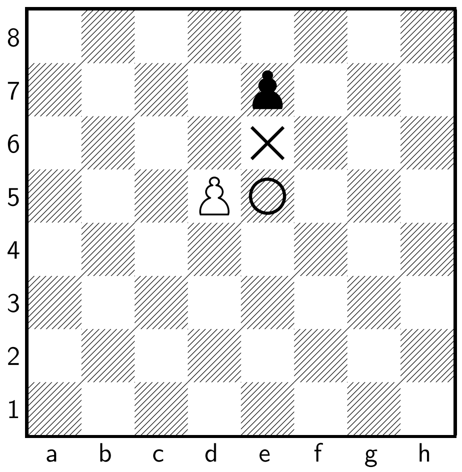
Figure 2.3: Movement of a knight

Pawn The movement of pawns (Figure 2.4) depends on the situation of the game:

- A pawn may advance to the square directly in front of it on the same file, provided the square is unoccupied.
- On its initial move, a pawn may proceed as above or choose to advance two squares along its file, provided both squares are empty (Figure 2.4a).
- A pawn may capture an opponent's piece that lies diagonally in front of it on adjacent files (Figure 2.4a).
- If a pawn attacks the square crossed by an opponent's pawn when advancing two squares, it may capture the opposing pawn as though it only advanced one square, provided this is done the very next move. This is known as an *en passant* capture (Figure 2.4b).
- If a pawn advances to the rank furthest from its starting position (i.e. a white pawn reaching the eighth rank or a black pawn reaching the first rank), it is exchanged for either another queen, rook, bishop or knight of the same colour. This is known as promotion.



(a) Crosses indicate the square an opposing piece must lie on for a pawn to capture it. Circles indicate movement of the pawn. As it has not yet moved, the black pawn may advance by one or two squares.



(b) If the black pawn advances to `e5`, the white pawn may capture it *en passant*. The white pawn would be placed on `e6` and the black pawn removed.

Figure 2.4: Movement of pawns for black and white

King A king may move to any adjacent square not attacked by an opposing piece (Figure 2.5). The king may also take part in a move known as castling (Figure 2.6). This is a single move involving the king and rook along the player's first rank. The king moves two squares towards the rook and the rook is moved to the square passed over by the king. A player forfeits the right to castle if the king has previously been moved. A player may also not castle with a rook that has been moved (but may castle with his other rook, provided it, too, has not moved). Castling is temporarily prevented if either the king's square, the square it is to pass over or the final square it will occupy is attacked by an opposing piece, or if there is any piece between the king and castling rook.

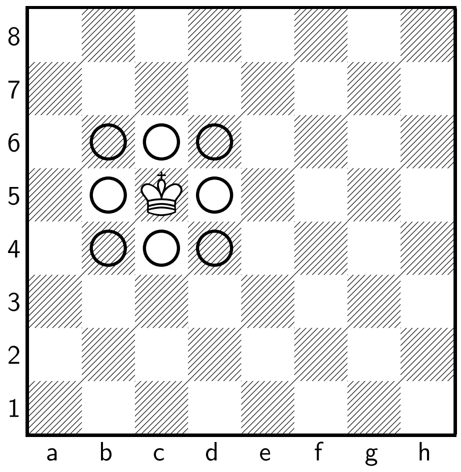
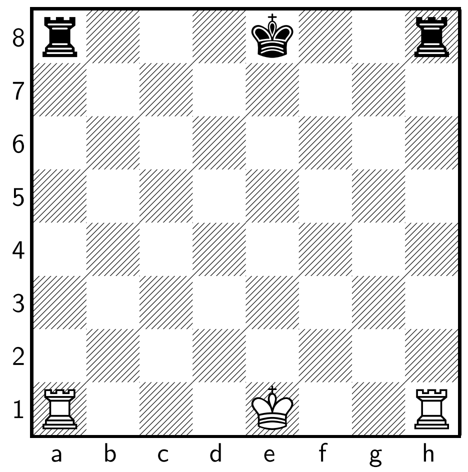
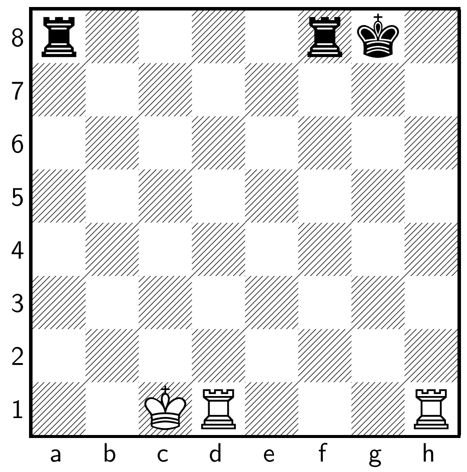


