Evolving Heuristic Based Game Playing Strategies for Board Games Using Genetic Programming

by

Clive Andre' Frankland

Submitted in fulfilment of the academic
requirements for the degree of
Master of Science in the
School of Mathematics, Statistics, and Computer Science,
University of KwaZulu-Natal,
Pietermaritzburg

January 2018

As the candidate's supervisor, I have/have not approved this thesis/dissertation for submission.

Name: Prof. Nelishia Pillay

Signed: _____

Date: _____

Preface

The experimental work described in this dissertation was carried out in the School of Mathematics, Statistics, and Computer Science, University of KwaZulu-Natal, Pietermartizburg, from February 2015 to January 2018, under the supervision of Professor Nelishia Pillay.

These studies represent original work by the author and have not otherwise been submitted in any form for any degree or diploma to any tertiary institution. Where use has been made of the work of others it is duly acknowledged in the text.

_____

Clive Andre' Frankland – Candidate (Student Number 791790207)

_____

Prof. Nelishia Pillay – Supervisor

Declaration 1 – Plagiarism

I, Clive Andre' Frankland (Student Number: 791790207) declare that:

1. The research reported in this dissertation, except where otherwise indicated or acknowledged, is my original work.
2. This dissertation has not been submitted in full or in part for any degree or examination to any other university.
3. This dissertation does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons.
4. This dissertation does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:
    a. Their words have been re-written but the general information attributed to them has been referenced.
    b. Where their exact words have been used, their writing has been placed inside quotation marks, and referenced.
5. This dissertation does not contain text, graphics or tables copied and pasted from the internet, unless specifically acknowledged, and the source being detailed in the dissertation and in the references sections.

Signed: _____

Clive Andre' Frankland

Declaration 2 – Publications

DETAILS OF CONTRIBUTION TO PUBLICATIONS that form part and/or include research presented in this thesis:

- Publication 1: C. Frankland, N. Pillay, “Evolving Game Playing Strategies for Othello”, *In: Proceedings of the 2015 IEEE Congress on Evolutionary Computation (CEC 2015)*, 25-28 May 2015, Sendai, Japan, pp. 1498-1504, 2015
- Publication 2: C. Frankland, N. Pillay, “Evolving game playing strategies for Othello incorporating reinforcement learning and mobility", *In: Proceedings of the 2015 Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*, 2015.
- Publication 3: C. Frankland, N. Pillay, “Evolving Heuristic Based Game Playing Strategies for Checkers Incorporating Reinforcement Learning”, NaBIC 2015 - 7th World Conference on Nature and Biologically Inspired Computing, pp. 165-178, 2015.

Signed:

_____ _____

Clive Andre’ Frankland Prof. Nelishia Pillay

Abstract

Computerized board games have, since the early 1940s, provided researchers in the field of artificial intelligence with ideal test-beds for studying computational intelligence theories and artificial intelligent behavior. More recently genetic programming (GP), an evolutionary algorithm, has gained popularity in this field for the induction of complex game playing strategies for different types of board games. Studies show that the focus of this research is primarily on using GP to evolve board evaluation functions to be used in combination with other search techniques to produce intelligent game-playing agents. In addition, the intelligence of these agents is often guided by large game specific knowledge bases. The research presented in this dissertation is unique in that the aim is to investigate the use of GP for evolving heuristic based game playing strategies for board games of different complexities. Each strategy represents a computer player and the heuristics making up the strategy determine which game moves are to be made. Unlike other studies where game playing strategies are created offline, in this study, strategies are evolved in real time, during game play. Furthermore, this work investigates the incorporation of reinforcement learning and mobility strategies into the GP approach to enhance the performance of the evolved strategies.

The performance of the genetic programming approach was evaluated for three board games, namely, Othello representing a game of low complexity, checkers representing medium complexity and chess a game of high complexity. The performance of the GP approach for Othello without reinforcement learning was compared to the performance of a pseudo-random move player encoded with limited Othello strategies, the GP approach incorporating reinforcement learning, the GP approach incorporating mobility and the GP approach incorporating both reinforcement learning and mobility. Genetic programming evolved players without reinforcement learning and mobility out performed the pseudo-random move player. The GP approach incorporating just reinforcement learning performed the best of all three GP approach combinations. For checkers, the approach without reinforcement learning was compared to the performance of a pseudo-random move player encoded with limited checkers strategies and the GP approach incorporating reinforcement learning. Genetic programming evolved players outperformed the pseudo-random

move player. Players induced combining GP and reinforcement learning outperformed the GP players without reinforcement learning. For chess, the performance of the GP approach incorporating reinforcement learning was evaluated in three different endgame configuration playouts against a pseudo-random move player encoded with limited chess strategies. The results demonstrated success in producing chess strategies that allowed GP evolved players to accomplish all three chess endgame configurations by checkmating the opposing King. Future work will look at investigating other options for incorporating reinforcement learning into the evolutionary process as well as combining online and offline induction of game playing strategies as a means of obtaining a balance between deriving strategies appropriate for the current scenario of game play will be examined. In addition, developing general game players capable of playing well in more than one type of game will be investigated.

Acknowledgements

The author acknowledges the financial assistance of the National Research Foundation (NRF) towards this research. Opinions expressed and conclusions arrived at, are those of the author and cannot be attributed to the NRF.

I thank my supervisor, Professor Nelishia Pillay, for her guidance, constant support and encouragement.

Special thanks go to my wife, Tammy, and my daughter, Tarryn, for their love and endless support during this study.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Purpose of this Study

Computerized board games have been used for decades to develop and investigate the behaviour of artificial intelligent (AI) agents. These game types have precise, easily coded rules that make them ideal test-beds for studying AI behavior and computational intelligence theories. The application of evolutionary algorithms for the induction of complex game playing strategies for different types of board games is gaining popularity in this field of research. Evolutionary algorithms are stochastic search methods that mimic the processes of natural biological evolution. One such evolutionary algorithm is genetic programming (GP).

Genetic programming has been used primarily to evolve board evaluation functions implemented in combination with other search techniques to produce strategies used by the computer players. An evaluation function calculates the strength of a particular board configuration after a move has been made. In most cases, these evaluation functions are produced off-line and usually in conjunction with domain specific expertise to guide the intelligence of the players. Previous work using GP has only focused on evolving evaluation functions that are used with other search techniques to generate game playing strategies. Genetic programming has not previously been used to generate the game playing strategies, thus the focus of the research presented in this thesis is to investigate the real-time evolution of heuristic based game playing strategies for board games using GP. The GP approach will be evaluated for games of varying level of complexity to investigate the scalability of the approach. The objectives of this dissertation are listed below.

1.2 Objectives

In order to fulfill the aim of this dissertation, an in depth survey of related literature on the development of AI in computerized board games, genetic programming and the application of GP

for producing game playing programs will be conducted. Based on the analysis of the literature, this study implements a GP approach for evolving heuristic based game playing strategies for board games. Three board games of different complexities were chosen, namely, Othello, checkers and chess. Othello representing a game of low complexity, checkers representing medium complexity and chess a game of high complexity.

The objectives of the research presented in this dissertation are as follows:

- Objective 1: Develop and evaluate the GP approach for Othello.
    - Evaluate the approach for a game of low complexity, namely, Othello.
    - Compare the performance of the approach to the performance of pseudo-random move players, the performance of the approach incorporating reinforcement learning, the performance of the approach incorporating mobility strategies and the performance of the approach incorporating both reinforcement learning and mobility strategies.
- Objective 2: Develop and evaluate the GP approach for checkers.
    - Evaluate the approach for a game of medium complexity, namely, checkers.
    - Compare the performance of the approach to the performance of pseudo-random move players and the performance of the approach incorporating reinforcement learning.
- Objective 3: Develop and evaluate the GP approach for chess.
    - Evaluate the approach for a game of high complexity, namely, chess.
    - Evaluate the approach for three chess endgame configurations.
    - Compare the performance of the approach incorporating reinforcement learning to the performance of pseudo-random move players for each of the endgame configurations.

1.3 Contributions

This dissertation makes the following contributions to the body of research investigating the application of genetic programming for evolving game playing strategies for computerized board games:

- This dissertation provides a thorough survey of contributing research in the field of AI for computerized board games.
- A survey and analysis of the application of GP for evolving game playing strategies for board games is provided.
- Based on a thorough investigation of the literature, this is the first study that investigates the use of GP to evolve heuristic based board game strategies. This research is also unique in that these strategies are evolved in real time without the use of game specific expertise or knowledge bases to build the intelligence.
- The scalability of the approach is investigated by evaluating the approach for games of varying levesl of complexity.
- This work extends and improves the GP approach by incorporating reinforcement learning and mobility techniques into the approach.

1.4 Dissertation Layout

This section provides a summary of the chapters presented in this dissertation.

Chapter 2 – Artificial Intelligence in Computer Games

In this chapter, an overview of artificial intelligence techniques used to generate game playing strategies is presented. Two game-tree search methods, the minimax algorithm with alpha-beta pruning and the Monte-Carlo tree search is presented. In addition, a section on machines that learn is presented, discussing artificial neural networks, evolutionary algorithms and neuroevolution.

Chapter 3 – Artificial Intelligence in Othello Research

In this chapter, the board game Othello is discussed, highlighting key areas of research contributing to the development of Othello playing programs. The Othello rules and strategies are outlined followed by an in depth discussion on artificial intelligence in Othello research. In this discussion, various methods for creating AI Othello computer players are reviewed. These methods include the minimax search with alpha-beta pruning, artificial neural networks, neuroevolution and genetic algorithms.

Chapter 4 – Artificial Intelligence in Checkers Research

An examination of the board game of checkers is presented in this chapter, briefly outlining the rules of the game and discussing some of the pioneering research that has been done on developing AI game playing agents for checkers. Two milestone projects that contributed significantly to AI in checkers and in game playing research as a whole is detailed. These milestone projects are the *Chinook* project implementing the minimax search with alpha-beta pruning and the *Blondie24* project implementing neuroevolution. This is followed by contributing research using genetic algorithms.

Chapter 5 – Artificial Intelligence in Chess Research

In this chapter research into artificial intelligent behavior using the classic board game of chess is examined. A background into approaches used to produce chess playing programs and related work is discussed. The controversial question of what constitutes AI and can deterministic searches such as the minimax search be considered artificial intelligence is briefly explored in the context of artificial intelligent chess playing programs. Research contributing to chess AI using artificial neural networks and genetic algorithms is also presented.

Chapter 6 – An Overview of Genetic Programming

This chapter presents an overview of genetic programming, discussing its application in context of this research. Sections of this chapter deal with the GP algorithm, the creation of the initial population, representation of individuals, selection methods, genetic operators and the GP algorithm parameters.

Chapter 7 – Genetic Programming in Board Game Research

Studies on the application of genetic programming for evolving game playing strategies for Othello, checkers and chess is presented in this chapter. Research into using GP for game playing, in most studies, describes the use of GP to evolve board evaluation functions that are used in conjunction with game tree search methods to induce game playing strategies. These search methods are used to explore the game tree to find the best candidate evaluation function in order to make a move.

Chapter 8 – Methodology

This chapter outlines the methodology used to achieve the objectives of this dissertation. A critical analysis of related literature is firstly presented. The aim and research methodology of this study is detailed, outlining the board games used to test the approach. A description of how this approach will be evaluated is presented followed by a technical specification of the hardware and software used to achieve the research objectives.

Chapter 9 – Genetic Programming Approach for Evolving Game Playing Strategies

In this chapter, the genetic programming approach for evolving heuristic based strategies for the board games Othello, checkers and chess is detailed. Sections present an overview of the GP algorithm detailing the GP representation, initial population generation, fitness evaluation, selection and regeneration. Incorporating reinforcement learning and mobility strategies into the approach is presented followed by a description of the GP algorithm parameters and justification for their values.

Chapter 10 – Results and Discussion

This chapter presents the results and discusses the outcomes of the GP approach for evolving game playing strategies for Othello, checkers and chess. In addition, an analysis of the best evolved game playing strategies for all three games is presented.

Chapter 11 – Conclusion and Future Research

Finally, this chapter provides a summary of the findings of this research, the outcomes of the objectives and how they were fulfilled and future extensions of this work based on the observations made during this research.

Chapter 2

Artificial Intelligence in Computer Games

2.1 Introduction

Artificial Intelligence (AI) according to John McCarthy, the father of AI, is the science and engineering of making machines intelligent through intelligent computer programs [1]. When behavioral scientists use the terms *intelligence* or *intelligent,* they refer to a collection of human cognitive behaviors. Similarly, when computer scientists speak of *artificial intelligence,* they are also referring to that same set of human behaviors. Although intelligence is associated with the way humans think, this same set of behaviors could be replicated using computational methods [2]. Artificial Intelligence is thus considered the simulation of human intelligence through computerized tasks [3].

In this chapter, an overview of AI techniques used to produce game playing programs is presented. Section 2.2 discusses board game AI followed by a description of two search methods used to develop AI players, the *minimax algorithm* with *alpha-beta pruning* and the *Monte-Carlo tree search*. These two search methods are presented in sections 2.3 and 2.4. Section 2.5 deals with machines that learn, discussing the contribution of *artificial neural networks*, *evolutionary algorithms* and *neuroevolution* to AI research in board games. Section 2.6 concludes with the summary of the chapter.

Ten years after the invention of the first programmable digital computer, in 1956 John McCarthy unveiled a new field of research called *artificial intelligence,* at a conference at Dartmouth College in New Hampshire [4]. AI was presented as a science, in which the phenomenon of *intelligence* would be studied using computers to simulate intelligent processes [5]. The first recognized AI research was presented by Warren McCulloch and Walter Pitts in 1943 [6]. Their research on brain neuron functionality lead them to propose a model of artificial neurons, in which each neuron is characterized as being *on* or *off*. The switch to *on* is triggered in response to stimulations by a sufficient number of neighboring neurons. McCulloch and Pitts

demonstrated that a network of connected neurons could compute any computable function. They went on to suggest that a suitably defined network could in fact learn. In 1950, Claude Shannon published the first paper on computer chess [7] signifying the dawn of the computer age and machine intelligence [8].

2.2 Board Game AI

Most computers, post second world war, in the 1950's were used for military applications, typically ballistic calculations for missiles [9]. In contrast, computerized board games provided a natural application for computer intelligence. This is an aspect the average person could relate too, and it was Turing in the early 1950s that proposed computerized board games as a means of studying AI [8]. The reasons being that board games have precise, easily coded unchanging rules that make them well suited to a programming environment [10]. These types of games provide competitive, dynamic environments that make them ideal test-beds for studying AI behavior, computational intelligence theories and architectures. Many founding figures of computer science, such as Alan Turing, John McCarthy, Claude Shannon and Arthur Samuel to name but a few have devoted the majority of their research to developing game-playing programs to be used in AI research [11]. In 1952 Arthur Samuel began his 25-year quest to build a strong checkers-playing program [12, 13]. This event opened the floodgates to developing computer programs that could rival human players [14]. It was not long after Samuel's checkers-playing program that the quest to develop a world chess playing program followed [15].

Board games like Othello, checkers and chess have been subjects of AI research for many years. These games are interesting because their strategies are complex with many possible board states while at the same time they are fully observable and deterministic. A board state is the configuration of the game board at a particular point in a game and is represented by the positions of pieces on the board. Deterministic games are those games that have non-random outcomes. For example, a piece moved in chess has to follow a strict move pattern that is governed by the rules of chess. These games also have a well-defined set of rules making them easy to implement [16]. Additionally, these game test-beds provide a quantifiable means of studying both the decision-making and the learning aspects of AI systems [17, 18] in that the rules of these games constrain the behaviour of the players in terms of which moves are legal and which moves are not. This

provides a fair amount of simplicity to the problem at hand. Furthermore, these games have a definite goal for the players to achieve and that is to win the game. This is accomplished by rewarding players that achieve particular goals through positive behaviours and good game-playing strategies under the constraints of finite resources such as the game pieces and board positions. Finally, these games provide enough variation that allow a wide range of complex behaviours, which are represented by the artificial game-playing agents. Therefore, attempts to develop intelligent (or at least pseudo-intelligent) computer players of strategy board games is an area of AI research that has been pursued since the field's onset [19].

Excelling at these games implies a high level of intelligence, so developing an intelligent game player would represent a significant step towards developing a more intelligent machine [5]. In recent years, AI has been used to derive complex game playing strategies for games such as chess, checkers, Othello, Chinese checkers, backgammon and until very recently, Go [20]. Traditionally, successful game playing strategies have been achieved through the input of human expertise, clever *brute force* programming techniques, huge opening move and endgame databases and effective *search strategies*. Two most notable examples of such search strategies are the *minimax algorithm with alpha-beta pruning* and the *Monte Carlo tree search*. The term *brute force programming* refers to a programming approach that considers every possible path to get to an answer. If one path is validated as being true then that is the answer otherwise go to the next possibility. The following sections will present the minimax search algorithm with alpha-beta pruning, the Monte-Carlo tree search method and machines that learn. In the section, Machines that Learn, three approaches for producing board game intelligence, namely, *artificial neural networks*, *evolutionary algorithms* and *neuroevolution* will be presented.

2.3 Minimax Search with Alpha-beta Pruning

In board games, the sequence of possible moves, starting at a current board position to a given depth is most commonly represented as a tree structure, termed *game tree*. Starting at the current board configuration, termed the *root node*, all possible moves are represented by *child nodes* in the tree. The terminating nodes are termed *leaf nodes*. Each leaf node is assigned a value determined by an evaluation function and is based on the board configuration at that node.

An evaluation function is used by the game playing programs to assess how promising a current board state in the game is by calculating a heuristic to determine what the chances of leading to a win from that state is. The evaluation function incorporates heuristics, namely rules that are applied to the leaf node board configuration to estimate the cost of moving from the current board state to that leaf node. The function looks only at the current position and does not explore other possible moves. Each level of the tree is termed a *ply*. Figure 2.1 represents a typical example of a game tree structure. The nodes represent board states and the edges represent moves.

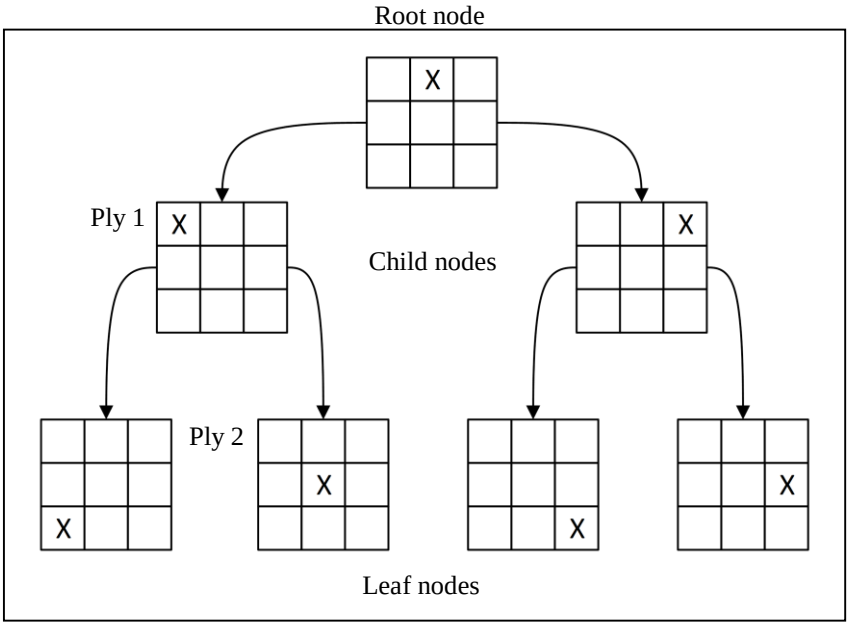


Figure 2.1. An example of a typical game tree.