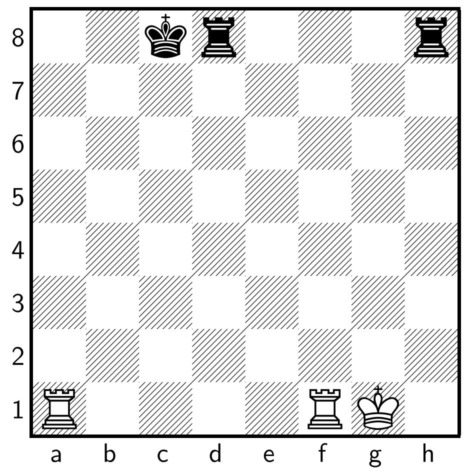
Figure 2.5: Movement of a king



(a) Before castling



(b) White castles queenside and black castles kingside



(c) White castles kingside and black castles queenside

Figure 2.6: Examples of castling

2.2.3 Game Completion

A king is said to be *in check* if the square it resides on is attacked by an opposing piece. Leaving one's own king in check or moving a piece that exposes the king to check is illegal. If a player whose king is in check has no legal move, he is said to be *checkmated* and loses the game. However, a player with no legal move whose king is not in check is said to be stalemated and the game is drawn. The game is also drawn when neither of the players is able to checkmate his opponent (*draw by insufficient material*), the identical position occurs three times during the game (*threefold repetition*) or no pawn has moved nor capture made in the last 50 moves (*fifty-move rule*).

2.3 Foundations of Computer Chess

The field of computer chess was strongly influenced by Claude Shannon's landmark paper. Shannon [1950] begins by defining the information needed to represent a position in chess. Important limitations of using a computer to play chess are explained. For example, it is completely infeasible to determine what the end result of any given position would be, owing to the exponential expansion of the game tree.

The author also presents the concept of the evaluation function — this is described thoroughly in Section 2.4.2. Important concepts in the field of computer chess are put forth, including quiescent positions (positions in which no immediate captures or moves that cause check can be made) and the minimax search algorithm (evaluating every reachable position up to a certain depth and selecting the optimal one). Shannon terms programs based on such brute-force strategies `type A`, with the vast majority of modern chess engines adopting this approach.

A weakness of a chess engine — that it is unable to learn from its mistakes or improve (without the code itself being improved) — is highlighted. This is followed by the idea for a self-improving program in which the evaluation function is altered (depending on the results it obtains in matches). This is similar to the approach adopted by Aksenov [2004] and David-Tabibi *et al.* [2011] more than 50 years later.

2.4 Anatomy of a Chess Program

As mentioned above, a chess engine can be divided into two major components: a search algorithm and an evaluation function. For example, the computer Deep Blue made use of a heavily optimised search algorithm and complex evaluation function to defeat a chess world champion [Campbell *et al.* 2002]. This section will first briefly examine the search algorithm before providing a detailed account of the evaluation function.

2.4.1 Searching the Game Tree

Although many optimisations are now applied to this component of the chess engine, the basic idea remains the same. In essence, the program implements a minimax algorithm — given a position, this algorithm computes all reachable positions. It does this by playing each possible move and, for each of these moves, every possible reply, and so on. This search must be limited to some depth, since it is impossible to perform a full search from the starting position to the end of a game (Shannon [1950] conservatively estimates that there are 10^{120} reachable positions from the beginning of a chess game).

Having enumerated all reachable positions, the algorithm then chooses the move that, given optimal play by both sides, results in the best position for the current player [Russell and Norvig 2010]. The minimax algorithm (or any other search algorithm used) thus requires some way of deciding how good a position is. This is the role of the evaluation function.

2.4.2 Evaluation Function

The majority of game-playing programs (including chess and checkers) makes use of a static heuristic known as an evaluation function. In essence, the evaluation function accepts a game position and returns an estimate of the value of that position. Most chess engines use the weighted sum of a number of domain-specific factors as their evaluation functions. For example, an evaluation function $f(Pos)$ may have the following form:

$$f(Pos) = w_1(Q - Q') + w_2(R - R') + w_2(B - B') + w_3(N - N') + w_4(P - P') + w_5(M - M') + \dots$$

where w_i represent weights, Q, R, B, N and P the number of white queens, rooks, bishops, knights and pawns and M the mobility of white. Primed letters represent the same aspects, but for black.

Note also that since chess is a zero-sum game (i.e. if white is winning by a score of 2, then black is losing by that same score), the same evaluation function can be used for both black and white.

There are numerous parameters that could be considered when devising an evaluation function — the professional chess engine FALCON contains well over 100 such parameters [David-Tabibi *et al.* 2011], whereas DEEP BLUE used more than 8000 [Campbell *et al.* 2002]. All of these parameters are derived from humans' understanding of chess.

The difficulties in formulating a near-optimal evaluation function are many. The first lies in deciding on which tactical aspects of chess will be included — using all conceivable parameters would yield extremely accurate results, but evaluating a position would take more time. This would increase the time taken to search the game tree, resulting in a shallower search depth being reached. The program designer must thus weigh the advantages of an accurate evaluation function with a shallow depth against a less accurate function that allows the engine to reach a deeper depth. Another issue is in the determination of the weights. Not only must the individual weights be given suitable values, but the parameters attached to these weights are likely to vary with time. The safety of the king, for example, is important early in the game, but less so as the endgame is reached. Despite this, weights are often treated as constants.

2.5 Artificial Neural Networks and Genetic Algorithms

Having discussed the basics of creating a chess engine and, in particular, the evaluation function used, focus now shifts to concepts that can be used for "learning". This section provides a brief overview of two of these ideas — ANNs and GAs — both of which are analogous to natural phenomena. After a brief explanation, their applications to the game of chess are discussed.

2.5.1 Artificial Neural Networks

An ANN is a mathematical model inspired by the human brain. It consists of individual units or neurons linked together to form a network. Before examining the manner in which these units are connected to one another, the structure of the units themselves must be explained. Figure 2.7 illustrates a simple neuron.

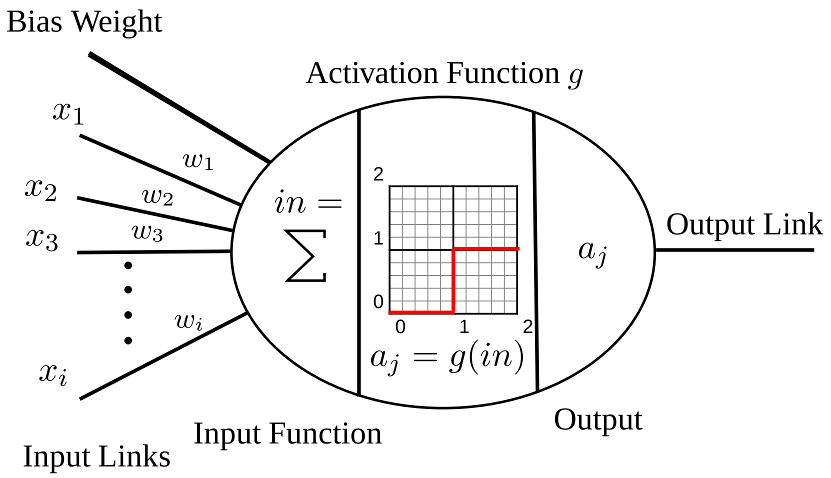


Figure 2.7: A simple illustration of a neuron with a threshold activation function

A neuron is connected to other neurons through links. Each of the input links to a neuron is weighted. In addition, each unit also has a fixed input with associated weight — this weight is known as the bias weight [Russell and Norvig 2010]. The neuron functions by computing the sum of the inputs of its links multiplied by their respective weights. In the above diagram, this sum would be given by $bias + w_1x_1 + w_2x_2 + \cdots + w_ix_i$. This value now serves as input to the neuron's activation function — this defines the output of a unit given an input. The activation function is chosen by the designer of the neural network. It may be a binary step function (as in the above neuron), or some other function (usually sigmoidal) may be chosen [Cheung and Cannons 2002]. The output of the activation function is then propagated along the neuron's output link.

Neurons are connected together to form ANNs. The number of neurons in the network and the network's topology is context-dependent. Networks are often arranged in layers, with each neuron receiving input only from neurons in the previous layer [Russell and Norvig 2010]. Layers can be divided into input layers, output layers and those in between (hidden layers)[Cheung and Cannons 2002]. Neural networks can be classified as either feedforward or recurrent. In a feedforward network, links between the neurons do not form cycles, whereas recurrent networks propagate its output back into its own inputs [Stergiou and Siganos nd]. An example of a feedforward neural network with one hidden layer is shown in Figure 2.8.

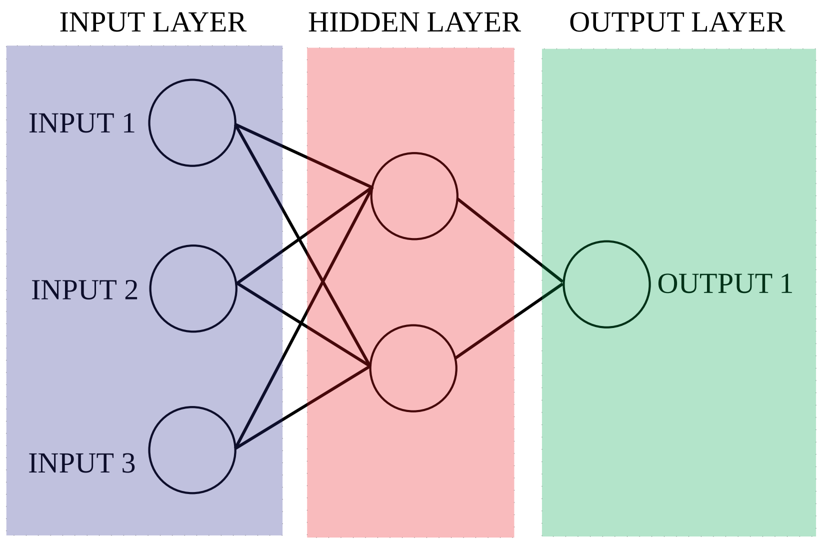


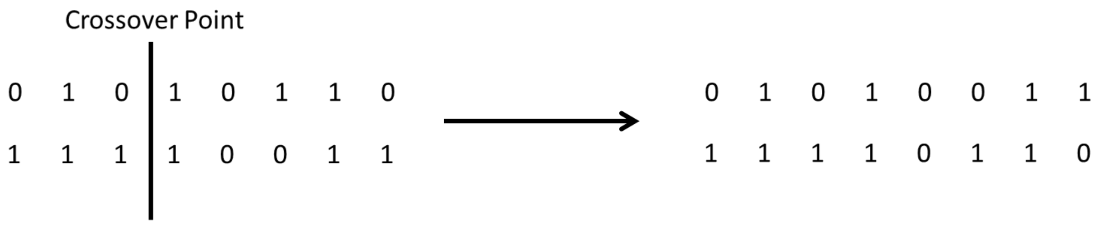
Figure 2.8: Neurons arranged in a simple feedforward network

Of great importance are the weights of the links between neurons. ANNs can be trained by altering these weights to approximate the required output. Techniques such as backpropagation can be used to train these networks when the desired output is known [Cheung and Cannons 2002]. However, since the correct output of the chess evaluation function is not known beforehand, this technique cannot be adopted. Instead, the weights of the ANN will be evolved using a genetic algorithm.