The minimax algorithm is a search tree algorithm for choosing the next move in a two-player *zero sum* board game [21, 22, 23], based on values associated with each position or state of the game. Two-player zero sum games involve only two players and are termed zero sum games because one player always gains that which the other player loses, for example, chess, tic-tac-toe, checkers etc. The following example, including the minimax algorithm pseudocode in Algorithm 2.1, illustrates how the minimax search is implemented.

Algorithm 2.1. The minimax algorithm

1. Repeat
2. If search limit has been reached
   - Return the heuristic value of the node.
3. If maximizing the player
   - Use the minimax on the children of the current position.
   - Return the maximum value of the results.
4. If minimizing the player
   - Use the minimax on the children of the current position.
   - Return the minimum value of the results.
5. Until the entire tree is traversed

Two players, player *A* and player *B* are playing the game. Player *A* is to make a move. A game tree representing all the possible moves from the current board configuration (root node) to a predefined ply *P* is generated. The leaf nodes are represented at ply *P*. An evaluation function is used to determine the heuristic of each of the resulting board configurations at ply *P*. This is illustrated in Figure 2.2.

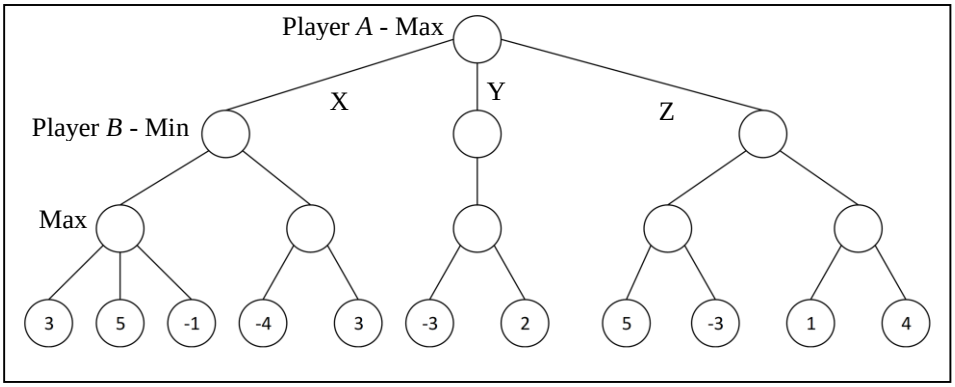


Figure 2.2. A game tree generated from root to ply 3

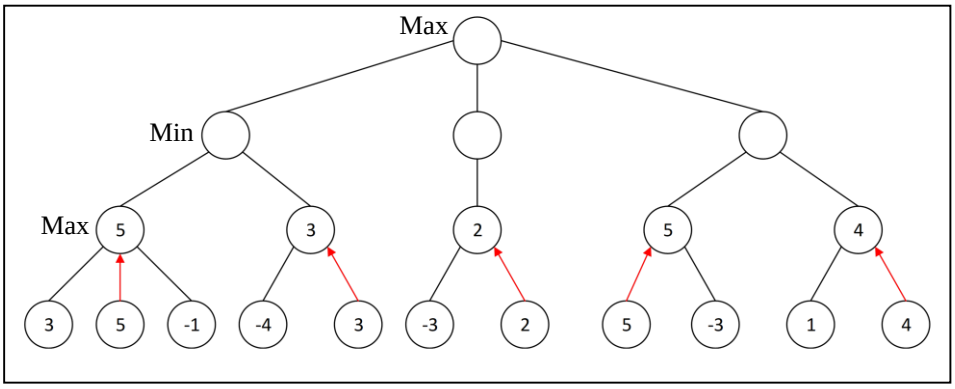


Figure 2.3. Maximize Player *A*'s score

A heuristic value is calculated for each leaf node that estimates how promising the node is to lead to a win. Player *A* has three move options, X, Y and Z. To achieve the best move Player *A* will attempt to maximize its score and minimize the opponents score. This can be achieved by propagating the node values up from the leaf nodes to the root node using the minimax search algorithm in the following way. Figures 2.3 to 2.5.

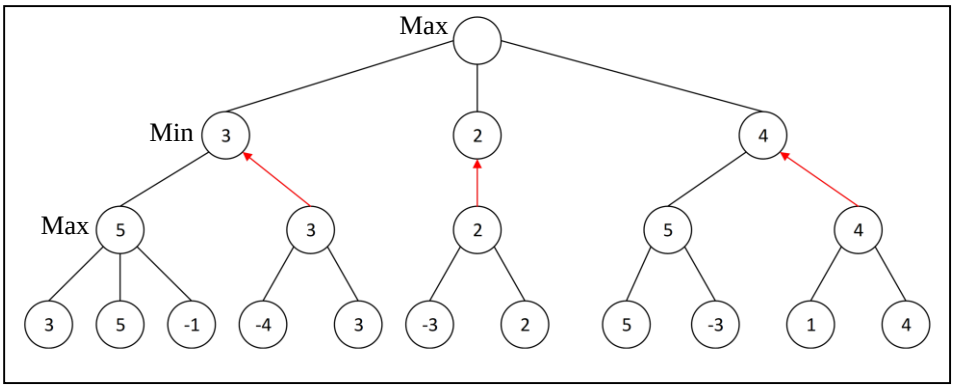


Figure 2.4. Minimize Player *B*'s score

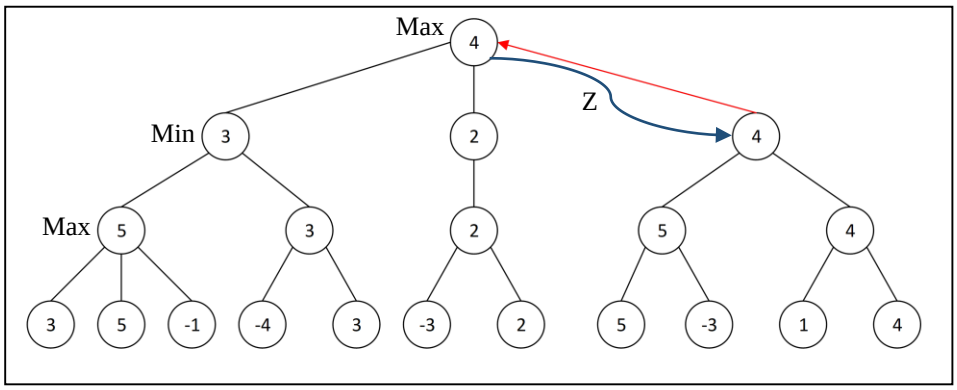


Figure 2.5. Maximize Player *A*'s score

The best move for Player *A* to make is move Z.

Alpha-beta pruning is an optimization method implemented in combination with the minimax search algorithm, which attempts to reduce the number of nodes that are evaluated by the minimax algorithm [24]. In this search method, when a state in the game tree has been found that proves to

be worse than a previously examined state it stops evaluating that state. The algorithm backtracks to the previous ply and the search continues from that node. This action, prunes the unexpanded branches from the search tree as these branches do not contain move possibilities better than the state being evaluated [22, 23]. An example illustrating this method is presented below along with the alpha-beta pruning algorithm in Algorithm 2.2. The *alpha* value represents highest value for the *Max* along the path. The *beta* value represents the lowest value for *Min* along the path. In this way *alpha* represents the score to be maximized and *beta* the score to be minimized.

Algorithm 2.2. The alpha-beta algorithm

1. Set the value of *Alpha* at the initial node to negative limit.
2. Set the value of *Beta* to positive limit.
3. Search down the tree to the given depth.
4. Evaluate the leaf node.
5. If the node is a leaf node then return the value.
6. If the node is a **Min** node then
    - For each of the children nodes apply the minimax algorithm with alpha-beta pruning.
    - If the value returned by a child node is less than the *Beta* value
        - Set the *Beta* value to this value.
    - If the *Beta* value is less than or equal to the *Alpha* value
        - Do not examine any more children nodes, i.e. prune the branch below from the tree.
7. Return the *Beta* value.
8. If the node is a **Max** node then
    - For each of the children nodes apply the minimax algorithm with alpha-beta pruning.
    - If the value returned by a child node is greater than the *Alpha* value
        - Set the *Alpha* value to this value.
    - If the *Alpha* value is greater than or equal to the *Beta* value
        - Do not examine any more children nodes, i.e. prune the branch below from the tree.
9. Return the *Alpha* value.

The following example below illustrates the alpha-beta pruning algorithm:

In this example 8 is propagated back and assigned to the beta value of the *Min* node at level ply. This is illustrated in Figure 2.6.

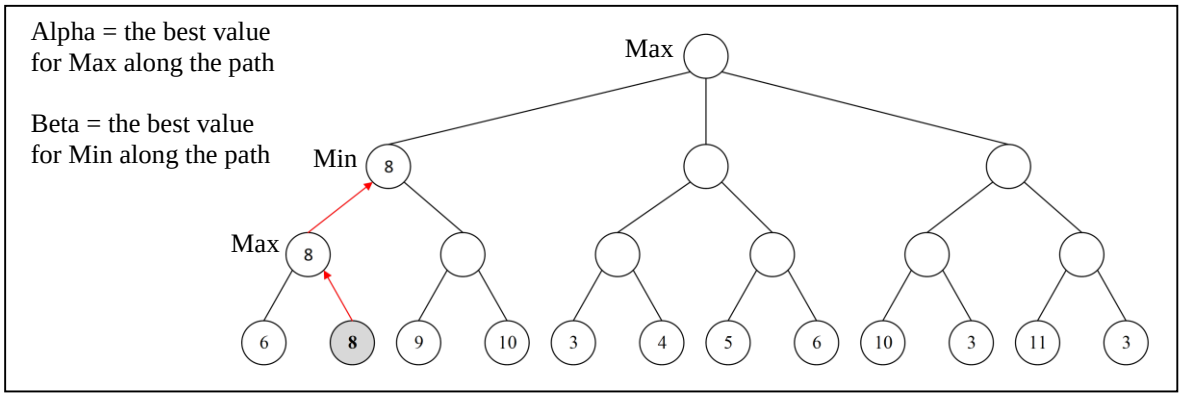


Figure 2.6. The minimax search algorithm with alpha-beta pruning

The next leaf node is explored. The value of 9 is propagated back to the alpha value of the *Max* node at level ply 2. This is illustrated in Figure 2.7.

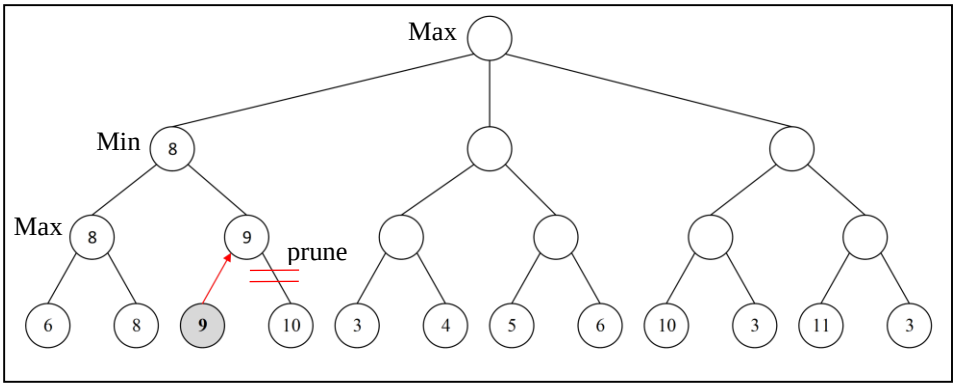


Figure 2.7. The minimax search algorithm with alpha-beta pruning

The *Max* alpha value is greater than the beta value of the *Min* parent node so the branch leading to the child node with value of 10 can be pruned from the tree.

The beta value of 8 is assigned to the root node. In a similar fashion, the next leaf node, namely 4, will be evaluated and its value propagated backwards to the beta value of the *Min* parent node. This is illustrated in Figure 2.8.

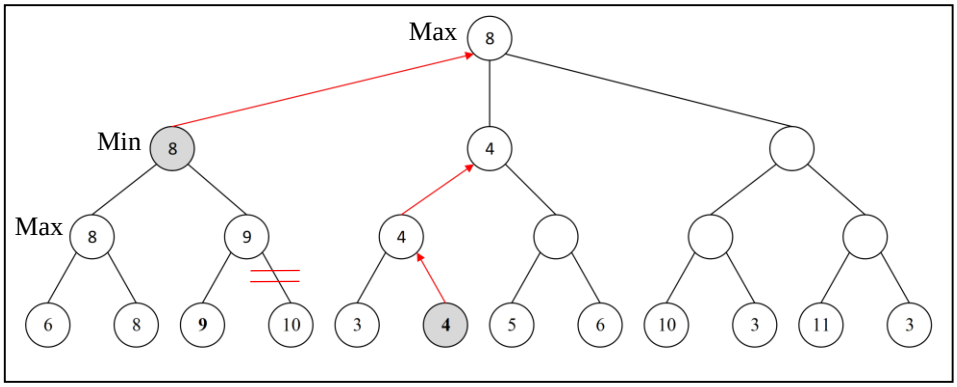


Figure 2.8. The minimax search algorithm with alpha-beta pruning

In this case, the *Min* beta value, that is 4, is smaller than the alpha value of the *Max* root node, that is 8, so the branch leading to the child node can be pruned from the tree. Shown in Figure 2.9.

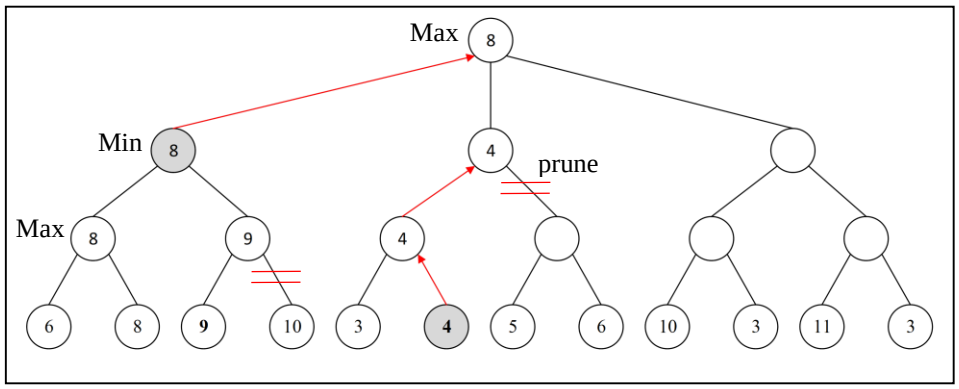


Figure 2.9. The minimax search algorithm with alpha-beta pruning

Exploring the right side of the tree, the minimax algorithm propagates the value of 10 back from the leaf node to the parent *Min* node beta value and 11 to the *Max* parent node. Illustrated in Figure 2.10.

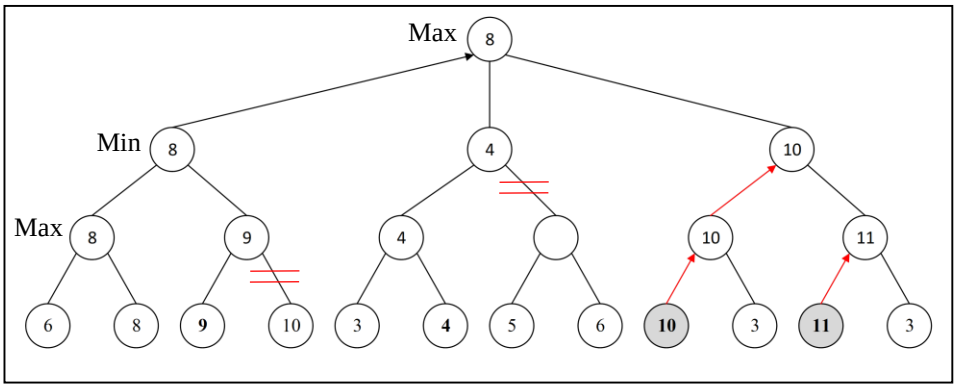


Figure 2.10. The minimax search algorithm with alpha-beta pruning

The *Max* alpha value of 11 is greater than the *Min* beta value of 10, therefore exploring the final leaf node will have no effect and is pruned from the tree. Finally, the *Min* beta value is greater than the *Max* root node value, therefore the root node is assigned a value of 10. The best move for the player to make is move down the right side path. Shown in Figure 2.11.

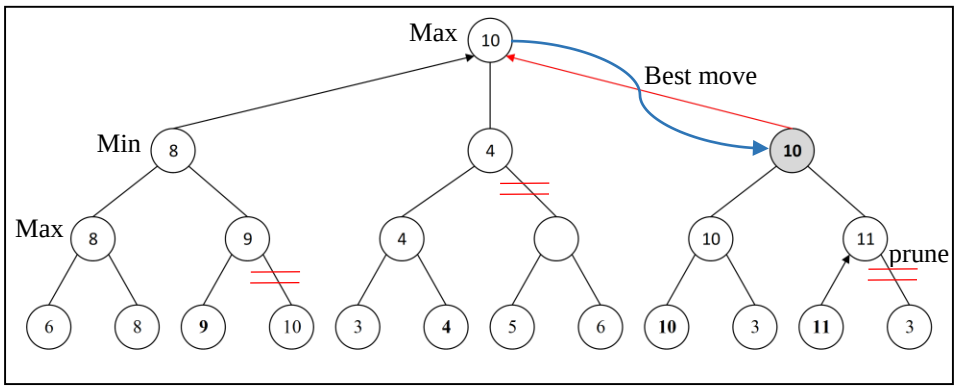


Figure 2.10. The minimax search algorithm with alpha-beta pruning

The success of this search method in game playing was highlighted in 1997 when the ©IBM's *Deep Blue* chess playing machine defeated Russian born grand master and world chess champion, Gary Kasparov [25].

2.4 Monte-Carlo Tree Search

For the majority of board games it is impractical to generate an entire game tree representing every move leading to every board state of that game. The Monte-Carlo tree search (MCTS) is a heuristic search algorithm that focuses on the analysis of the most promising moves by expanding

the search tree based on random sampling of the search space [26, 27, 28]. The application of the MCTS in board games is based on simulating many game *playouts*. Each node of the game tree represents one state of the game. Each node is assigned two values expressed as a ratio. The first is the number of wins in simulated games through the node and the second, the number of simulated games through the node. In each playout, the game is played out to the end or to a predefined ply by choosing moves at random. The final result of each playout is then propagated back towards the root node and used to weight the nodes along the path leading to the root node, i.e. the current board configuration. Once a move has been made, the next child node along the path becomes the new root node. The most common way playouts are used is to apply the same number of playouts after each legal move and then choose the move that leads to the most victories [29].

The MCST process can be sub-divided into four processes, namely, *selection*, *expansion*, *simulation* and *backpropagation*. The following illustrates these sub-processes. Figure 2.11 presents an overview of the MCTS algorithm.