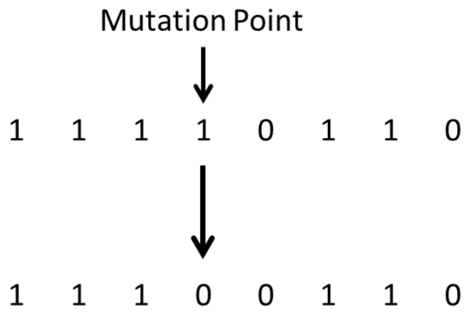
2.5.2 Genetic Algorithms

While artificial neural networks are designed to imitate the human brain, genetic algorithms are based on the theory of natural evolution. GAs begin with a set of randomly-generated individual states known as the population. Each individual consists of the parameters that need to be evolved and represent a candidate solution to the problem at hand. Decomposing a possible solution into a set of parameters that can be manipulated by the algorithm is known as coding [Beasley *et al.* 1993].

The GA evaluates every member of the population using a fitness function — a measure of how good the state is. Pairs of individuals are then selected at random, with those having higher evaluations being favoured. These pairs are called parents and are used to generate individuals (children) that will make up the new, smaller population. Genetic operators are applied to parent states to produce children. A common method of generating a child state is to splice together the parameters of its parents. This is known as crossover. Another step, mutation, may also be applied, with some parameters of the new individual being slightly altered at random [Russell and Norvig 2010]. The process then begins again using the new population, with the final individual representing the evolved agent. An example of crossover and mutation is presented by Figure 2.9, while the basic structure of the entire algorithm is illustrated by Figure 2.10.



(a) An example of crossover. Two parents are recombined to form two new children



(b) An example of mutation. One of the children's parameters is chosen to be mutated.

Figure 2.9: Illustration of genetic operators (crossover and mutation)

```
1:  function GENETIC-ALGORITHM($population$)        ▷ $population$ is a set of random individuals
2:      repeat
3:          $newPopulation \leftarrow \emptyset$
4:          for $i = 1 \rightarrow \frac{\text{LENGTH}(population)}{2}$ do
5:              $x \leftarrow$ RANDOM-SELECTION($population$)              ▷ selecting parents
6:              $y \leftarrow$ RANDOM-SELECTION($population$)    ▷ biased towards higher fitness scores
7:              $u, v \leftarrow$ COMBINE($x, y$)                 ▷ crossover producing 2 children
8:              if true with small random probability then                     ▷ mutation
9:                  $u \leftarrow$ MUTATE($u$)
10:             end if
11:             if true with small random probability then
12:                 $v \leftarrow$ MUTATE($v$)
13:             end if
14:             add $u, v$ to $newPopulation$
15:         end for
16:         $population \leftarrow newPopulation$
17:     until $population$ has converged or an individual is fit enough
18: end function
```

Figure 2.10: Pseudocode describing a genetic algorithm that uses both crossover and mutation. Here each generation is the same size as the previous.

2.5.3 Neural Networks and Genetic Algorithms in Checkers

Although both checkers and chess programs use evaluation functions of the form given in Section 2.4.2, the less complex nature of checkers (there are fewer types of pieces, for one) results in a simpler evaluation function. It thus represents a good starting point for investigating the application of ANNs and GAs in evaluating positions.

Chellapilla and Fogel [2001] propose evaluating a position by considering the material of each player, the position of these pieces and the output of a neural network. A genetic algorithm is applied to optimise not only the neural network, but also the value assigned to a king (which is more valuable than a regular piece). The genetic algorithm consists of fifteen randomly initialised neural networks. Players make use of these neural networks (described as strategies) to compete against one another in a tournament, with the best performers being selected as parents for the next generation. The children of these individuals are generated by varying the weights and biases of the neural networks, as well as the value of the king.

At the conclusion of the genetic algorithm, the best neural network was used to play against human opponents on a web site. After more than 100 games, the neural network obtained an expert rating, higher than 99% of all the players on the site.

Another experiment was conducted, with the neural network competing against a program that evaluated a board using piece differentials only. While the neural network was not able to reach the same search depths as its opponent in the given time, it still managed to win comprehensively. This indicates that the neural network does indeed contribute to the strength of the program.

2.5.4 Evolving Chess Programs

Having briefly investigated the use of ANNs and GAs in checkers, the game of chess can now be examined. Before describing a similar experiment to the one conducted in Chellapilla and Fogel [2001], it should be noted intuitively that using a single ANN to evaluate a chess position would be less effective than using one to evaluate a checkers position. This is owing to the increased complexity of chess and the fact that many pieces affect squares outside their immediate surroundings.

To counter the above issue, Fogel *et al.* [2004] proposed using three criteria to evaluate a position on a chessboard: the material of each player, the values derived from having specific pieces on certain squares (positional value tables — PVTs) and the output of **three** neural networks. These networks are each associated with specific areas on the chessboard. One corresponds to the front two ranks, one to the back two ranks and the third to the middle sixteen squares. These areas were chosen for their importance when playing chess.

As in the previous section, a genetic algorithm that seeks to determine the best values for each piece, the PVTs and weights and biases of the nodes in each neural network is presented. Multiple games were played amongst the initial population, with the best-performing individuals going on to become parents for the next generation. The best individual after the genetic algorithm had concluded was used for further tests. It consistently defeated a non-evolved player in ten trials, as well as defeating the chess engine Pocket Fritz 2.0, which would classify the evolved player as a high-level master.

A different approach is adopted by Hauptman and Sipper [2005]. The authors eschew the conventional evaluation function (and all its associated problems) in favour of strategies represented as tree expressions. They choose to limit their focus to endgames consisting of kings, queens and rooks, with their ultimate aim being the development of more general strategies.

The experiment carried out by Hauptman and Sipper [2005] is described thus: an individual plays by performing a 1-ply search and selecting the move that results in the most favourable position according to its evaluation function. Individuals are then evolved by playing a fixed number of games amongst one another. Those that manage to checkmate their opponents receive more points than those that only manage to achieve material advantage or a draw. The best-performing individuals are selected as parents for the next generation.

The evolved program was then pitted against a human-defined strategy and the chess engine Crafty. It managed a win against the human-defined strategy and had near-draw scores against Crafty — particularly impressive when its small lookahead is taken into account.

2.5.5 Optimising Evaluation Functions

Arguably one of the most effective techniques for training chess engines involves using a GA to optimise an evaluation function — recall that this method was suggested by Shannon [1950]. This involves choosing parameters to be included in the evaluation function, and then allowing a GA to evolve the weights associated with those parameters. Both Aksenov [2004] and David-Tabibi *et al.* [2011] investigate this technique.

Aksenov [2004] selects a subset of twenty-six parameters to form the evaluation function to be used in the paper's experiment. Justification for choosing each parameter is provided, as is an explanation of each one. These parameters, initialised to random numbers within certain ranges, form the population of the genetic algorithm. These ranges are based on human estimates of the importance of the individual parameters. Crossover and mutation are selected as the genetic operators, with the fitness of each individual calculated by means of a round-robin tournament.

The outcome of the experiment produces mixed results: certain parameters converge to expected values, while others do not. The paper concludes by providing possible explanations for the above problem. These explanations include the aforementioned problem of certain parameters varying in importance over time. This problem is exacerbated by the fact that dividing a chess game into phases (opening, middlegame and endgame) is non-trivial; thus allowing the evaluation function's weights to change over time is difficult to implement.

A similar experiment is presented by David-Tabibi *et al.* [2011]. However, its authors propose using an expert chess engine, as opposed to the round-robin used by Aksenov [2004], to evolve a set of evaluation parameters. The method used to evaluate the fitness of an individual is as follows: both the expert program and the individual analyse a list of random positions, with the fitness of the individual determined by how close the differences in their evaluations of the positions are. Thus the best individual will be the one that produces very similar evaluations to that of the expert. The expert program (Falcon) is then presented. Falcon differs only in that its evaluation function contains more than one hundred parameters, while the individuals in the genetic algorithm contain less than forty.

The best evolved individual (Evol*) was then tested against the expert, with Evol* winning 54% of all games against Falcon. This is attributed to the fact that the fewer parameters in the evaluation of Evol* allow it to apply its evaluation function faster than Falcon, which results in deeper searches. Evol* was also tested against other expert engines, defeating Hiarcs and being on a par with Fritz 4.2. Evol* made an appearance at the 2008 World Computer Chess Championship, placing a respectable 2nd in the speed event, and 6th in the normal event. The paper concludes by observing the success of the evolved program, especially considering the organisms were evolved within minutes (as opposed to round-robin style experiments which could take hours or days).

2.6 Conclusion

The topic of the proposed research focuses on evolving a chess evaluation function. This chapter therefore attempted to provide background on the subjects necessary to investigate this idea. Starting with an explanation of the game of chess, the chapter then detailed the approach to creating a chess engine. This included arguably the most influential paper in the field of computer chess — Shannon [1950]. A basic description of a chess engine — consisting of a search algorithm and evaluation function — was then provided.

Key to this research topic, the mathematical form of a conventional evaluation function was outlined, as were some of the problems associated with it. A brief explanation of genetic algorithms and neural networks followed. This explanation also detailed the way in which ANNs can be trained and how GAs are able to evolve individuals. The simpler game of checkers was first introduced as a starting point for modifying the evaluation function using ANNs and GAs.

Different experiments applied to the game of chess were then detailed. Some of these implemented an evaluation function using an ANN, whilst some omitted the ANN, opting instead for large sets of evaluation parameters. However, none of these experiments combined an ANN with a meaningful subset of evaluation parameters. The most impressive of these experiments was that which used an expert chess engine to guide the evolution of an engine using a conventional evaluation function (the evolved engine managed to defeat the original expert). However, experiments using ANNs also performed well and therefore cannot be discounted.

With the exception of David-Tabibi *et al.* [2011], the experiments described in this chapter indicated that using GAs to evolve an evaluation function resulted in weaker play when compared with expert chess engines. However, all unanimously stated that, despite this, the evolved program was better able to adapt than classical chess engines whose evaluation functions were handwritten. Furthermore, many believe that genetic programming is a viable means to machine intelligence in general.