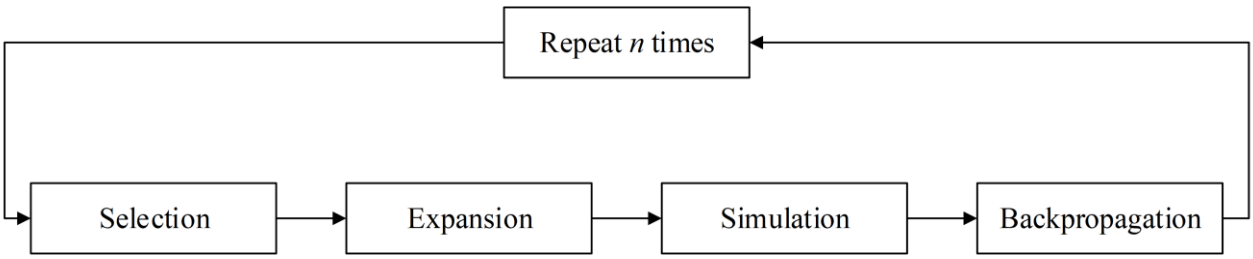


Figure 2.11. The Monte-Carlo Search Algorithm

Selection: Starting at the root node, child nodes with the greatest ratio of win rates per number of times visited are selected until a leaf node is reached. This causes the tree to grow more deeply down promising branches, i.e. those branches leading to wins. Consequently, the algorithm does not waste time exploring paths that lead to bad moves.

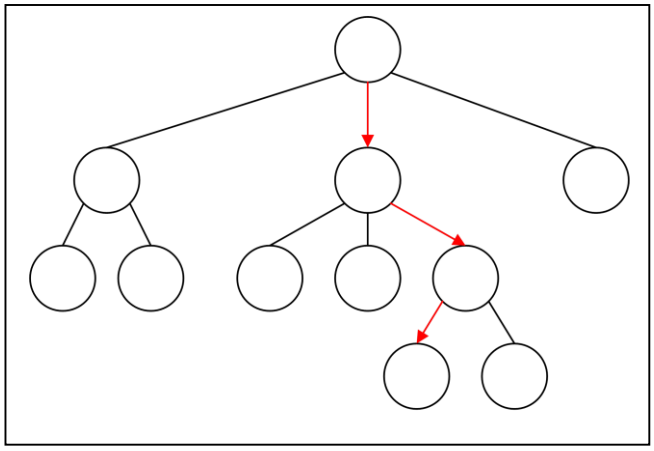


Figure 2.12. The Monte-Carlo Search - Selection

Expansion: If the leaf node is not the end of a game, i.e. a terminal node, one or more nodes are created and one randomly selected.

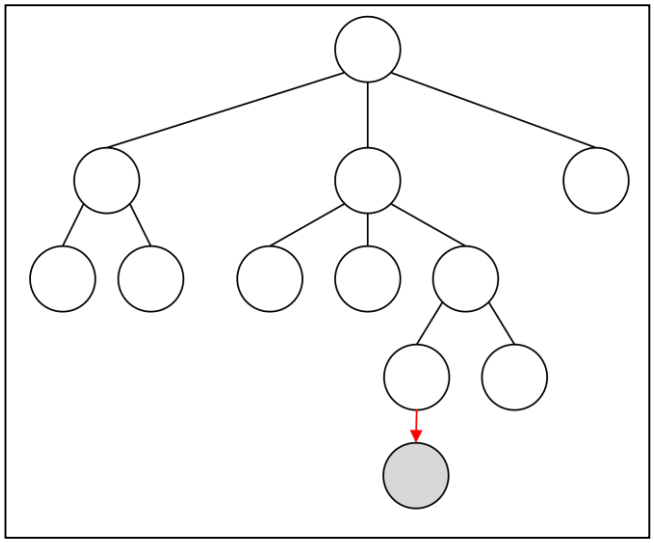


Figure 2.13. The Monte-Carlo Search - Expansion

Simulation: *p* number of playouts are run from the selected node.

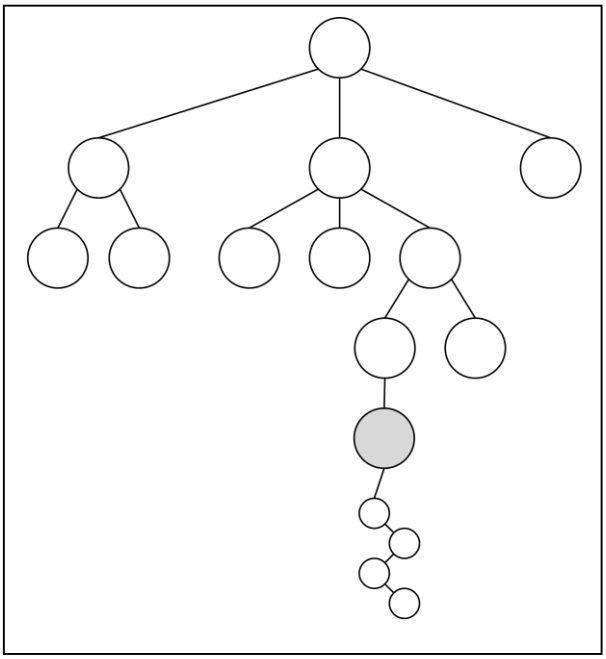


Figure 2.14. The Monte-Carlo Search - Simulation

Backpropagation: The move sequence from the current node to the root node is updated with the simulation result.

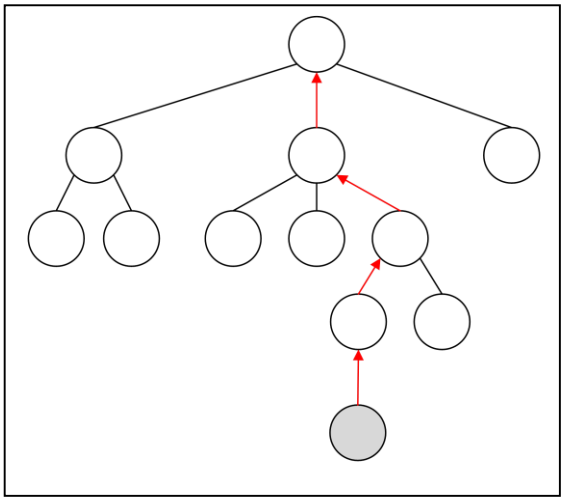


Figure 2.15. The Monte-Carlo Search - Backpropagation

The best move to make from the root node will be to the child node that has the greatest heuristic value, i.e. the best chance of leading to a win. This is the node with the greatest ratio of number of wins in simulated games through the node per the number of simulated games through the node.

The advantage of the MCTS is that it does not require any strategic or tactical knowledge about a game to make reasonable move decisions. The algorithm can function effectively with no knowledge of a game apart from its legal moves and game end conditions [30]. Furthermore, this search technique significantly minimizes the space search. The disadvantage of the MCTS is a speed issue in that it takes many iterations to converge to a good solution. For example, hundreds of thousands of playouts are required for the game of chess and millions for the game of Go [26, 31].

The impact of this powerful search strategy in game playing was demonstrated in 2016 when *AlphaGo*, a Go playing program developed by the ©Google DeepMind team, implementing a state of the art MCST method, beat the world Go champion and grandmaster Lee Sedol, 4 games to 1 [31, 32, 33].

2.5 Machines that Learn

Advances in AI research soon led to new innovative methods for developing machine learning agents. These agents are able to self-learn without human intervention and with limited domain specific expertise. The term *self-learning machines* defines those machines that learn by observation under limited supervision, and are able to adapt by observing the surrounding environment [34]. Self-learning machines using *artificial neural networks*, *evolutionary algorithms* are two such areas of research that have produced significant machine learning methods used to develop computer players for many types of board games [35, 36].

2.5.1 Artificial Neural Networks

Artificial neural networks (ANNs) are computational models based on the neural structures of the brain [37]. These artificial networks are designed to produce, or at least mimic, intelligent behavior. Unlike classical AI systems that are designed to directly mimic logical or rational reasoning, neural artificial networks give rise to intelligence as part of their adaptive anatomy [37,

38]. In combination with heuristic search methods, ANNs have produced AI agents that have surpassed human ability to play games. The reason ANNs have gained popularity for use in game playing programs is that for most board games it is not practical to store all the board states for evaluation. Chess, for example, has 10^{43} board states [39]. By training an ANN to recognize and evaluate board states from a pool of board states generated from a large number of game simulations, it is possible for the ANN to evaluate an unseen board state based on similar board states that it has learned.

A typical ANN consists of nodes that are connected to each other and exist in several different layers. These layers are the input layer, the hidden layer and the output layer [37, 38].

The following figure illustrates the structure of a simple artificial neuron, namely, the perceptron [38].

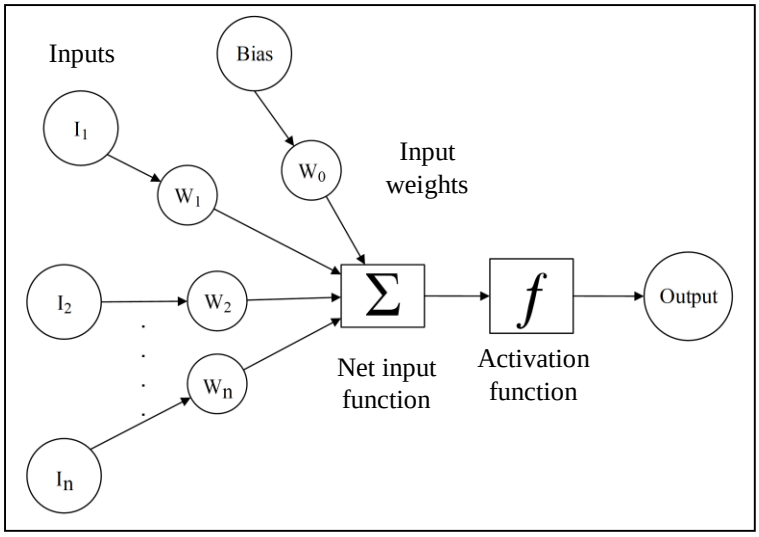


Figure 2.16. A simple artificial neuron

Net input function = Bias * $W_0 + \sum_n I_n W_n$ *Ref: Hajek* [38].

The activation function determines whether the neuron fires or remains at rest.

Trained artificial neural networks consisting of many hidden layers are most commonly used as evaluation functions in combination with game tree search algorithms to evaluate board states

at the leaf nodes of a game tree [31, 40, 41]. The board states are represented as vectors consisting of board positions and are mapped to the inputs of the trained ANN [40]. Each position is mapped to an input. Figure 2.17 illustrates this for a 3x3 Tic-Tac-Toe game board.

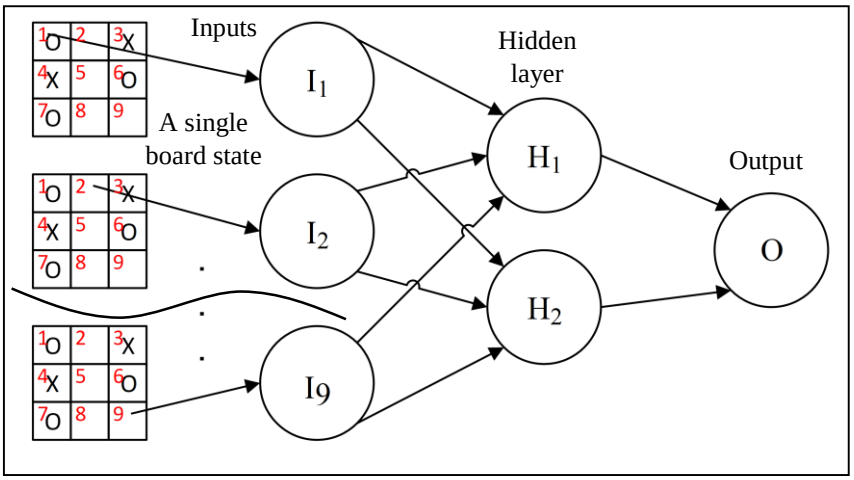


Figure 2.17. Board state mapped to a trained ANN

Artificial neural networks are usually trained by a combination of supervised learning from human expert games and reinforcement learning from games of self-play. Supervised learning takes place under the supervision of a *teacher*, so the learning process is dependent. During training, the input vector, i.e. the board state after a move is made, is presented to the network, which will produce an output vector. This output vector is compared with the target output vector, that is the board state after the best move is made, and an error signal is generated if there is a difference between the actual output and the target output vector. On the basis of this error signal, the weights of the inputs (Figure 2.16) are adjusted until the actual output is matched with the desired output. Reinforcement learning from games of self-play occurs when two similar ANN trained by tournament play against each other over a predetermined number of games. Board states that lead to wins will be used as target outputs for the learning ANNs.

2.5.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) are problem-solving programs, which use computational models of natural evolutionary phenomena in their design and implementation [18, 42]. These evolutionary algorithms are stochastic search methods that mimic the processes of natural biological evolution. Evolutionary algorithms operate on a population of candidate solutions by

applying the principle of *Darwin's natural selection* to produce progressively better approximations to a solution. At each generation, a new population of candidate solutions is created through the process of selecting individuals according to their level of fitness within the problem domain. These individuals are bred together using genetic operators. This process leads to the evolution of increasingly fitter populations, similar to that of natural adaptation. At the end of the evolutionary process, the fittest candidate solution is designated as the result. Two such evolutionary algorithms that are now widely used in the development of board game AI agents are *genetic algorithms* (GAs) [43, 44] and *genetic programming* (GP) [45, 46].

2.5.2.1 Genetic Algorithms

A genetic algorithm (GA) is a stochastic search that is inspired by Charles Darwin's theory of natural evolution as the algorithm reflects the process of natural selection where by the fittest individuals are selected for reproduction to produce offspring of the next generation [43]. The offspring inherit characteristics, namely genes, of the parents and if the parents have good genes, their offspring have a better chance of surviving. A GA encodes a potential solution to a problem in a chromosome-like data structure called the *genotype* or *genome*. The GA creates an initial population of random chromosomes which are evaluated on the basis of some fitness function. Those chromosomes that represent better solutions to the problem are given opportunities to reproduce, producing chromosomes for the next generation.

A population of chromosomes is depicted in Figure 2.18.

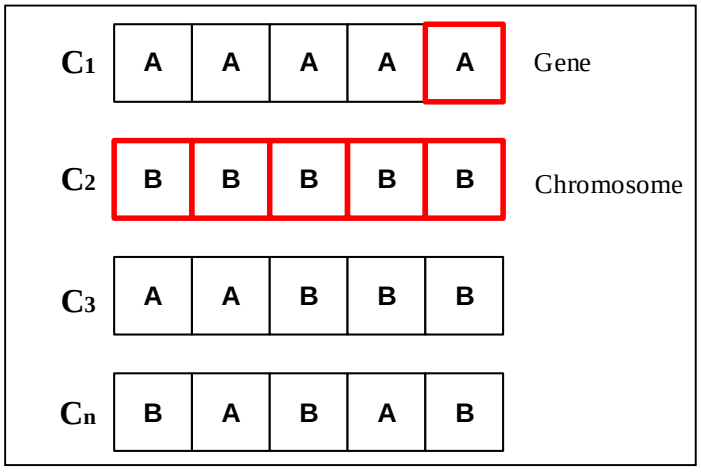


Figure 2.18. A population of chromosomes

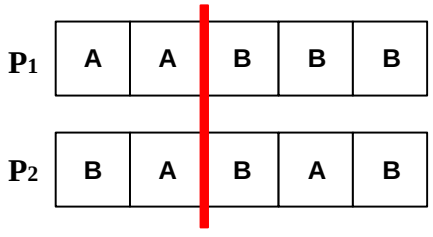
The following presents a basic overview of the genetic algorithm, Algorithm 2.3.

Algorithm 2.3. The genetic algorithm

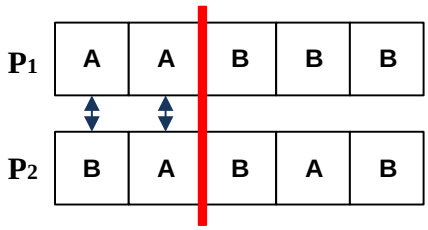
1. Create an initial population.
2. Evaluate the individuals in the population.
3. Select two individuals from the population for reproduction.
4. Transform the selected individuals by apply genetic operators, crossover and mutation.
5. Copy offspring into the new population.
6. Replace old population with new population.
7. Repeat 2 to 6 until a termination criterion is met.
8. Return the fittest individual.

The process begins with generating a population of *p* randomly produced chromosomes. Each chromosome is comprised of randomly selected parameter values. In board games, genes are usually encoded with the parameter values of a candidate board evaluation function. The fitness of each chromosome in the population is determined via a fitness function. This function calculates a fitness score for each chromosome based on its genes and is the ability of an individual to compete with other individuals in the population. The probability of an individual being selected as parents for reproduction is dependent on its fitness score. The next step in the process is to transform the selected individuals to produce off spring for the next generation. An individual may be copied unchanged into the next generation. The genetic operator in this case is referred to as the *reproduction* operator. Two other genetic operators, namely, *crossover* and *mutation* are used to transform the parents.

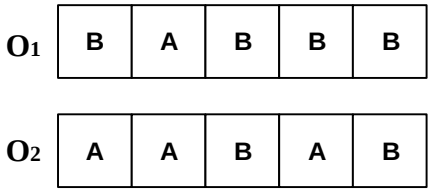
For *crossover*, two parents are selected. The same crossover point for each parent to be mated is chosen randomly. P = Parent, O = Offspring.



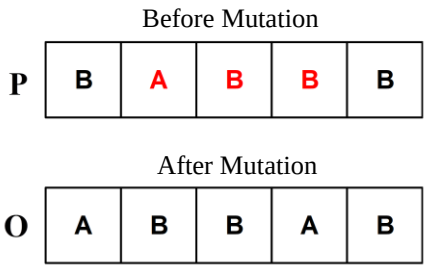
Offspring are created by the exchange of genes until the crossover point is reached.



The new offspring are copied into the next generation.



For *mutation*, one parent is selected. One or several genes are selected at random and their parameter value(s) changed to a random value within a preset range. Mutation is used to maintain diversity within the population and to prevent premature convergence. Convergence occurs when no offspring are produced that are significantly different from the previous generation.



The algorithm terminates when the population has converged, found a solution or reached a predefined criterion, e.g. a preset number of generations. In this GA overview the population has a fixed size and as new generations are formed, individuals with least fitness die off providing space for new fitter individuals. The sequence of processes is repeated to produce individuals in each new generation that are better than the previous generation.

The underlying difference between *genetic algorithms* and *genetic programming* is that GAs are search methods that search the solution space whereas GP is a search method that searches the program space. Genetic algorithms produce chromosomes that represent candidate solutions [43]. Genetic programming on the other hand is an evolutionary algorithm that evolves programs represented by *parse trees* [45, 46]. A detailed discussion of GP as it pertains to AI in board games and this dissertation will be presented in Chapters 6 and 7.

Current literature suggests that GAs and GP methods in game playing are used primarily to evolve the evaluation functions for board games [10, 47, 48]. Although game playing strategies evolved through evolutionary algorithms to date have not produced the same *sensational human vs machine* results as have *Deep Blue* and *AlphaGo*, they are set apart from traditional search methods in that they are able to learn without human input and limited domain specific knowledge, that being the rules of the game [9, 49, 50].