Chapter 3

Research Methodology

3.1 Introduction

The previous chapter presented GAs and ANNs and their applications to the game of chess. Having investigated this, the main goal of the proposed research can now be stated. This chapter formally states the objectives of this research and details the research methodology to be followed. It is laid out thus: Section 3.2 presents the research hypotheses and motivates the choice of these hypotheses. Section 3.3 then describes the steps to be taken in order to accept or reject these hypotheses, followed by a short summary of the salient points of this chapter (Section 3.4).

3.2 Research Hypothesis

As discussed in the previous chapter, ANNs can and have been used as part of a chess engine's evaluation function. The purpose of this research is to examine the effect of adding an ANN to a simple evaluation function. The term "simple" here refers to a function with only few parameters. An example of such a function might only consider material and piece mobility. Contrast this with an expert engine that may use over 100 parameters. The performance of an evaluation function augmented with an ANN can thus be measured against both a simple and expert evaluation function. This statement leads to the following two hypotheses:

Research Hypothesis 1: Augmenting a simple chess evaluation function with an evolved neural network results in a more accurate function and hence improved play.

Research Hypothesis 2: An expert's evaluation function is significantly better than a simple evaluation function augmented with a neural network.

3.3 Methodology

In order to test the above hypotheses, an experiment must be conducted. This section details the design of such an experiment.

3.3.1 Phase 1: Implementation

In order to test the application of an ANN to a chess evaluation function, a chess engine must naturally first be created. The chess engine's design will follow the basic framework of a chess program presented in Section 2.4. A more detailed account of its construction is provided in Section 4.2.1. This phase will also involve the programming of an artificial neural network to be used in the engine's evaluation function.

The result of completing this phase is a fully-functional chess engine whose evaluation function has been augmented with an ANN.

3.3.2 Phase 2: Training

Having created the necessary chess program in phase 1, the focus now shifts towards evolving the weights in the evaluation function. This is done, as mentioned previously, using a genetic algorithm. One part of the GA is the fitness function which, in this instance, takes the form of a round-robin tournament. This phase constitutes two discrete sections. The first is concerned with the creation of the genetic algorithm and a mechanism for allowing engines to compete in a round-robin tournament. The second part involves actually evolving the engines using the genetic algorithm.

The final product of this phase is a chess engine whose evaluation function and neural network's weights have been optimised by the genetic algorithm.

3.3.3 Phase 3: Testing

The evolved chess engine can now be used to either accept or reject the research hypotheses as follows. First, the evolved engine is pitted against the same engine whose evaluation function does not use an ANN. Having played a significant number of games, it can be determined whether there is any significant difference between the two (by means of a non-parametric sign test). The first hypothesis can be accepted if it is found that the engine using an ANN is significantly better than its opponent. To further support this conclusion, both will be used to play against human competitors. The rating assigned to each will further indicate the difference (if any) between the two.

The second hypothesis can be tested in a similar manner. In this case, however, the hypothesis is rejected if there is no significant difference between the evolved and expert engine (or if the evolved engine is significantly better than the expert, which is unlikely). Since the expert engine is likely to be able to search to a deeper depth than the evolved engine, and since the focus of this research is the evaluation function, the expert engine's search depth will be limited. This means that the difference in skill levels between the two is a result of their respective evaluation functions only.

3.4 Conclusion

This chapter presented the methodology of the proposed research. The research hypotheses were formally stated, as were the steps necessary to accept or reject the hypotheses. The manner in which these steps lead to the acceptance or rejection of the hypotheses was also provided. The next chapter provides a more concrete plan for implementing the above experiment.

Chapter 4

Research Plan

4.1 Introduction

The previous chapter introduced the hypotheses and the research methodology to be adopted. This chapter presents the manner in which the methodology will be carried out in practice. Aside from creating the report, the work to be done can be divided into three distinct phases, as previously mentioned. Section 4.2 provides a detailed account of the tasks required for each phase. The remainder of the chapter consists of a time schedule for these activities (Section 4.3) and a discussion of any potential issues that may arise over the course of the project, as well as possible solutions (Section 4.4).

4.2 Deliverables

4.2.1 Phase 1: Implementation

Programming a chess engine is no trivial undertaking. At the very least, the tasks that need to be completed to ensure a full-functional chess engine are:

- **Board Representation**
  A data structure is required to represent the board's current position, as well as any auxilliary data (side to move, castling rights, etc.).
- **Move Generation**
  Given a position and the side to move, the program must be able to compute all possible legal moves that can be played. Test suites exist that can verify a move generation algorithm.
- **Game Tree Search Algorithm**
  This subtask can be achieved by implementing the minimax search algorithm. This allows the engine to construct the game tree from the current position and select the best move to make.
- **Evaluation Function**
  The evaluation function accepts a position and outputs a score based on aspects of the position

(see Section 2.4.2). Since the evaluation function also takes into account the output of a neural network, the construction of the neural network falls under this activity.

- **Optimisations**
  There exists well-known optimisations (not mentioned in this document) that allow the search algorithm to search a position to a deeper depth in the same amount of time. Certain of these optimisations should be implemented to speed up the engine's play and, ultimately, the amount of time needed to complete the genetic algorithm.

The chess engine will be written in the C++ programming language.

4.2.2 Phase 2: Training

Evolving the chess engine requires the construction of a framework that allows the engines to play tournaments against one another. This phase consists of the following:

- **Tournament Framework**
  This task consists of code that allows a number of chess engines to compete against one another in a round-robin tournament. The results of the tournament must be saved in order to be used by the genetic algorithm.
- **Genetic Algorithm**
  The genetic algorithm that will modify the weights of the evaluation function and neural network must be implemented. Using the results from the tournament, the algorithm will select the parents for the next generation and apply crossover and mutation to produce their children.