2.6 Neuroevolution

Neuroevolution is an AI approach that uses GAs to design and evolve ANNs. In this approach GAs are used to evolve the ANN weights and evolve the topology of the ANN [16]. In this way, for methods that evolve the ANN weights, the weights of connection can be stored in a chromosome. For those methods that evolve the topography of the ANN, chromosomes are made up of nodes and connectors. Connectors are weighted links that join two nodes.

Two commonly used neuroevolution techniques are *symbiotic adaptive neuroevolution* (SANE) and *neuroevolution of augmenting topologies* (NEAT) [51, 52].

The SANE technique uses a GA to evolve a population of neurons to form a neural network. In this way, evolving a population of neurons instead of a full network makes it possible to produce partial solutions to a proposed problem. Therefore, instead of solving the entire problem, individual neurons aim to solve a particular aspect of the problem. Each partial solution can then be combined with others to form a complete and effective solution to the problem [40, 47]. Hence, neurons are chosen from the population pool and combined to form a complete neural network.

Once the fitness of the formed ANN is determined, that fitness value is assigned to all of the participating neurons making up the neural network. This allows an individual neuron to be a part of any number of neural networks [16]. An extension of this technique termed *enforced sub-populations* (ESP) [53] evolves separate sub-populations of neurons. Neurons of one sub-population can only recombine with neurons of its own sub-population. Each hidden layer of the ANN is made up of neurons selected from only one sub-population of neurons. The advantage of ESP over SANE is that sub-populations specialize faster than in the case of SANE, where all specializations have to emerge from one large pool. Furthermore a drawback with SANE is the interbreeding of different specializations which result in many individuals with similar characteristics which tend to dominate over the still weak immerging specializations [16].

The NEAT method uses a GA to evolve both weights and topology of ANNs. Like SANE and ESP weights are evolved through generations allowing a better solution to be reached. In addition, changes can be made to the topology in terms of connectors and nodes. The initial topology of the ANN in NEAT is pre-determined to fit the problem to be solved. The genotype consists of node genes and connector genes. Node genes represent the input, hidden and output nodes. Hidden nodes are removed or added through evolution. Connector genes represent the links and their weights between nodes. A connector gene defines one link between two specified nodes and can be enabled or disabled through mutation and crossover operators.

2.7 Chapter Summary

This chapter provided a detailed overview of AI approaches used to produce the intelligence behind board-game playing programs. Two important, most commonly used search methods in game playing are the *minimax search* with *alpha-beta pruning algorithm* and the *Monte-Carlo tree search*. These methods were presented, highlighting their success in producing world-class game playing programs. The section covering machines that learn using *artificial neural networks*, *evolutionary algorithms* and *neuroevolution* was discussed, presenting each approach and its application to board games. Artificial intelligence research of three well know but significantly different types of board games, namely, Othello, checkers and chess is discussed in Chapters 3, 4 and 5 respectively. An overview of the evolutionary algorithm, *genetic programming* pertaining to this research, will be presented in Chapter 6 followed by a discussion, in Chapter 7, on the

application of GP for producing game-playing programs for the above three mentioned board games.

Chapter 3

Artificial Intelligence in Othello Research

3.1 Introduction

Othello is somewhat unique as far as board games go in that it is a disc-placing game. That is, moves are made by placing new pieces on the board rather than by moving existing pieces or removing them as in chess or checkers. Although the rules of the game are simple, the increase in the number of reversible coloured pieces during game play results in highly complex game playing strategies. These aspects and the fact that an entire game of Othello is played in 60 or less moves provides one with an atypical means of implementing and studying AI techniques.

In this chapter, the board game Othello is discussed, highlighting key areas of research contributing to the development of Othello playing programs. Section 3.2 introduces Othello describing its rules and strategies. Section 3.3 and 3.4 presents an in depth discussion on AI in Othello research and provides details of the historical development of Othello playing programs. In this discussion various methods for creating AI playing Othello agents are reviewed. These methods include the minimax search with alpha-beta pruning, artificial neural networks (ANN), neuroevolution and genetic algorithms. Section 3.5 concludes this chapter with a summary of the chapter's salient points.

3.2 Othello

Othello, also known as *Reversi*, was invented in the late 19th century by two Englishmen Lewis Waterman and John Molletta [16, 54]. It was formally published as a board game titled 'Reversi' in 1898. In 1974 the name Othello and the Othello tournament rules were formalized in Japan by the company Kabushiki Kaisha Othello. Othello is derived from the ancient Chinese game of Go and as in Go the strategies of Othello are concerned with the capturing of your opponents territory [16]. This game has a long research history and its attraction lies in the fact that Othello rules are simple, but complex strategies must be mastered to play the game well [55].

Othello, is a two-player zero sum game and the version of Othello used in this study is played on an 8x8 non-checkered board with 64 discs. All 64 discs are identical with one white side and the other black. To start the game each player takes 32 discs and chooses one colour to use throughout the game (black or white). At the start of the game one player places two black discs and the other places two white discs at the centre of the board. Each player takes turns placing discs on the board with their colour facing up. If a newly placed disc bounds the opponent's discs in a straight line, the bounded discs are to be flipped, adding to the player's disc count. The game ends when all of the positions of the board are occupied or no possible moves are left. The player with the higher disc count is declared the winner. In order to clarify Othello game-play, a brief description of the rules and strategies are presented is sections 3.2.1 and 3.2.2.

Details of the official Othello tournament rules can be found at the *USA Othello tournament rules website* [56].

3.2.1 Othello Rules

To start the game each player takes 32 discs and chooses one colour to use throughout the game. Black places two black discs and White places two white discs as shown in Figures 3.1 and 3.2. Two alternatives of start positions are shown in Figures 3.1 and 3.2. The game always begins with either one of these setups. Black always chooses the start configuration and starts the game.

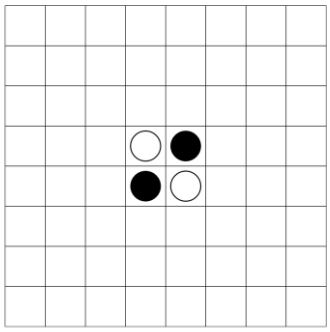


Figure 3.1. Othello start position 1

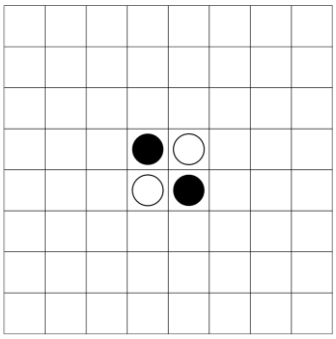
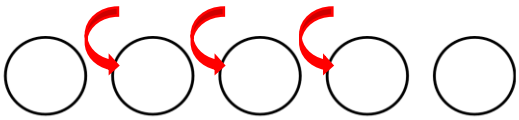


Figure 3.2. Othello start position 2

Each move consists of ‘outflanking’ the opponent's disc(s), then flipping the outflanked disc(s) to the colour of the player. In the example below, white outflanks black.

To outflank means to place a disc on the board so that your opponent's row or rows of disc(s) is bounded at each end by a disc of the players colour. A row may be made up of one or more discs. In the example below, white flips the outflanked black discs turning them to white.



If a player cannot gain one or more opposing discs by outflanking the opponent’s disc(s), that player’s turn is forfeited and the opponent moves again. However, if a move, that is the ability to outflank a disc or discs of the opponent, is available to a player, that player must make the move and cannot choose to pass.

A disc may outflank any number of discs in one or more rows in any number of directions at the same time - diagonally , horizontally or vertically. A row is defined as one or more discs in a continuous straight line (Figures 3.3 and 3.4).

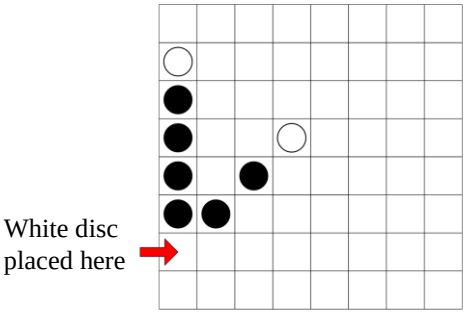


Figure 3.3. Othello rows before

Flipped discs

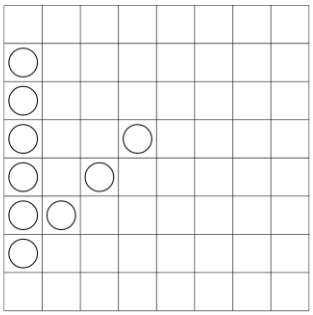


Figure 3.4. Othello rows after

Discs may only be outflanked as a direct result of a move and must fall in the direct line of the disc that is placed (Figures 3.5 and 3.6).

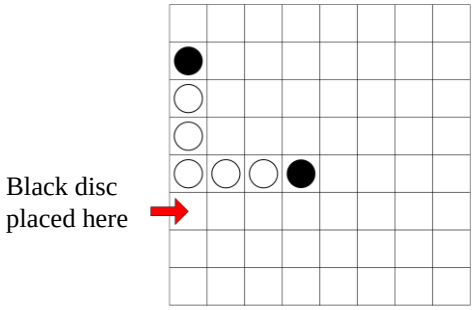


Figure 3.5. Othello outflanked row before

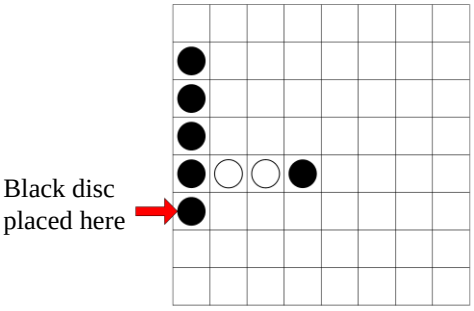


Figure 3.6. Othello outflanked row after

All discs outflanked in any one move must be flipped, even if it is to the player's advantage not to flip them at all. The game is over when it is no longer possible for either player to move. Discs are counted and the player with the most discs of their colour is declared the winner. It is also possible for a game to end before all 64 squares are occupied if no moves are possible.

3.2.2 Othello Strategies

Othello, as in other two-player zero sum games, is played within a finite number of moves, 60 moves or less. Since discs are placed on the board during play and not removed the number of positional options is reduced aiding look ahead to determine winning or losing positions. This and the fact that the rules of Othello are simple, makes Othello an ideal game for machines to play. The game is divided into the opening, mid and endgame. Each stage consists of approximately 20 moves. The battle to gain territory is waged in the mid-game where two important strategy types are employed, *positional strategies* and *mobility strategies* [16, 55].

3.2.2.1 Positional strategies

Positional strategies are used to capture discs directly and thus capture territory. These types of strategies are easy to learn but are less effective than mobility strategies. The two most important positional strategies are:

- *Capturing corner positions*: The most important positional strategy in Othello is to capture the corners. With the corners, a player can outflank the opponent’s discs situated on the edges or through the diagonal positions of the board. Besides capturing friendly discs (discs

of the same colour as that of the players discs), this gives a player the added advantage of staying in the game, as these corners cannot be outflanked. Positions around the corner are unsafe if played as this gives an opponent the opportunity to capture the corner position and thus gain the advantage.

- *Capturing edge positions*: The rows and columns on the edges of the board are secondly the most important positions to capture as this gives a player the added advantage for gaining discs across the board. Edge discs can only be outflanked by other edge discs, therefore players can use an edge position to flank an entire row thus gaining, for example, eight more discs on an 8x8 Othello board.

3.2.2.2 The Mobility Strategy

The mobility strategy is the most effective mid-game strategy, as its success always leads to supremacy in the endgame. This strategy is based on the ability to look ahead in order to force an opponent to forfeit moves or make moves to their disadvantage. This is accomplished by controlling the centre of the board forcing the opponent to surrender corner and edge positions. A low disc count and a larger number of available move positions characterize mobility for the player during the mid-game, as opposed to the opponent who will have many discs and very few move positions [55]. Mobility techniques are difficult to learn, not only for human players, but are extremely challenging for machines to learn [55].

3.3 Artificial Intelligence in Othello

The standard Othello board is an 8x8 lattice. Each position on the board can be black, white or empty. Othello has a low *branching factor* compared to other two-player zero sum games such as chess and Go. The branching factor is the number of child moves that are available at each level of the game tree. A game tree with a branching factor of two is illustrated in Figure 3.1.

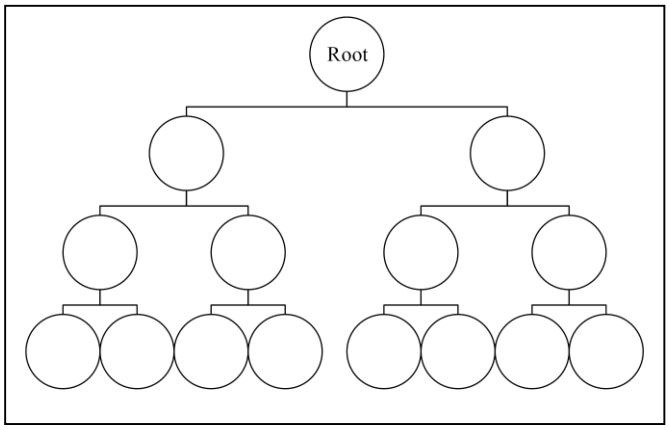


Figure 3.7 Game tree with a branching factor of two

Othello has an average branching factor of 10, checkers 4, chess 36 and Go 250 [54, 57]. This relatively low branching factor of Othello makes this an ideal game for computers to play. The minimax search with alpha-beta pruning [58] in combination with an extensive Othello knowledge base is traditionally the way Othello playing programs are developed [41]. In addition, the accuracy of the evaluation function (Chapter 2, subsection 2.5.2) in determining the *optimal route* for the search, within a reasonable amount of time, is key to the success of the game playing program. Improving the accuracy of the evaluation function means bettering the playing ability of the program [59, 60]. Othello programs developed using alpha-beta pruning algorithms in conjunction with expert game specific databases are now able to beat human world champions. This has resulted in a greater interest in the game in order to develop better Othello computer players [61].

3.4 Background and Related Work

The first Othello playing program to beat a world champion Othello player was a program named *Moor*, written by Mike Reeve and David Levy. *Moor* beat world champion Hiroshi Inoue one game out of six in 1980. This was the first time that an Othello playing machine beat an expert human Othello player [62]. The following subsections present the research implementing different approaches for producing Othello playing programs. These approaches are the minimax search algorithm with alpha-beta pruning in subsection 3.4.1, artificial neural networks in subsection 3.4.2, neuroevolution in subsection 3.4.3 and genetic algorithms in subsection 3.4.4.

3.4.1 Minimax Search with Alpha-Beta Pruning

This subsection discusses the research dealing with the implementing of the minimax search algorithm with alpha-beta pruning for Othello game playing programs.

Moor (Section 3.4) was developed using the minimax search with alpha-beta pruning combined with opening move and endgame tables. These tables were referenced by the programs evaluation function to calculate the value of each move. In the early 1980's, Rosenbloom presented his research into generating world-class game playing strategies for Othello. His Othello playing program likewise implemented the minimax search algorithm with alpha-beta pruning [54]. The evaluation function was based on an expert Othello knowledge base and together with *iterative deepening* and *move ordering* the program achieved a world-class level of play [54, 63, 64]. Rosenbloom's 'state of the art' Othello program named *IAGO* won the Santa Cruz Open Machine Othello tournament to become a world leading Othello competitor, surpassing some of the world's top human Othello players. Additionally *IAGO* was able to complete all of its moves within the 30 minute time allocation allowed in Othello tournaments. This was achieved through iterative deepening which determined the depth of search given the time constraints allocated for each move. By keeping track of how much time each move required, *IAGO* was able to allocate more time to deeper searches where necessary. IAGO made extensive use of move ordering by generating a move-ordering table at each node by looking at the rating of the responses to the move at that node. After the 46 move, IAGO was able to determine the outcome of the game, win, lose or draw, from any position after a move has been made. This gave *IAGO* an unquestionable advantage over its human opponents. The success of *IAGO* was evaluated against other available Othello programs (up until 1980), resulting in a 10-0 win record. Furthermore, by comparing the analysis of *IAGO*'s performance with those of expert human players, showed that *IAGO* was able to match their level of performance while avoiding some of their major strategy errors.

Despite *IAGO's* success as a world class Othello program, it has shortcomings in that the approach combines all of its game playing strategies into a single linear evaluation function for the entire game, even though different strategies are needed for different stages of the game [64]. Secondly, the application coefficients in the evaluation function are hand crafted, which leaves a significant margin for error [64]. Lee and Mahajan [65] addressed these short falls by creating a

program named *BILL* [66, 65], which implemented the minimax search algorithm with alpha-beta pruning but made use of pre-computed tables in its evaluation functions that allowed it to recognize hundreds of thousands of board configuration patterns. Pattern recognition techniques compare board configurations to each other during game play to determine the best moves. In this study the evaluation function was used to map a particular set of pattern features to an output value to be used in conjunction with the minimax search to select consecutive moves. An algorithm implementing Bayesian learning [67] was used to combine features of the evaluation function, which directly estimated the probability of winning. *BILL* was able to beat *IAGO* in 80% of the time all subsequent test matches. In February 1989, *BILL 3.0* was entered in the North American Computer Othello Championship and finished first out of thirteen programs.

Several noteworthy word-class Othello programs implementing the minimax search with alpha-beta pruning followed. Michael Buro developed a strong Othello playing program, namely, *LOGISTELLO* [68] that combined the minimax search algorithm with evaluation functions that calculated the value of a move by recognizing specific board configurations. Several million board positions were generated by allowing the program to play against itself which were used by the evaluation functions in conjunction with the minimax search to determine next best move. *NTest* created by Chris Welty superseded *LOGISTELLO* in 2004 followed by *Zebra* written by Gunnar Andersson in 2005.

WZebra [69], successor of *Zebra*, was written by Gunnar Anderson and Lars Ivansson and is currently one of the most formidable minimax Othello playing programs. The strength of this program stems from the combination of its minimax search algorithm and state of the art evaluation function that is able to reference an ever-increasing user defined knowledge base. This knowledge base stores the Othello game logic and positional strategies. *Herakles*, using a state of the art evaluation function and ever increasing knowledge base, is currently the strongest 10x10 minimax Othello playing program [70]. Apart from these world class Othello programs that are developed implementing alpha-beta methodology, today there are many powerful Othello playing programs that have been written, using ANNs, evolutionary algorithms and a combination of both. The following sections present some noteworthy examples of research that has been conducted in

developing Othello game playing programs using artificial ANNs, neuroevolution and genetic algorithms.

3.4.2 Artificial Neural Networks

This subsection discusses research dealing with implementing artificial neural networks for Othello game playing programs.

Leouski [64] presented an alternative to minimax search Othello playing programs by training a ANN to evaluate Othello positions via *temporal difference learning* [71]. Temporal difference (TD) learning, according to Sutton and Barto [71] is a prediction-based machine learning method, primarily used in reinforcement learning techniques. Leouski's approach is based on an artificial ANN architecture that reflects the temporal and spatial organization of the domain using the temporal difference predictive algorithm. Two networks of similar structure but with weights and biases initialized with random values were played against each other to achieve an intermediate level of play. The advantage of this method is that the ANN can be trained while playing itself and does not require precompiled training data or another Othello opponent. The network, comprising of one hidden layer of 50 units mapped preselected features of board positions to an output. The value of the output was determined using an evaluation function and each output estimated the probability of a winning move. The ANN was trained using TD learning with a two-move look ahead for a period of 30000 games using random board positions as the input. The Othello playing program for this study was evaluated against another Othello playing program, *Wystan* written by Jeff Clouse [72], and won 54 out of 60 games.

3.4.3 Neuroevolution

This subsection discusses research dealing with evolving artificial neural networks for Othello game playing programs.

A more promising approach to using ANNs to play computer games is to evolve populations of ANNs using evolutionary algorithms, termed *neuroevolution* [47, 40]. Moriarty and Miikkulainen [73, 47] used a genetic algorithm (GA) to evolve a population of ANNs for their Othello playing program. Each network in the population receives the current board configuration as its input and

uses a hand crafted evaluation function to determine the value of each possible move as the output. In contrast to traditional methods of searching through all possible game scenarios to determine the best move, the ANNs rely on pattern recognition to determine which move is the most promising. An evaluation function is used to assign a value to a particular pattern. This value is used as an output to determine which move is to be made. In addition, the evolved ANNs were required to differentiate among all possible moves, both legal and illegal ones, and then choose the best legal move to continue the game. Moriarty and Miikkulainen's Othello program was evaluated firstly against random-move players to determine the quality of strategies produced by the GA evolved ANNs and then secondly against a program using the minimax search with alpha-beta pruning. Populations of 50 individuals required 24 hours on an IBM© RS6000 25T computer to evolve any significant behavior. The GA evolved ANNs produced high levels of performance, producing complex Othello strategies such as mobility strategies and strategies defending edge and corner positions. The weakness of this approach is that the ANNs are evolved by playing against other Othello programs and as a result, the quality of the final evolved ANNs performance is proportional to the quality of the opponent Othello playing programs.

Chong *et al*. [40] in a similar study evolved a population of artificial ANNs over 1000 generations on a 1.8 GHz Intel© PC which took 16 days to complete. Each ANN represented an Othello strategy that was assigned a value calculated by a hand crafted evaluation function. During the evolutionary process the ANNs competed against each through tournament bouts. Selection was used to eliminate those ANNs that performed poorly relative to the other ANNs. A GA was used to mutate the weights and biases of individual ANNs on each generation. Two similar ANN systems were co-evolved and were allowed to compete against each other. For each game move the minimax search was used to select the ANN representing the best strategy to make the next game move. The ANN architecture was similar to the multilayer perceptron model used by Chellapilla and Fogel [36] to develop their self-learning world-class checkers program (discussed in Chapter 4). On each generation, two handcrafted Othello computer players were used to *monitor* the performance of the ANNs. The first monitor was programmed to capture as many pieces as possible and the second was programmed to capture strategic locations on the board. Each of the ANNs on each generation played two games against the monitoring programs alternating between black and white. A record was kept of the number of wins, losses and draws between the evolving

ANNs and the monitoring programs. By using two computer players, using different strategies, the response of the evolving ANNs to opponents playing different styles could be observed. In this way, an independent observation to the evolutionary behavior of the evolving ANNs was achieved.

Results of the experiment for this study revealed that, not only can co-evolved ANNs learn to play Othello without relying on preprogrammed expert knowledge, but also that these ANNs are able to produce complex game playing strategies such as mobility and positional strategies. The evolved ANNs played Othello much like human players do. That is, in the beginning, moves were made simply by placing discs in legal positions without any strategy, only for positional gain. As the ANNs evolved they produced more complex strategies such as mobility and adaptive game playing strategies.

3.4.4 Genetic Algorithms

This subsection discusses research dealing with the implementation of genetic algorithms for producing Othello game playing programs.

Alliot *et al*. [74] demonstrated improved performance of their Othello program by evolving the parameter values controlling the evaluation functions (Chapter 2, subsection 2.5.2) by means of a genetic algorithm. Given any board position, the evaluation function was divided into 12 parameters, 10 controlling the static evaluation of the board position, 1 for allocating bonuses for connecting discs along edges and 1 parameter for penalizing liberty discs (disc allowing the opponent to move). The Othello program implemented the minimax search with iterative deepening in conjunction with the evaluation functions to select a move. The Othello playing program was evaluated against various public domain programs and human players. Evaluating 100 elements (candidate evaluation functions) in a population evolved over 100 generations required 30 hours of computing time on a Hewlett Packard© 730 computer. Parallel processing was implemented and the population was split into three groups. Two groups of 33 elements and one of 34 elements. Each group was evaluated on a separate computer. The Othello program was able to beat all of its human opponents and won 75% of the games played against other public domain Othello programs.

Lia *et al*. [59] used a GA to evolve the parameter values controlling six different evaluation functions that were used to assign score values to Othello board configuration patterns. The parameter for each evaluation function included:

- Line state.
- Weighted nodes.
- Stability.
- Mobility.
- Unbalanced wedges.
- Even balanced wedges.

Each evaluation function was assigned a different set of parameters. A genetic algorithm was used to evolve the values of these parameter sets. Five hundred generations were used. Each evaluation function was used at different stages of game play. Two evaluation functions for the opening game, i.e. the first 1 to 12 moves, two for the middle game, i.e. 13 to 47 moves and two for the endgame, i.e. the last 48 to 60 moves. The minimax search algorithm with alpha-beta pruning searching to a depth of six was used in conjunction with the evolved evaluation functions to select moves. The Othello playing program for this study was evaluated against several public domain Othello playing programs, namely, Deep Green Reversi [75] and won 29 out of forty games 29-11, 3D Reversi [76] winning 28-12, WZebra [69] losing 9-31 and Herakles [70] losing 7-33.

In contrast to evolving good parameter values to improve evaluation functionality, Cunningham [20] presented a different approach for producing strategies for Othello playing programs using a GA to evolve board position heuristics. In this study, the minimax search algorithm was not implemented. Each position of a 6x6 Othello board was assigned a random value from a range of values -2 to 2 and used as position weights. The four quadrants of the board were symmetrically divided into six unique strategy areas, namely, corner positions, good and bad edge positions, good and bad bounding positions and the non-occupied center positions. These strategy areas were used by a hand crafted evaluation function to determine the value of a particular board configuration. A population of 100 chromosomes (Chapter 2, subsection 2.5.2) each consisting of 60 board position heuristic values was generated as the initial population and evolved over 30 generations. The

fitness of each chromosome (board configuration) was determined by iterating through all of the possible legal moves resulting from that board configuration, and then adding up the number of friendly (the colour being played) and enemy (opponent's colour) discs after each move. Each disc on the board was assigned the weight value of the square it occupied. The difference between the total sum of the friendly and enemy discs calculated after each move was determined to be the fitness of that board configuration. After 30 generations of evolution, the fittest board configuration was used to make a move against the evaluating program. The evaluating Othello playing program implemented a hand crafted evaluation function in conjunction with the minimax search, searching to a depth of six. The Othello playing program for this study proved successful beating the evaluating player in every game. This approach demonstrated that heuristic based genetic principles can be applied to developing Othello playing programs.

3.5 Chapter Summary

This chapter provided an overview of the board game Othello and discussed the AI research conducted for producing Othello playing programs. A discussion of related work detailing the various AI approaches was presented. These approaches included the minimax search with alpha-beta pruning, artificial neural networks, neuroevolution and genetic algorithms. Research into developing Othello playing programs has concentrated mainly on fine-tuning evaluation functions to be used with game tree search algorithms to make consecutive moves during game play. An in depth discussion on the contribution *genetic programming* has made on research in producing game playing programs will be presented in Chapter 7. This discussion will be preceded by an overview of GP, presented in Chapter 6. The following chapter, Chapter 4, presents a discussion on AI research for producing checkers playing programs.

Chapter 4

Artificial Intelligence in Checkers Research

4.1 Introduction

Information regarding the origins and history of the board game *Checkers* or *Draughts*, as it is known in Britain, is sparse. Literature obtained from the American Checkers Federation [77] and the British International Draughts Federation [78] both record that this game has ancient roots dating back as far as 3000 years B.C. Fragments of stone boards and disc shaped pieces were discovered in an archeological dig site at Ur in Iraq and carbon dating was used to determine the age of the fragments. Records of an ancient Egyptian 5x5 board game with similar rules to checkers, called *Alquerque* and dating back to 1400 B.C. have been discovered in various dig sites in Northern Africa and Western Europe. The first official scripted records of checkers, including its rules, were recorded in France in 1100 A.C., depicting a game, called *Fierges* or *Fereses*, played on an 8x8 checkered board with 12 pieces on each side. The rules for this game were adopted shortly afterwards in Great Britain for the game of *Draughts* and in America for *checkers*. In 1756 an English mathematician, William Payne wrote the first dissertation on Draughts, formalizing the game and its rules.

A discussion of the board game checkers is presented in this chapter, briefly outlining the rules of the game and discussing some of the pioneering AI research that has been done on developing checkers game playing programs. Section 4.2 presents the variation of the game used for this research followed by the background of checkers AI research and related work in section 4.3. This section includes, minimax search with alpha-beta pruning, neuroevolution and genetic algorithms. In addition, milestone projects that have contributed significantly to AI research in checkers are highlighted. These are, the first checkers playing program written by Arthur Samuel in 1955. The *Chinook* project which for the first time saw a grand master checkers player being defeated by a checkers playing program and *Blondie24* which became the first world-class checkers playing program that is self-learning without any built-in human expert knowledge.

Contributing AI approaches for checkers playing programs implementing minimax search with alpha-beta pruning, neuroevolution and genetic algorithms is presented in subsections 4.3.1, 4.3.2 and 4.3.3. The chapter concludes with a summary of the chapter in section 4.4.

4.2 Checkers

There are two official variations of tournament checkers played today, one version played on an 8x8 checkered board [78] and the other played on a 10x10 checkered board [79]. The 8x8 version will be dealt with, in this dissertation. The following points briefly describe the rules of checkers:

- Checkers is a two-player zero sum game (Chapter 2, section 2.3). Each player begins the game with 12 discs. One set coloured black and the other red or white.
- The board consists of 64 squares, alternating between 32 dark and 32 light squares. Usually black and white, black and red or red and white. The board is always positioned with the light square on the bottom right-hand side.
- Each player places their colour discs on the dark squares on their side of the board as shown in Figure 4.1.

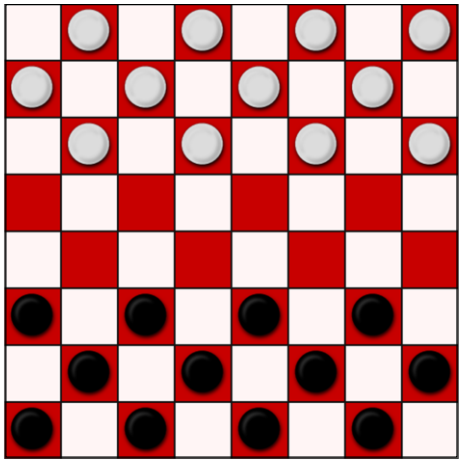


Figure 4.1. Setup of a checkers board

- Black always moves first and players alternate moves.
- Moves are made diagonally, one square at a time in the direction of the opponent.
- Capturing a piece requires the player to jump over the opponent's piece. Jumps are compulsory moves. Multiple jumps are allowed in any single turn, provided each jump

continues to lead immediately to the next jump. This is illustrated in Figure 4.2. Captured discs are removed from the board.

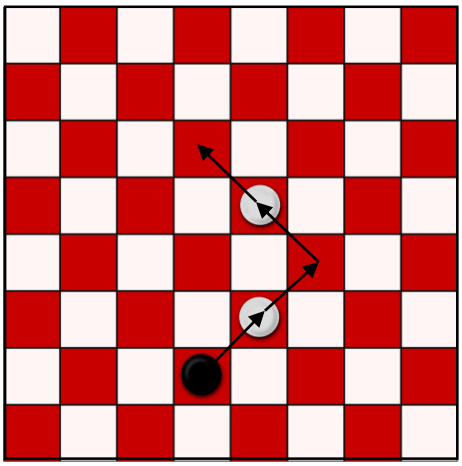


Figure 4.2. Black captures two white discs by jumping over them

- A piece moved onto the furthest row from the player's side is promoted to a King. This is illustrated in Figure 4.3. A second disc is placed upon that disc. A King cannot capture an opponent's disc on the first move after being promoted. King pieces may move in all four diagonal directions as indicated in Figure 4.3.

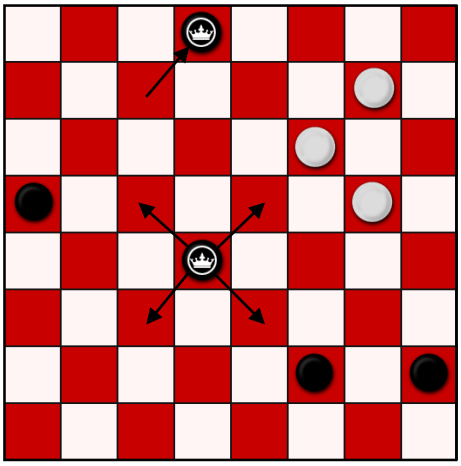


Figure 4.3. A promoted single disc becomes a King and can move and capture in all four diagonal directions

- The game ends when the loosing player is unable to move or all the discs of that player have been captured.
- A player wins by either capturing all of the other player's discs or putting them into a position where they cannot move. A player can also win if the other player resigns or forfeits the game as a result of a violation of the rules. At the end of play, if players have the same amount of pieces on the board, the player with the most Kings wins.
- A draw is declared:
    - If, at the end of play, players have an equal number of discs and the same number of Kings.
    - Upon mutual agreement of the players, or in a tournament, at the discretion of a third party under certain circumstances.
    - If no piece is captured or promoted to a King in the preceding fifty moves.
    - When neither player can force a win.
    - At any time, both players agree to it.

A full set of international tournament rules for checkers may be obtained from the official U.S.A. checkers tournament website [77].

4.3 Background and Related Work

Arthur Samuel wrote the first checkers playing program in 1955 on an ©IBM 701 computer and published his results in the IBM Journal in 1959 [12, 13]. This was to be a milestone in the history of AI research for developing smart checkers playing programs. The following subsections present the research implementing different approaches for producing checkers playing programs. These approaches are the minimax search algorithm with alpha-beta pruning in subsection 4.3.1, neuroevolution in subsection 4.3.2 and genetic algorithms in subsection 4.3.3.

4.3.1 Minimax Search with Alpha-Beta Pruning

This subsection details the research dealing with implementing the minimax search algorithm with alpha-beta pruning for checkers game playing programs and highlights a milestone in checkers AI, namely, the *Chinook* project.

Samuel [12, 13] combined two learning methods to for his checkers program, *rote learning* and *a generalization learning*. Rote learning refers to learning in which sequences are memorized for fast recall [80]. A record was kept of all board states encountered during play together with its backed up value determined by the minimax procedure. The result was that if a board state that had already been encountered were to occur again as a terminal position of the search tree, the depth of the search was effectively expanded, since for this board state, its stored value and children nodes are already known from the one or more searches conducted earlier. By referencing this memory record, the minimax search time was significantly reduced, thus allowing for deeper searches in less computing time. The evaluation function used to evaluate board states in this study implemented *generalized learning*. In this way, the parameter values of the evaluation function were re-evaluated and updated at different stages of the game. Two copies of the program were played against each other in a self-play tournament until the two programs' learning stabilized and no further improvement to the game playing strategies could be observed. In this study, the rote learning procedure learned continuously and was observed to be most effective in the opening and endgame scenarios. The opening game in checkers is roughly the first 8 to 12 moves. The endgame is when there is less than six pieces on the board including the King pieces. The *generalization learning* procedure contributed marginally to the overall performance of the game. Samuel's checkers playing program beat checkers master, Robert Nealey in 1962 but lost to grand masters Walter Hellman and Derek Oldbury in 1966 [81]. Although this checkers playing program is of a world class standard it has two flaws. The evaluation function was poorly implemented as the same function was used for board state evaluation throughout all the stages of the game. This led to the absence of proper strategizing during the opening, middle and endgames. Secondly, the generalized learning procedure performed poorly as it learned too slowly and did not contribute significantly to the overall performance of the game. In 1970 Jensen, Truscott and Bierman [82] from Duke University addressed these flaws by implementing an evaluation function that could adapt to the different stages of the game. Their minimax checkers playing program defeated Samuel's program but narrowly lost to grandmaster Elbert Lowder in a 5 game match, 1 win, 2 losses and 2 draws.

A significant milestone in AI research for checkers playing programs is the *Chinook* project. In 1989, a team of scientists from the University of Alberta, Edmonton in Canada, led by Jonathan

Schaeffer, set out with the short-term goal of developing a checkers playing program capable of defeating human world champions. Their long-term goal was to solve the game of checkers [83]. The project lasted 18 years. Schaeffer named the checkers playing program, *Chinook*, after the icy winds of the Canadian Prairies. The strength of *Chinook* lay in its effective minimax search with alpha-beta pruning and a highly accurate evaluation function that is able to reference a large expert endgame database. This database contained all the board configurations and moves of all positions with six discs or less, including the King pieces. The only flaw of this approach was that *Chinook* had a poor opening game database. The reason being, in order to achieve a significant opening move database, more than 300 000 board configurations would have to be manually entered and this was not possible at the time. This shortfall was remedied when Schaeffer *weakly solved* checkers [84, 85], ending his 18 year quest creating a near perfect checkers playing program. A weakly solved game is one that secures a win for one player or a draw for either, against any possible moves by the opponent from the beginning of the game [84]. For example, *Chinook* would never turn a drawn position into a losing position, but could turn a winning position into a draw if the opponent plays a poor move that cannot not result in a win. *Chinook* won its first *human vs machine* tournament in 1990, in the Mississippi State Open championship. There it defeated three grandmaster human players. That year *Chinook* achieved second place in the U.S. National Open championship, narrowly losing to the current world champion, Dr. Marion Tinsley. Tinsley had not lost a match in the last 40 years. This event saw *Chinook* performance elevated to world-class championship status.

4.3.2 Neuroevolution

This subsection presents research that deals with implementing neuroevolution for checkers game playing programs and highlights the *Blondie24* project.

The majority of the literature on using evolutionary algorithms to create checkers playing programs deal with variations of the approach for *neuroevolution*. These experiments all involve evolving populations of ANNs in an attempt to better the evaluation function. The ANNs are used as board state evaluation functions and the minimax search with alpha-beta pruning is used to select a move.