The genetic algorithm and tournament framework will also be written in C++.

4.2.3 Phase 3: Testing

Having trained the chess engine in the above phase, it now needs to be tested. The engine needs to compete against a similar engine that does not implement a neural network, as well as a commercial engine. This can easily be achieved by allowing the three engines to compete in a tournament against one another (a function provided by many chess GUIs). After a statistically significant number of games, the results can be used to determine if there is any significant difference between the three.

The engine will also be used to compete against players on a chess website to achieve an Elo rating, which provides an indication of its strength.

4.3 Time Plan

The following time plan is based on the abovementioned phases. Of interest are the five weeks set aside for training. This provides leeway should problems occur that require training to be restarted.

Hours are not allocated for this phase, since no human interaction is necessary. Note, too, that the grouping of tasks closely follows the three phases mentioned in the previous section.

Week	Activity	Allocated Hours
June 03	Create basic chess engine	25
10	Create basic chess engine	25
17	Create basic chess engine	25
24	Create basic chess engine	25
July 01	*Winter Vacation*	–
08	*Winter Vacation*	–
15	Design architecture of neural network to be used	10
22	Construct neural network to be used in evaluation function	15
29	Construct round-robin tournament framework	30
Aug 05	Design genetic algorithm (crossover point, mutation rates, etc.)	15
12	Implement abovementioned genetic algorithm	10
19	Implement abovementioned genetic algorithm	10
26	Training of chess engines through genetic algorithm	–
Sep 02	Training of chess engines through genetic algorithm	–
09	Training of chess engines through genetic algorithm	–
16	Training of chess engines through genetic algorithm	–
23	Training of chess engines through genetic algorithm	–
30	Compare evolved program with unevolved program and commercial engine	10
Oct 07	Use evolved program to compete against players online	20
14	Interpret results	15
21	*Examinations Week*	–
28	Report write-up	30
Nov 04	Report write-up	30
11	Report proofreading and presentation preparation	30
18	Presentation preparation	30
25	Research presentation	–

Table 4.1: Project schedule on a per-week basis

4.4 Potential Issues

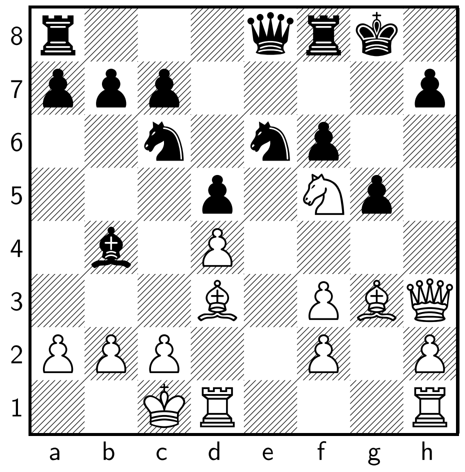
4.4.1 Running Time

One significant issue is the time required to evolve the chess engine. As an example, consider 20 agents contesting a round-robin tournament, where each agent plays every other agent twice. A total of 380 games would have to be played. If each game takes around ten minutes to be completed, then one tournament would take more than two days to finish. Since many tournaments need to be played, this situation is untenable.

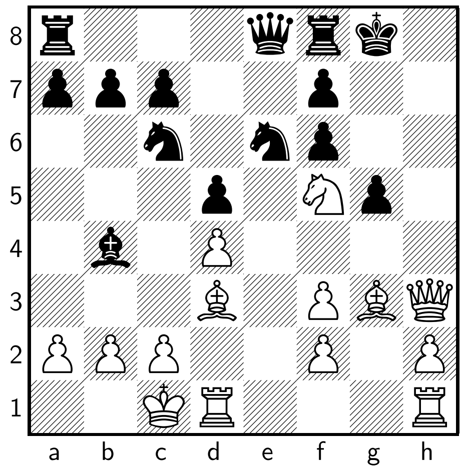
A solution is to limit the depth to which an engine searches. This will affect its play, but can be offset by running the tournament on high-powered computers, thus allowing the engine to search to a reasonable depth quickly. Another solution, which can be used in conjunction with the first, is to run a single tournament in parallel. Thus multiple games could be played simultaneously, which would greatly improve the time required to complete a tournament.

4.4.2 No Significant Improvement

At the conclusion of the project, it may be discovered that the chess engine that uses a neural network is no better than an engine that does not utilise one. While not ideal, this is not a failure — it is merely an indication that neural networks are not suited to the game of chess. Indeed, given that neural networks produce similar output for similar input [Cheung and Cannons 2002], this may not be too surprising. To illustrate, consider the two positions below, with black to play:



(a) Black's position is poor, but he can still play on.



(b) Black's position cannot be salvaged.

Figure 4.1: Similar positions with completely different evaluations

Though they may be almost identical, black's position is far inferior[1] in the right diagram. In fact, while black is behind in both games, he cannot prevent checkmate in the second position. Thus we see a situation where similar positions would yield vastly different evaluations.

4.5 Conclusion

Given the sheer scope of this project and the limited time available, a detailed and well-balanced schedule is of the utmost importance. This chapter attempted to make the proposed research more tractable by partitioning the project into manageable phases. A time plan, constructed from these phases, was proposed. This schedule allowed extra time for the genetic algorithm in order to offset any problems that might occur should the training need to be restarted. The chapter concluded by considering any potential issues that might arise and solutions to overcome them.

[1]His pawn shield is decimated, his king lies adjacent to an open file on which the opposing queen lies and black's queen is unable to assist in the defence of his king

Chapter 5

Conclusion

The evaluation function is key to any chess engine. It is also, above all else, the one component whereby a chess engine can improve its play, as suggested by Shannon [1950] decades ago. It is for this reason that attempts to train and evolve chess programs are centred around the evaluation function. The research proposed in this document is simply one more effort in this area.

Recall that a chess evaluation function requires a compromise. The function must be accurate enough, but not too detailed so as to negatively impact the search algorithm. Calculating certain features of a chess position is often an involved and time-consuming process. Using an ANN in conjunction with only a few other parameters may result in a relatively accurate evaluation that can be produced in less time than it would take an evaluation function with many parameters.

This paper proposes augmenting an evaluation function with a neural network. The function is then evolved and the final result tested against simple and sophisticated evaluation functions. A similar experiment has been conducted by Fogel *et al.* [2004], with some success. However, the evaluation function proposed by this research differs from that of Fogel *et al.* [2004] — the architecture of the ANN will be different and more parameters will be used to evaluate a position.

After constructing a chess program whose evaluation function makes use of an ANN, training must then take place. This involves optimising the weights of the evaluation function and neural network by means of a GA. A round-robin tournament will serve as the fitness function for each individual in the population.

The evolved program produced by the GA will be used to compete against a chess engine with a simple evaluation function, as well as an expert chess program. These tests will be performed in such a manner that the only difference between the engines will be their respective evaluation functions. This ensures that, if it is found that one engine is significantly better than another, it can be concluded that the first engine's evaluation function is superior to the latter's.

It is not hard to imagine why the idea of a self-improving chess engine is an attractive one. Modern chess programs have, as previously mentioned, become highly optimised searching machines. The focus of the computer chess community lies in improving algorithms and techniques that allow for deeper searches to be performed, with less thought devoted to creating intelligent chess engines. This notwithstanding, techniques from the field of artificial intelligence are indeed sometimes used, but often only to improve the weights of a static evaluation function. Both Aksenov [2004] and David-Tabibi *et al.* [2011] are examples of this.

This trend will undoubtedly continue, unless it can be shown that intelligent engines can be created. Not only must such an engine display the ability to learn, but it must also be able to compete with traditional chess engines at the highest level. It is therefore hoped that this research will give some indication as to the viability of a self-improving chess engine.

References

[Aksenov 2004] Petr Aksenov. Genetic algorithms for optimising chess position scoring. *Master's Thesis, University of Joensuu, Finland*, 2004.

[Beasley *et al.* 1993] David Beasley, Ralph Martin, and David Bull. An Overview of Genetic Algorithms: Part 1, Fundamentals. *University Computing*, 1993.

[Campbell *et al.* 2002] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134:57 – 83, 2002.

[Chellapilla and Fogel 2001] Kumar Chellapilla and David B. Fogel. Evolving an Expert Checkers Playing Program without Using Human Expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428, 2001.

[Cheung and Cannons 2002] Vincent Cheung and Kevin Cannons. An Introduction to Neural Networks. *Signal & Data Compression Laboratory, Electrical & Computer Engineering University of Manitoba, Winnipeg, Manitoba, Canada*, 2002.

[David-Tabibi *et al.* 2011] Omid David-Tabibi, Moshe Koppel, and Nathan S. Netanyahu. Expert-Driven Genetic Algorithms for Simulating Evaluation Functions. *Genetic Programming and Evolvable Machines*, 12(1):5–22, March 2011.

[FIDE 2008] FIDE. *Laws of Chess*, 2008. Retrieved 9 May 2013, from `http://www.fide.com/component/handbook/?id=124&view=article`

[Fogel *et al.* 2004] David B. Fogel, Timothy J. Hays, Sarah L. Hahn, and James Quon. A self-learning evolutionary chess program. In *Proceedings of the IEEE*, pages 1947–1954, 2004.

[Hauptman and Sipper 2005] Ami Hauptman and Moshe Sipper. GP-EndChess: Using Genetic Programming to Evolve Chess Endgame Players. In *Proceedings of the 8th European conference on Genetic Programming*, EuroGP'05, pages 120–131, Berlin, Heidelberg, 2005. Springer-Verlag.

[Montfort 2005] Nick Montfort. *Twisty Little Passages: An Approach To Interactive Fiction.* MIT Press, 2005.

[Russell and Norvig 2010] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 3 edition, 2010.

[Shannon 1950] Claude E. Shannon. XXII. Programming a computer for playing chess. *Philosophical Magazine Series 7*, 41(314):256–275, 1950.

[Standage 2002] Tom Standage. *The Turk: The Life and Times of the Famous 19th Century Chess-Playing Machine.* Walker, 2002.

[Stergiou and Siganos nd] Christos Stergiou and Dimitrios Siganos. *Neural Networks*, n.d. Retrieved 3 May 2013, from `http://www.doc.ic.ac.uk/~nd/surprise\_96/journal/vol4/cs11/report.html`

Index