Blondie24 is an expert artificial ANN checkers playing program that was written by Kumar Chellapilla and David Fogel from 1999 to 2001 [36, 41]. The program was developed on a 400 MHz Pentium II computer. The intelligence of the program was achieved by evolving populations of ANNs to be used as board evaluation functions. The checkers-playing program achieved an expert rating on the international game-playing website *The Zone* [86] and became the first world-class checkers playing program that is self-learning without any built-in human expert knowledge. Using an evolutionary algorithm, Chellapilla and Fogel evolved a population of 15 multilayered, ANNs, each representing an evaluation function that was used to evaluate board positions. On each generation, each ANN in the population played one game against five randomly selected ANNs from the population. Each game was played using the minimax search using alpha-beta pruning to select consecutive moves. Those performing poorly relative to the other ANNs were eliminated. Offspring ANNs were created by iteratively updating the connection weights and biases values of their parents. No attempt was made to optimize the run-time performance and as a result, 100 generations took approximately two days to evolve. After 250 generations, the best performing ANN was tested against a range of human opponents over the internet. Over the course of two weeks, 90 games were played. *Blondie24* beat several expert players and on one occasion, drew against a master player. This study demonstrated that self-learning machines, for example *Blondie24*, could learn to play checkers without human intervention with only the rules of the game to guide them.

In a study conducted by Walędzik and Mańduzuik [87], two methods were combined to create evaluation functions. The first method, a *layered learning* method, which entailed the gradual, systematic training of the individual ANNs in a population, starting with endgame positions and gradually moving towards the root of the game tree (Chapter 3, section 3.3). The second method was to evolve the trained ANNs using a GA. The ANNs were comprised of 32 neurons, each one representing a position on the board, and14 layers. Each layer evaluated board positions depending the on state of the game, i.e. opening, middle or endgame. A vector containing a board description made up of five simple game-state description heuristics was used as the input to the ANN. Eight variations of the ANN architecture were implemented and trained. After the ANNs were trained, an evolutionary process was introduced. A population, consisting of 100 trained ANN (100 from each ANN variation), was evolved over 40 generations, using a genetic algorithm. Each ANN

variation produced a different evaluation function. The best performing ANN from each variation was selected to play against each other in tournament bouts of 20 games, alternating between red and black, to evaluate their performance. The minimax search with alpha-beta pruning, searching to a depth of four, was used to select a move. This approach was not evaluated against other checkers playing programs or human opponents as the results of the experiment simply determined which network architecture produced the best evaluation function. In this study, Walędzik and Mańduzuik did however present a new and innovative approach to creating evaluation functions for two-player zero sum games [88].

4.3.3 Genetic Algorithms

This subsection reports on research dealing with implementing genetic algorithms for checkers playing programs.

In a previous study to the one presented above, Kusiak, Walędzik and Mańduzuik [88] presented a heuristic based approach for evolving evaluation functions for checkers using a genetic algorithm. Their approach used two types of heuristics, linear and nonlinear. The linear heuristics made use of combinations of twenty-five predefined parameters relating to board configurations and the values of these parameters were calculated separately for each evaluation function in the population. The parameters were divided into three categories, simple features, layout features and pattern features. Each parameter represented a specific state of the checkers board. A genetic algorithm was used to generate coefficients of the linear combinations of heuristics. The coefficients of the heuristics were represented as a chromosome (vector) consisting of real numbers and each coefficient denoted a single gene. On each generation the population of evaluation functions were evaluated by randomly selecting four elements from the population and playing them against each other in tournaments of 5 games each. The fittest player was selected to be a parent to produce offspring for the next generation.

Combined with the linear heuristic method, three handcrafted, nonlinear evaluators, were used to evaluate board positions. Each was composed of a number of *IF then ELSE* conditions that divided the game into various disjointed stages representing the opening, middle and endgame. The linear combination of parameters was used by the nonlinear heuristics for board position

evaluation. The three nonlinear evaluators, were played against one another in tournament bouts of ten games each, alternating between red and black to determine their playing performance quality. Two points were assigned to the winner and one point for a draw. All the games were played using the minimax search with alpha-beta pruning to a search depth of six.

The performance of the winning evaluator was evaluated against three strong minimax checkers playing programs with their opening and endgame database functionality disabled. The first against *Simple Checkers*, set to search to a depth of six. The checkers playing program lost 49 games to one. Secondly, against *Actual Checkers 2000A* set to search to a depth of ten, it drew three of the four games and lost one game. Finally, against the checkers playing program, *Mad Checkers*, set to a search depth of six, the checkers playing program for this study won one game and drew three. The shortfall of this approach is that quality of the evaluation function is dependent on the nonlinear evaluator, which for this study was hand crafted. Secondly, the same evaluation function was used to evaluate the game through its entirety and not separately for the opening, middle and endgame states.

4.4 Chapter Summary

This chapter provided an overview of the board game checkers and presented the AI research conducted for producing checkers playing programs. A discussion of related work detailing the various AI approaches was presented. These approaches included the minimax search with alpha-beta pruning, neuroevolution and genetic algorithms. Three milestone projects that have contributed significantly to game playing AI research were highlighted. These being the first checkers playing program written by Arthur Samuel in 1955. The *Chinook* project which saw, for the first time, a grand master checkers player being defeated by a checkers playing program and *Blondie24* which became the first world-class checkers playing program that is self-learning without any built-in human expert knowledge. The following chapter, Chapter 5, presents a discussion on AI research for producing chess playing programs.

Chapter 5

Artificial Intelligence in Chess Research

5.1 Introduction

The origins of chess is undecided but scripted records of the game have been uncovered in China, Persia, Turkey and India dating back to 850 A.D. with references to the game dating back to the year 600 [89]. Chess was introduced to Europe in 1050 A.D. and soon became a feature of noble life. Over the ages, the rules of chess have been modified and are continually being updated but essentially the basic rules taken from the Persian and Turkish rules, 500 A.D. are still used today. Tournament rules today are based on the European rules of 1500 A.D. whereby the Pawn's opening move was extended to one or two squares forward, the Bishop could move diagonally over any number of squares and the Queen could move in any direction over any number of squares. Today chess is considered a *touchstone of the intellect* [11] and since the dawn of AI, researchers have been trying to find a solution that will allow a machine to play a perfect chess game [48].

In this chapter, a discussion of the board game chess is presented, briefly outlining the game and discussing some of the pioneering AI research that has been done on developing chess game playing programs. Section 5.2 presents the variation of the chess game used for this research followed by the background of computer chess playing programs and related work is discussed in section 5.3. In subsection 5.3.1, the controversial question of what constitutes AI and can deterministic searches such as the minimax search be considered artificial intelligence. This question is raised after ©IBM's minimax chess playing machine beat world champion and chess grandmaster Gary Kasparov. Section 5.3.2 looks at using artificial neural networks to produce chess playing programs. The use of GAs for evolving evaluation functions for chess playing programs is discussed in section 5.3.3. The chapter concludes with a summary in section 5.3.

5.2 Chess

Chess or *The Game of the Kings* as it is often referred to in the literature [90], is a classic zero sum two-player strategy board game. It consists of 32 chess pieces and an 8x8 board consisting of alternating black and white squares. The aim of the game is to capture the opponent's King. As simple as this sounds, chess is considered one of the most advanced board strategy games, second only to Go [90, 91]. The 32 piece, 8x8 checkered board version is used in this study. The official tournament rules for chess used in this dissertation can be found on the World Chess Federation website [92].

5.3 Background and Related Work

Computer chess has over the years become a significant part of AI research and since the mid 1960s, researchers in computer science have notoriously referred to chess as the *drosophila* of AI research. *Drosophila* are fruit flies used extensively in genetic research [39, 93, 94]. The metaphor *chess as the drosophila of AI* simply means that computer chess, like *drosophila*, represents a relatively simple system that nevertheless could be used to explore larger, more complex phenomena. Literature on AI systems often refer to chess as being the ideal experimental technology for AI because it is both simple enough to be able to formalize mathematically and yet complicated enough to be of theoretical interest [7, 95].

In 1950, Claude Shannon published an article titled "Programming a Computer for Playing Chess" in which he described the immense complexity of chess [7]. In his article, he presented the argument that playing a perfect game of chess using *brute force* programming methods (Chapter 2, section 2.1) is impossible, as there are an estimated 10^{120} nodes in a full-width chess tree. Chess has an average branching factor of 36 (Chapter 3, section 3.3) and even computers capable of evaluating 10^{16} positions per second would need in excess of 10^{95} years to fully evaluate the game tree. As a result, any chess program to be written will only be able to approximate or simulate the reasoning used by chess masters to pick moves from an extremely large search space [7]. The following subsections present the AI research that has contributed to producing chess playing programs.

5.3.1 Minimax Search with Alpha-Beta Pruning

This subsection deals with pioneering research that uses the minimax search algorithm with alpha-beta pruning for chess playing programs.

Claud Shannon in 1988 [39] presented his minimax search chess playing program in a paper that was to spearhead, in earnest, the study of computer chess in the science of artificial intelligence. The strength of this chess playing program lay in the fact that several evaluation functions, each encoded with a sizeable library of chess strategy rules, were implemented at each stage of the game, namely, opening, middle and endgame, in combination with the minimax search algorithm with alpha-beta pruning. The chess playing program was evaluated against several noteworthy human chess opponents and managed to beat several chess experts.

Following Shannon's publication, the focus of computer chess research moved towards building a chess machine that would defeat a world champion chess player [25]. In 1997, the *Deep Blue* chess machine created by ©IBM accomplished this goal by defeating Russian born grand master and world champion, Gary Kasparov in a six game match. The overall success of the machine and its incredible power was achieved through a combination of distributed computing and state of the art alpha-beta pruning techniques. *Deep Blue* ran on a 36-node ©IBM RS/6000® SPTM computer, and used 216 chess chips to carry out the alpha-beta minimax game tree searches. Each chip searched about 1.6 – 2 million chess positions per second. The overall search speed was 50 –100 million chess positions per second. Furthermore, *Deep Blue* made use of several state of the art evaluation functions that referenced expert opening game and endgame strategy databases. The opening game in chess is the first 12 moves of the game. The endgame in chess begins when there is less than 8 pieces left on the board. *Deep Blue* won four of the six games against Gary Kasparov.

The controversial question that soon followed Kasparov's defeat was "Did *Deep Blue* use artificial intelligence to play chess?" [96]. The website dedicated to covering the match between *Deep Blue* and Kasparov, was sponsored by ©IBM and made no reference to artificial intelligence at any stage during the tournament. In a news coverage after the tournament, ©IBM representatives responded to the question that *Deep Blue* did not use AI to play chess. According to renowned AI

specialist, Richard Korf [96], this claim is inaccurate and could only be true in terms of thinking that AI is the simulation of human thinking. By that measure, *Deep Blue* did not use AI to play the game. *Deep Blue* plays very differently to that of a human. The machine generates and evaluates 200 000 chess positions per second, something a human can never achieve. Artificial intelligence is much broader than the simulation of the human thought processes, and includes techniques that are in many ways not humanly conceivable [21, 96]. By this definition, *Deep Blue* defeated Kasparov using artificial intelligence. Artificial intelligence techniques such as heuristic search methods and in particular, the minimax search with alpha-beta pruning, were developed specifically to deal with problems such as selecting the best move outcomes in computer chess games. Hence, as the alpha-beta algorithm is key to the program's decision-making process, this clearly makes *Deep Blue* a product of AI, even though it does not encompass human cognitive processes.

5.3.2 Artificial Neural Networks

This subsection presents research that deals with implementing artificial neural networks for chess playing programs.

In the literature citing current research using artificial ANNs to develop the AI for chess playing programs, topics range from pure ANNs to hybridized versions as well as those networks incorporating reinforcement and temporal difference (TD) learning. Based on these literature reviews it is evident that artificial ANN based systems for creating chess playing programs are extremely computational expensive, requiring vast training sets and extensive expert databases. The resulting performance of these chess playing programs is however outstanding, in that most of them are able to play chess at master or grand master levels [97, 98, 48].

Thrun [97] presented an ANN chess playing program, *NeuroChess*, that incorporated temporal difference learning [71] to produce strong evaluation functions from the final outcome of games. The ANN, comprising of 165 hidden layer of 175 input nodes and 175 output nodes mapped preselected features of board positions to the 175 inputs. A vector of 64 board positions was passed to each of the inputs. The value of the output was derived by estimating the probability of a winning move by evaluating the input board configuration. The ANN was combined with the minimax

search algorithm and was pre-trained from a database of 120 000 expert games using TD learning looking ahead three moves. The pre-trained ANN was then trained and evaluated against a strong chess playing program, namely *GNU-Chess* [99], over two weeks and 2400 games. In the first stages of training, i.e. first 100 games, the chess playing program won only 13% of the games but its performance gradually improved until it won 316 out of the last 400 games. In a similar approach, adopted by Sharma and Chakraborty [100], an artificial ANN was trained using a database consisting of thousands of master and grand master games. The ANN was made up of 2 hidden layers of 16 input nodes each signifying one white chess piece and 16 output nodes each signifying one black chess piece. The trained ANN, combined with the minimax search algorithm, produced a strong chess playing program that was then used to produce high-level strategy hints for teaching human players to play chess.

David *et al.* [101] presented an *end-to-end* learning method for their chess playing program, *DeepChess*. In this research the end-to-end learning method refers to training the ANNs to learn an evaluation function *from scratch* without incorporating the rules of the game or any game specific features. Instead, the ANNs were trained from end to end from a large dataset of chess positions using a combination of unsupervised pre-training and supervised training without pre-programmed human expertise or *a priori* knowledge regarding chess strategies. *A priori* refers to the understanding of how certain things work based on prior knowledge [102]. The approach involved three phases. Firstly, unsupervised pre-training by training the ANN on a dataset of several million board states. This allowed the ANN to convert any given chess position into a vector of values representing high-level features of the game. High level features are those features representing, for example, the safety of the King, mobility of a piece, the ability of a piece to move out of danger and the ability of a piece to attack another piece to the advantage of the player. In the second phase, two copies of the pre-trained ANNs were connected via a second ANN. The second ANN implemented a handcrafted evaluation function that excluded *a priori* knowledge and was trained to predict which positions, received from the two trained ANNs, resulted in a win. This was done by comparing two chess positions, one from each of the trained ANNs and selecting the more favorable one. The ANNs received board states as an input and provided a score as the output. The score represented how good a board state is relative to other board states.

The ANN, representing the evaluation function, was combined with the minimax search algorithm with alpha-beta pruning to produce the chess playing program. *DeepChess* was evaluated against two international standard chess programs, *Falcon* [103] and *Crafty* [101]. *Falcon* achieved second place in the World Computer Chess Championships of 2008. *Crafty* is frequently used in the literature as a standard reference [101]. One hundred games were played against each. *DeepChess* ran four times slower than *Falcon* but won 64 out of 100 games and won 59 of the 100 games against *Crafty*. Without any prior chess knowledge, *DeepChess* managed to perform as well as other chess playing programs that have finely- tuned state of the art handcrafted evaluation functions. This study documents the first machine learning-based method that is able to learn from scratch to achieve grandmaster-level performance.

5.3.3 Genetic Algorithms

This subsection presents research dealing with implementing genetic algorithms for chess playing programs.

The bulk of the literature covering genetic algorithms used to develop AI chess playing programs deals with using the GAs to evolve parameter values for fine tuning evaluation functions [11, 48, 104]. The disadvantage with this approach is that in experiments that involve GAs to fine-tune evaluation functions, the functions are usually designed with fewer parameters than handcrafted evaluation functions and as a result do not produce high quality results. The issue of learning in computer chess is essentially an optimization problem and dependent on the evaluation function, which encodes most of the program's knowledge. Even automatic tuning methods that incorporate reinforcement learning [105] or temporal learning [106] have produced limited success in producing quality evaluation functions for chess playing programs. Good quality evaluation functions result in good quality programs [101].

Kendall and Whitwell [104] achieved limited success with a GA in their approach for tuning evaluation function parameters to be used with their minimax search chess playing program. The philosophy adopted in this study was that for game playing programs, the main problem arises in choosing the respective weighting of the parameters in the evaluation function. If these weightings are poor, poor evaluation functionality will be carried throughout the entire game. A slight change

in evaluation parameter weightings can be enough to completely change the entire game plan, and thus putting in place the mechanism to create different players with different playing strategies. So instead of handcrafting the parameter weightings for the evaluation function, it could be more beneficial to let an evolutionary algorithm evolve the weightings through learning techniques. In this approach, a previously developed, handcrafted evaluation function was used to generate a population of evaluation functions each with randomly assigned parameter values. The fitness of the population was improved through the process of evolution. Individuals that performed well were kept and selected as parents to produce the offspring for the next generation of parameters for the evaluation function.

The fitness of each evaluation function in the population was determined through tournament play against each other. The evaluation function was used with the minimax algorithm, searching to a specified depth, with alpha-beta pruning to decide which move to make. Starting with the first individual in the population, that player competed with the next individual. The winner was selected as a child for the next generation. A copy of the evaluation function was modified using the *mutation* genetic operator (discussed in Chapter 6 section 6.8.1.3) and placed into the next generation of evaluation functions. The losing player was discarded. In the case of a draw, both players were mutated and placed in the next generation. The winning player went on to play the next individual in the population. The fittest evaluation function, after 45 generations, was then selected to be evaluated against a chess playing program, namely, *Chessmaster 2100* [107]. *Chessmaster 2100* was set to look three moves ahead. To measure the success the evolved evaluation function, a handcrafted evaluation function was firstly evaluated against *Chessmaster 2100* in a two game tournament, alternating between black and white. This handcrafted evaluation function lost both games in less than 60 moves. The evolved evaluation function was then evaluated against *Chessmaster 2100* in a two game tournament and won playing white in 69 moves but lost playing black in 97 moves.

In a further study, David-Tabibi *et al.* [11] presented a successful approach to producing an expert minimax search chess playing program in which a GA was used to evolve the parameter values of an evaluation function. The success of their technique, termed *mentor-assisted evaluation function optimization*, was achieved by evolving an evaluation function that simulates

the behavior of a superior expert evaluation function. In this case, the expert evaluation function is not available for scrutiny or personal use but forms part of a commercially available chess playing program. Their novel expert-driven approach effectively evolved the parameter values for their evaluation function from randomly initialized values to highly tuned ones, producing an evaluation function that yielded similar performance to that of the expert evaluation function.

While the evaluation functions of most chess playing programs are well guarded and unavailable for scrutiny, the score values of a board states after evaluation can be extracted thus allowing the score for any given board state encountered in a game to be revealed [11]. By utilizing these scores, David-Tabibi *et al.* were able to evolve the evaluation function parameters for their chess playing program, *Evol*, by allowing it to learn from the expertise of another evaluation function. The *Falcon* minimax chess program [103] was used as the *expert*. A population of 1000 evaluation functions, each with 40 randomly initialized parameters, was generated and evolved over 300 generations by simulating the behavior of the *expert* evaluation function. On each generation the *crossover* and *mutation* genetic operators (discussed in Chapter 6 sections 6.8.1.2 and 6.8.1.3) were used to modify the evaluation functions. Initially, 10000 board states from a database of grandmaster chess games were randomly selected. *Falcon* was used to evaluate each of the board states. The evaluation scores were extracted and stored to be used as the evaluating criteria for the evolving evaluation functions. On each generation, each evaluation function in the population was subject to evaluate 1000 randomly selected board states from the set of 10000 initially selected. Based on the evaluation scores *Falcon* obtained for each of these board states a fitness value was calculated for the evaluating function. After 300 generations, the best performing evaluation function was selected and used in conjunction with the minimax search algorithm to produce the chess playing program. *Evol* was evaluated against *Falcon* and *Crafty* in tournament bouts of 300 games each alternating between white and black. *Evol* won 54% of the games against *Falcon* and 59% of the games against *Crafty*. The authors did not specify how far *Falcon* and *Crafty* were set to look ahead. *Evol* went on to take 2nd place at the 2008 World Computer Speed Chess Championships and 6th place at the World Computer Chess Championships.

5.4 Chapter Summary

This chapter provided an overview of the board game chess and presented a discussion on the AI research conducted for producing chess playing programs. The milestone accomplishment by ©IBMs chess playing machine highlighted the advances made in machine learning technology when it beat world chess champion Gary Kasparov. A discussion concerning whether the minimax search approach constitutes AI was also presented. Further, research covering the implementation of ANNs and GAs for evolving the evaluation functions for chess playing programs was covered. Chapter 6 presents a detailed overview of genetic programming as implemented for this dissertation.

Chapter 6

An Overview of Genetic Programming

6.1 Introduction

Programming computers to automatically solve problems is central to artificial intelligence, machine learning and the general area of AI research covered by that which Turing called "machine intelligence" [108]. Genetic programming (GP) was introduced by Koza in 1990 and is an evolutionary algorithm, which searches a program space rather than a solution space, which is typical of genetic algorithms [45]. Genetic programming functions by applying the power of evolution by natural selection [109] to artificial populations of *computer programs*. Like in nature, offspring are not identical to their parents but are changed through small amounts of random mutations and through the recombination of parental genes. These offspring, as a result, may not be as fit as their parents but every so often a mutation occurs or an improved combination is produced resulting in an offspring that is fitter than its parents. These offspring have a better chance of surviving to be selected as parents to produce future generations of offspring.

This chapter presents an overview of genetic programming, introducing the GP algorithm in section 6.2. A program produced by GP is represented as a parse tree made up of terminal and function nodes and is described in section 6.3. Terminals and functions are discussed in section 6.4. The various methods used to generate an initial population of candidate programs are described in section 6.5. Each individual in the population must be evaluated to measure how able the individual is at solving the problem at hand. This is a measure of the individual's fitness. The evaluation of individuals to determine their fitness is described in section 6.6. Selection of individuals from a population to be parents is discussed in section 6.7. The genetic operators used to alter parents to produce offspring are presented in section 6.8. The parameters that are used by GP algorithm are discussed in section 6.9. The GP algorithm is executed until a termination criterion is met. Terminating and solution designation are discussed in sections 6.10. Section 6.11 provides a summary of this chapter.

6.2 The Genetic Programming Algorithm

Genetic programming was introduced by Koza [45] and is an evolutionary algorithm inspired by biological evolution to evolve programs that can provide solutions to problems [110]. The GP algorithm is applied to a search space of programs, each representing a set of instructions. A fitness function is used to measure how well those instructions can perform a particular task or to solve a given problem. The GP algorithm is illustrated in Algorithm 6.1. The basic control flow of the GP algorithm is represented in Figure 6.1. In GP a population of programs is stochastically transformed into new, hopefully better, populations of programs through the process of evolution. The population is modified by applying genetic operators (section 6.8) to individuals in the population over a pre-determined number of generations, or until a termination criterion is met. This is termed a *run* [45, 110]. Furthermore, since GP relies on randomness and uses a random seed value on each execution of the algorithm, a different solution can be obtained on each run.

The fitness function determines how well a program works by running it through an interpreter, and then comparing its behaviour to some ideal. This comparison is quantified to give a numeric value called fitness. Those programs that do well are chosen to breed and produce new programs for the next generation. The process is terminated when a solution is found or some other criteria are met, for example, the preset number of generations reached.

Algorithm 6.1. The genetic programming algorithm

1. Randomly create an initial population of programs from elements of the terminal and function sets
2. Repeat
   - Execute each program to determine its fitness
   - Select one or two parent program(s) from the population based on fitness
   - Create new individual program(s) by applying genetic operators with specified application rates
3. Until a termination criteria has been met
4. Return the fittest individual

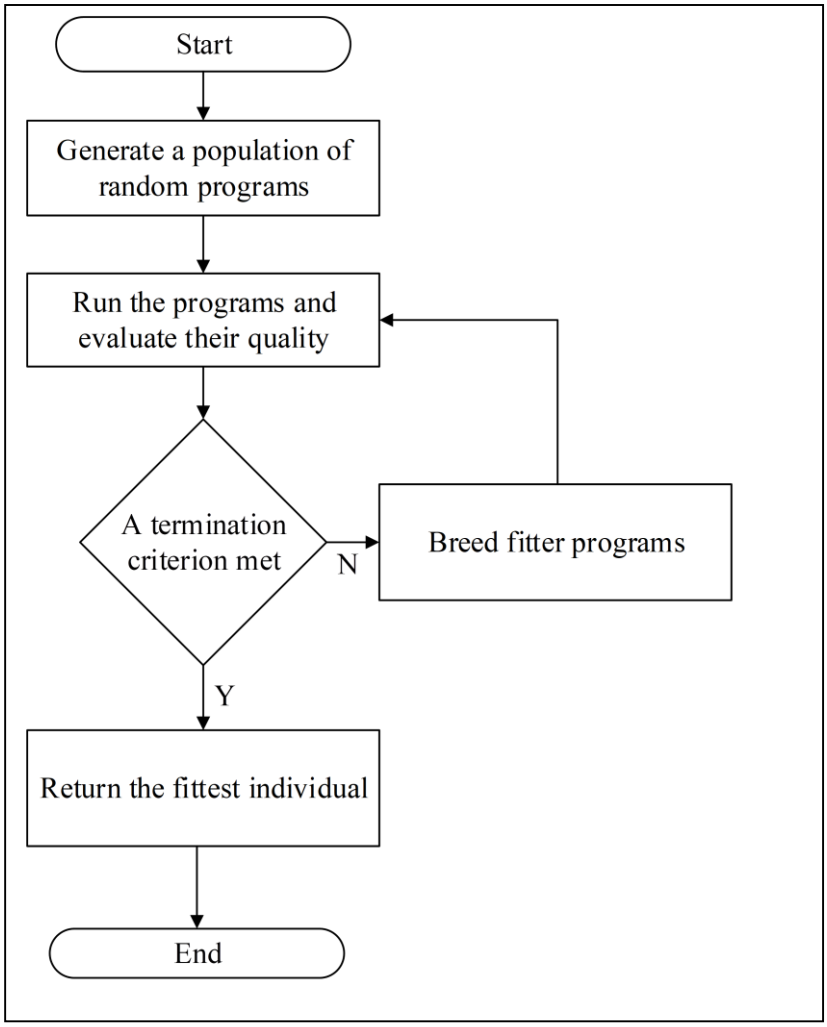


Figure 6.1. The basic control flow for the genetic programing algorithm

In general, a GP algorithm works on a population of individuals termed chromosomes. Each chromosome represents a program that when executed may produce a potential solution to the given problem [110]. Koza [45] determined that in order to solve the problem the following must be specified:

- **Terminal set:** A set of input variables or constants. These are represented by the leaf nodes of the parse tree.
- **Function set:** A set of domain specific functions used in conjunction with the terminals to construct a potential solution to the problem. These are represented by internal nodes of the parse tree.

- **Fitness function:** This function assigns a derived numerical value, termed fitness, to each individual of a population based on how close it comes to solving a given problem.
- **Algorithm parameters:** These values are problem specific and include parameters such as population size, maximum tree depth, initial population generation method, selection method, genetic operator application rate and number of generations in a run.
- **Termination Criteria:** Pre-specified maximum number of generations to be run or some additional problem-specific success predicate that has been satisfied.

6.3 Representation

Before GP can be implemented the problem that is to be solved must be clearly defined. Genetic programming typically uses parse trees to represent programs. These programs are also referred to as chromosomes [45]. These programs are made up of domain specific syntax or meaningful arrangements of information that are acted upon by genetic operators to produce offspring. The syntax of these chromosomes is made up of elements selected from the terminal and function sets, illustrated in Figure 6.2. The programs represented by these chromosomes are executed directly using an interpreter, discussed in section 6.6.

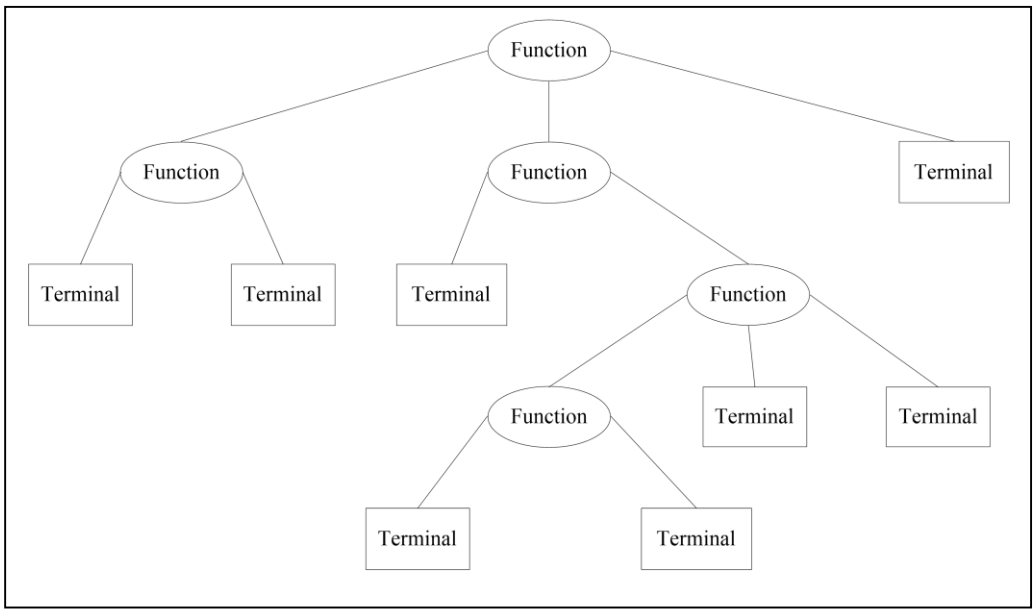


Figure 6.2. A parse tree or chromosome evolved through GP

6.4 Terminal and Function Set

The terminal and function sets make up the components of the program. Each node in the parse tree is an element of either a terminal set $T = \{t_1, t_2, t_3, t_4 \ldots t_n\}$ or function set $F = \{f_1, f_2, f_3, f_4 \ldots f_m\}$ where n and m are the numbers of terminals and functions respectively. Elements of the terminal and function sets are termed primitives [45, 110].

6.4.1 Terminal Set

The terminal set is the set of elements that form the leaf nodes of a parse tree. They have an arity of zero meaning they do not take arguments. Arity is the number of arguments a function or operator takes. Terminals are the input to the GP produced computer program. In turn, the output of the computer program consists of the value(s) returned by the program. Single values or an ephemeral random constant (a set of fixed random constants that are generated) may be used as an element of the terminal set. If the ephemeral constant is chosen as a terminal value, it is replaced by a value randomly chosen in that specified range.

6.4.2 Function Set

Elements of the function set are problem dependent and have an arity greater than zero. They can include mathematical operators, logical operators and program defined operators.
Examples are:

- Arithmetic Functions: +, -, ×, ÷.
- Conditional operators: if, else.
- Loop statements: While … Do, Repeat … Until, For … DO.
- Assignment functions: Assigns a value to a variable.
- User defined operator that combine programming statements and terminals. These special functions are termed “Blocks” [45]: Examples are PROG2 accepts two arguments, i.e. arity of two. PROG3 has an arity of three etc.

6.5 Creating the Initial Population

As with other evolutionary algorithms, for GP the individuals in the initial population are randomly generated. Individuals are made up of nodes randomly selected from either the terminal or the function sets. The root node is always chosen from the function set [45]. A GP parameter

specifying tree size must be set. This value specifies either the maximum depth of the parse tree or the number of nodes making up that tree. The three most commonly used methods for generating the initial population are the *full* and *grow* methods and a widely used combination of the two known as the *ramped half-and-half* method [45, 110]. These three methods are discussed in sections 6.5.1, 6.5.2 and 6.5.3.

6.5.1 Grow Method

Starting from the root and moving down to a random depth of no deeper than a predefined depth, each node is randomly chosen from either the terminal or function sets.

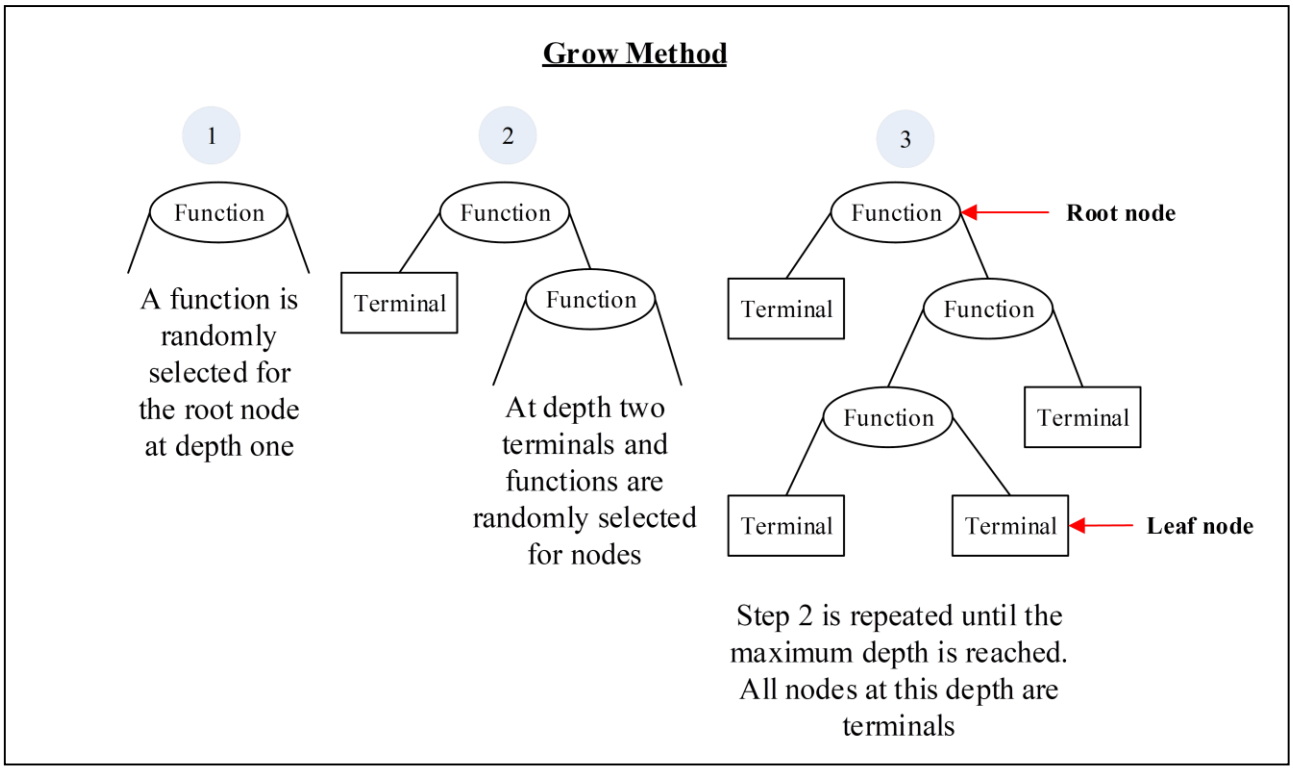


Figure 6.3. A parse tree created using the grow method

The leaf nodes are always selected from the terminal set as illustrated in Figure 6.3. This method produces trees of different shapes and sizes and provides a greater diversity in the population. Greater diversity in the initial population facilitates GP in converging to a solution sooner [45, 110].

6.5.2 Full Method

Creating a parse tree using the full method starts from the root and moves down to predefined depth. Each node above this depth is randomly chosen from the function set. Leaf nodes terminate the tree at this depth and are randomly chosen from the terminal set. The full method is depicted in Figure 6.4. This method produces trees of the same size even though the number of nodes in each tree may not be equal.

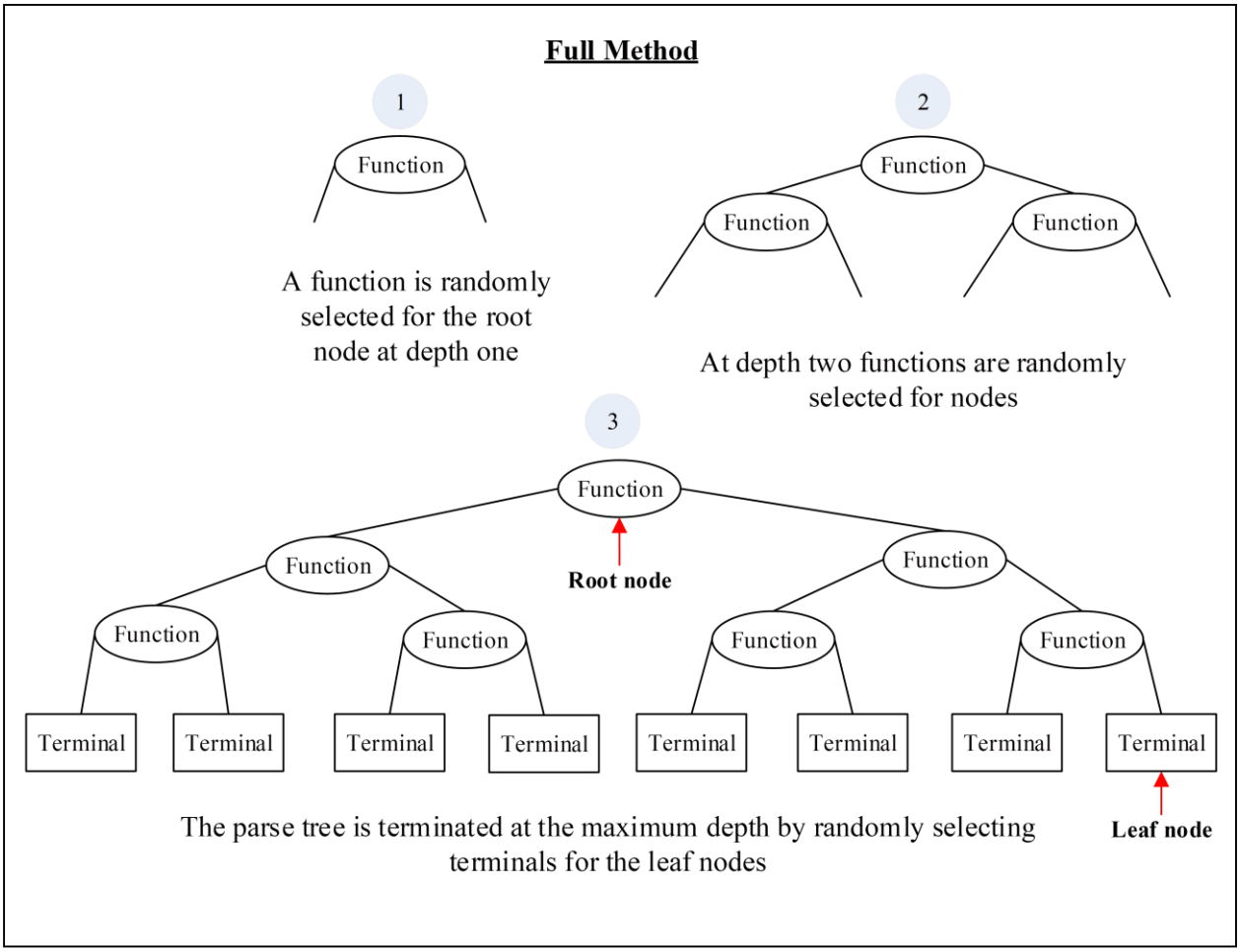


Figure 6.4. A parse tree created using the full method

Diversity in a population created using the full method is usually less compared to populations created using the grow method. This may lead to the GP algorithm converging prematurely [45, 111].

6.5.3 Ramped Half-and-Half Method

In this method, illustrated in Figure 6.5, half the initial population is created using the grow method and the other half using the full method. At each depth, starting below the root node to the specified maximum depth, an equal number of individuals of that depth are created using each method. For example, if the maximum depth is 4 then 33.3% of the trees will have a depth of 2, 33.3% will have a depth of 3, and 33.3% will have a depth of 4.

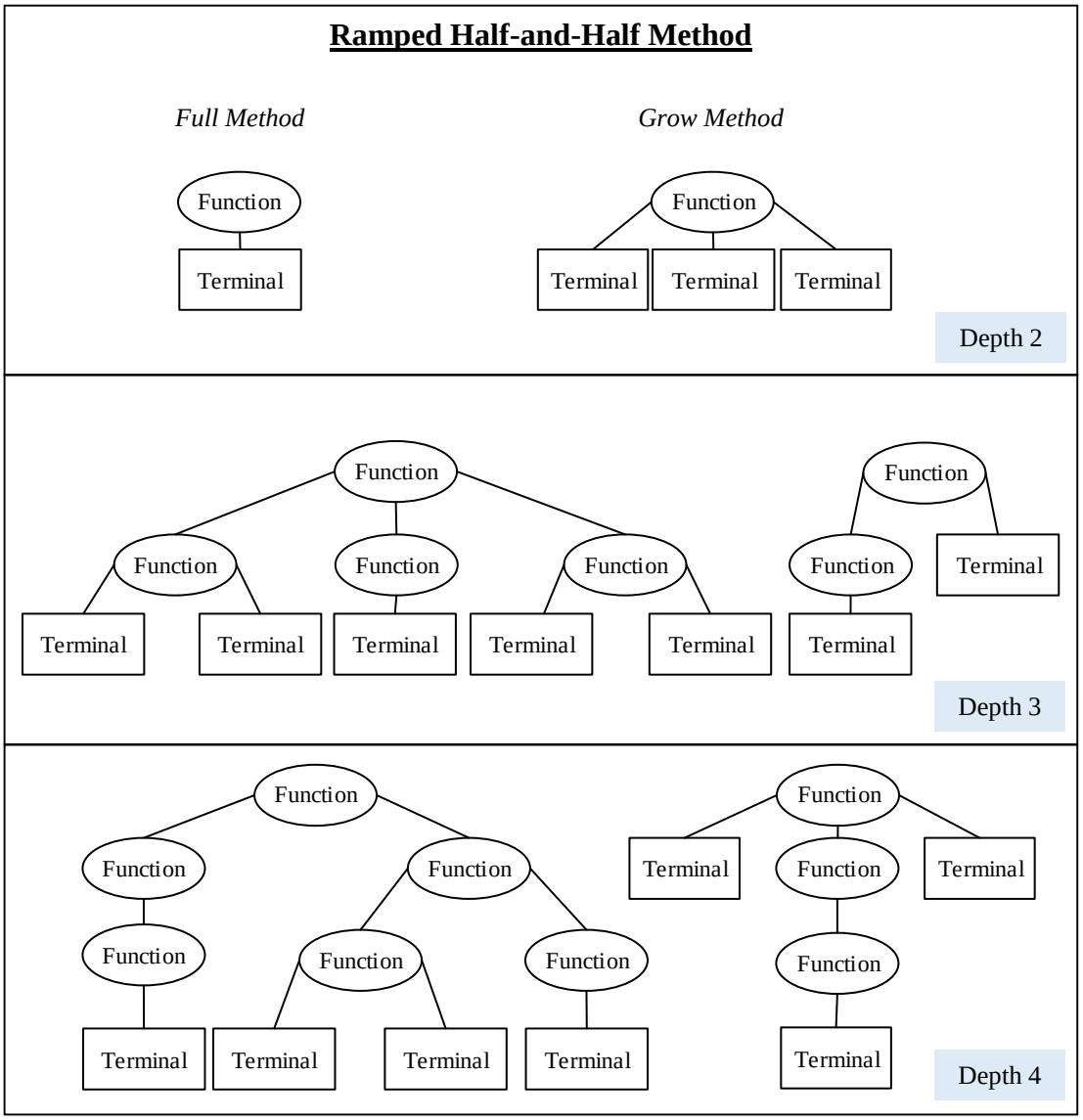


Figure 6.5. Parse trees created using the ramped half-and-half method

Consider a population size of 6 and a maximum depth of 4. Half the population is created using the grow method and the other half using the full method. Thus a total of 2 individuals are created at each depth, one using the grow method and one using the full method. As a result, this method produces a wide variety of trees of various shapes and sizes [110]. The major advantage of this method over the other two methods is that it promotes diversity in the initial population.

6.6 Fitness Evaluation

On each generation, all individuals in the population need to be evaluated by providing a measure of how able they are in solving the problem at hand. Each GP individual is a candidate program and is made up of genes and is referred to as a chromosome. Each gene may represent a tiny part of the program the GP is trying to evolve. A measure of how able an individual is in solving the problem at hand is termed *fitness*. A problem specific fitness value is termed *raw fitness* and in this case a greater value depicts a better individual. The fitness in GP is usually a scalar value. The fitness value is calculated by applying the program to a set of *fitness cases.* Even though the fitness of individual programs will range from low to high, the potential to solve parts or the entire problem will be present in the branches of these individual programs. A *fitness function* is used to determine the fitness of a GP individual, i.e. the performance of the program based on the fitness cases [64]. This is achieved by running the program through an *interpreter*. A fitness case is problem dependent and comprises an input or a set of input variables and their corresponding output values that describe the target behaviour of the required program [45]. Fitness cases must represent a sufficient amount of the problem domain since GP will evolve solutions which meet as many of these fitness cases as possible. Various objectives can be tested by the fitness function [112]. These include the correctness of a program, a minimization of the runtime of the programs and a minimization of the program size. A fitness function can be a measure of multiple criteria defined for the GP algorithm [46] and each individual is evaluated by means of this measure. The fitness function plays an important role in guiding the GP algorithm towards the global optimal solution and therefore needs to be correctly defined.

There are four common fitness functions, each to calculate one of the following, *raw fitness*, *standardized fitness*, *adjusted fitness* and *normalized fitness*. Standardized fitness is calculated from the raw fitness value such that a lower value is better. Adjusted fitness is calculated by

converting standard fitness to a value ranging from 0 to 1. Normalized fitness is represented by a ratio of the individual's adjusted fitness and the sum of the adjusted fitness of all the individuals in a population.

6.7 Selection

Having applied the fitness function and established the fitness of all the GP individuals of the initial population, the evolutionary process may begin. Selection methods are used to select individuals, termed *parents*, which are used to create new individuals, termed *offspring*. Individuals with good fitness have a better chance of being selected as parents to create offspring for the next generation [113]. Likewise, individuals with poor fitness have a less chance of being selected as parents. Koza [45] describes numerous selection methods that are used in GP, with *tournament selection* being the most commonly used [46, 110]. This selection method as applied in this dissertation is discussed below.

In tournament selection, a pre-defined number of individuals are randomly selected from the population, usually 2 to 10 [45, 46]. This pre-defined number is termed *tournament size*. Based on their fitness values, the individual with the best fitness is chosen to be a parent. The pseudocode for tournament selection is presented in Algorithm 6.2. Tournament selection looks at which individual is better than the others in the tournament and not how much better. Thus, tournament selection amplifies small differences in fitness in that the chosen individual is the fittest individual, but may only be marginally better than other individuals in a tournament. Tournament size is important as this has a direct influence on *selection pressure*. A small tournament size results in a smaller amount of selection pressure. The greater the tournament size the greater the amount of selection pressure. Large tournament sizes will predominantly lead to the selection of the fittest individuals and as a result could allow the GP algorithm to converge prematurely to a local optimum. A correct tournament size is one that will drive the GP algorithm towards the global optimum, namely, a solution.

Algorithm 6.2. Pseudocode for tournament selection

Tournament Selection

```
input  : tournament_size
output : GP individual
begin
  for count ← 1 to tournament_size do
      if count = 1 then
          best_individual ← randomly choose an individual from the population
      else
          random_individual ← randomly choose an individual from the population
          if random_individual > best_individual then
              best_individual ← random_individual
      end else
  end loop
  return best_individual
end
```

An alternative to tournament selection is another popular selection method termed *fitness-proportionate selection* [45, 114]. In fitness-proportionate selection, as described by Holland [114], the fitness of a particular individual is calculated proportionate to the fitness of the entire population. The advantage of tournament selection over fitness-proportionate selection is that fitness-proportionate selection requires more calculations compared to tournament selection. This is because in fitness-proportionate selection both the *normalized* and *adjusted fitness* [45] of each individual in the population need to be calculated. As a result, fitness-proportionate selection is more computationally expensive than tournament selection [45, 114].

Algorithm 6.3. Fitness-proportionate selection

1. Calculate standard fitness.
2. Calculate adjusted fitness.
3. Calculate normalized fitness.
4. Create a mating pool by applying the following to each individual in the population:
    a. Multiply the normalized fitness by the population size.
    b. Round up this value and assign it to the individual. This will determine the probability of that individual being copied into the next generation.
    c. Insert the individual into the mating pool a number of times specified in b above.
5. Randomly select an individual from the mating pool.
6. Return that individual as a parent.

In this way there is a higher chance that an individual with a better fitness will be selected, while those with poor fitness have a less chance of being selected.

6.8 Genetic Operators

The primary objective of *genetic operators* is to create new individuals from existing ones [110]. Genetic operators represent the search operations, which are used by the GP algorithm. Although there are a number of genetic operators that can be used to evolve a population, the three most commonly used are *reproduction*, *crossover* and *mutation*. These genetic operators are detailed in the sub-sections that follow.

Genetic operators are used to duplicate, combine or alter the genetic material from parent individuals to produce offspring. The idea being that in a population there are a number of individuals that are able to solve parts of a given problem, so by combining these useful parts the overall performance of the program may be contributed to [45]. Genetic operators transform the population in an attempt to drive the population towards a solution. Each genetic operator can be categorized as being a global or a local search operator. A *global search* operator allows an

evolutionary algorithm to explore different areas of the program space, termed *exploration*. A *local search* operator, on the other hand, uses *exploitation* to examine neighboring areas of the program space [45, 46] and thus promoting convergence. GP makes use of genetic operators to negotiate the program space through exploration and exploitation and it is important to carefully balance these processes. This is done via the application rates of the genetic operators. This balance directly influences the evolutionary process in such a way that a greater proportion of global search, i.e. *mutation*, will cause the GP algorithm to jump to random areas of the program space and never be able to converge towards the global optimum. Alternatively, if a large proportion of local search, i.e. *crossover* and *reproduction*, is applied, then the GP algorithm may get stuck in a local optimum, unable to explore other areas of the program space.

6.8.1.1 Reproduction

The application of the reproduction genetic operator is the simplest, whereby the chosen individual of a tournament is simply copied, unchanged, into the new population. Koza [45] recommends no more than 10% of the new population should be reproduced. The advantage of reproduction is that the good genes of an individual will be carried through to the next generation unaltered. This is especially important as the GP algorithm converges to a global optimum.

6.8.1.2 Crossover

This operator creates an offspring by combining randomly chosen parts from two selected parents. The crossover operator mimics the gene recombination process in nature. This operation requires two parents and produces two new offspring with mixed genetic material in the next generation as illustrated in Figures 6.6 and 6.7. This operator is a local search operator and is aimed at exploring the surrounding neighborhoods of the candidate solutions. In the crossover operation two parents are selected and a crossover point in each of the parents randomly chosen. Sub-trees rooted at these points are cut from the tree and these fragments are swapped between individuals thus creating two new offspring.

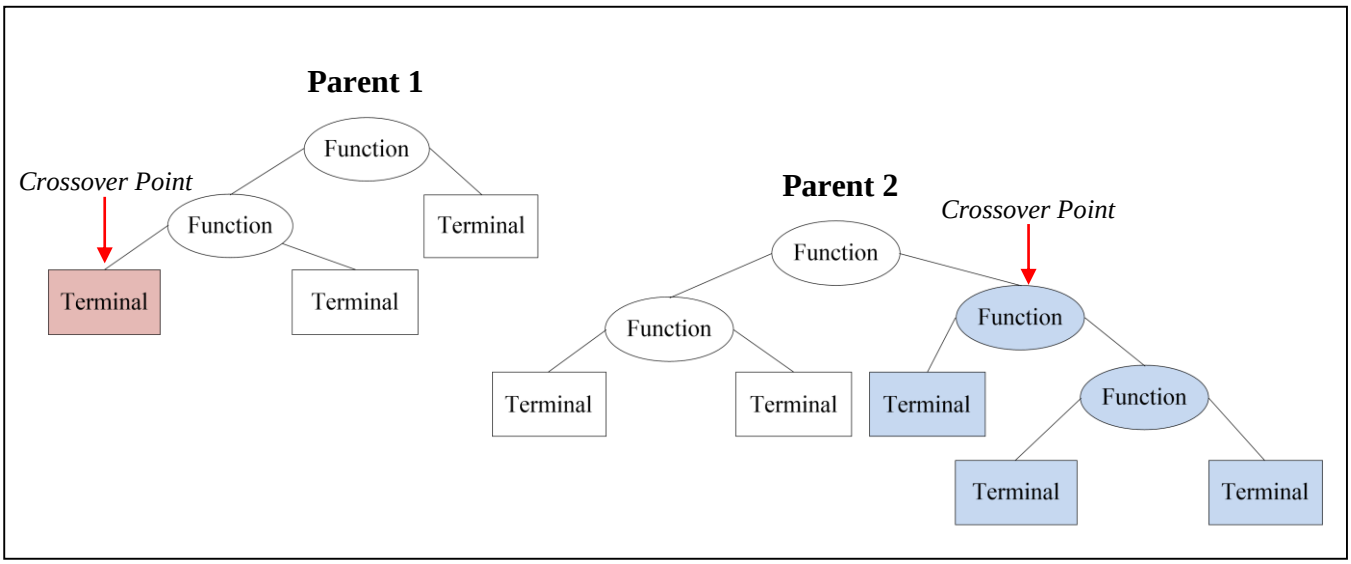


Figure 6.6. The crossover genetic operator showing two selected parents

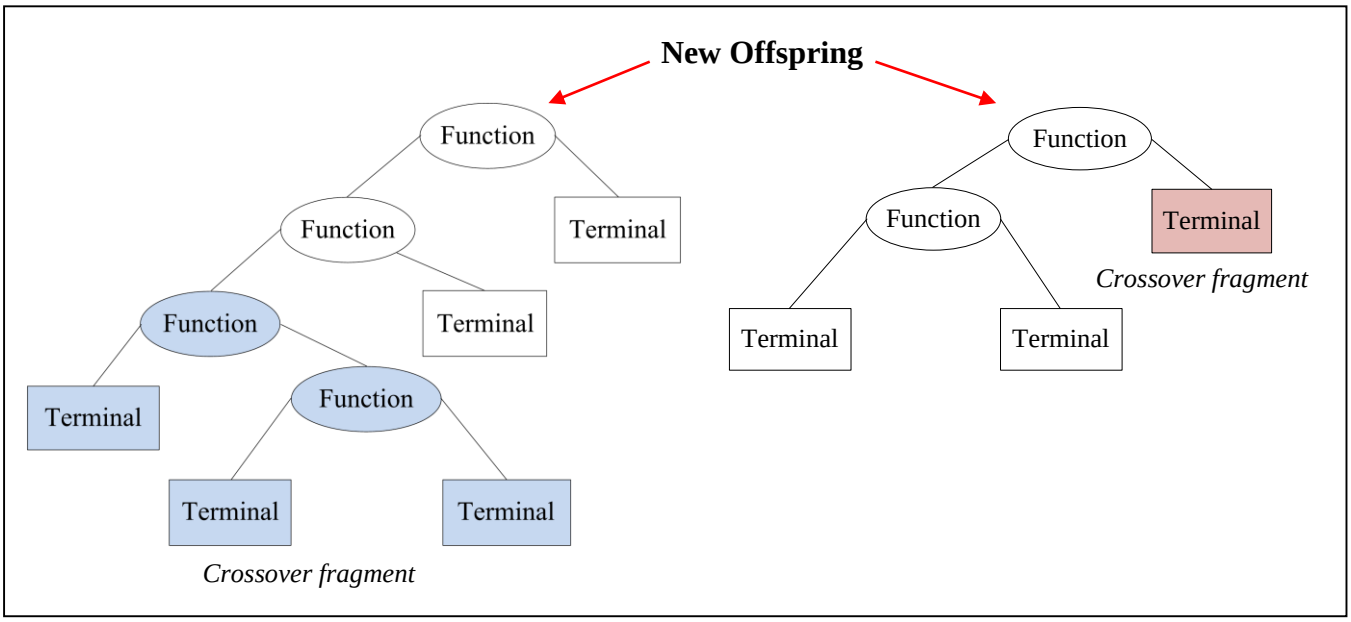


Figure 6.7. The crossover genetic operator showing the resulting offspring

With this operation, good genes may be preserved which inevitably are combined with other good genes producing better individuals than their predecessors. To control the size of the offspring created using crossover, a limit is usually placed on the tree size. The tree size is measured in terms of the tree depth or number of nodes in the tree. If the depth of an offspring

exceeds the specified maximum depth each of the function nodes at the specified maximum depth is replaced by a randomly selected terminal node. Placing a limit on the tree size reduces *bloat*. The term bloat refers to an excessive increase in the size of the tree without a change in the fitness of the individual. In certain cases redundant code, termed *introns*, may be inserted into the parse tree. Introns have no effect on the overall fitness of an individual program and do not directly affect the survivability of the GP individual. The excessive growth of introns eventually leads to bloat.

Poli *et al.* [110, 113] report that GP algorithms using high application rates of the crossover operator are susceptible to premature convergence. The crossover operator is also quite destructive [46] as it may often break up good building blocks that could form part of a solution. Banzhaf *et al*. [112] suggest implementing the crossover operator with a bias towards choosing a function node as a crossover point. The choice of the crossover point could also be biased to avoid mere swapping of the terminals. Beadle and Johnson [115] observed that preventing the crossover from merely swapping a terminal is likely to cause a bigger jump in the search space thereby limiting the probability of premature convergence.

6.8.1.3 Mutation

The mutation operator creates a new offspring by randomly altering a randomly chosen part of a selected parent. This genetic operator is a global search operator and is aimed at increasing diversity by directing the search towards different parts of the program search space [45, 46]. The most commonly used form of mutation in GP, is termed *sub-tree mutation* [110]. This operator selects a single individual for the operation and a random node is selected as the mutation point. At this point the sub-tree is removed from the tree. This is illustrated in Figure 6.8. A randomly generated sub-tree is then inserted at the mutation point. Shown in Figure 6.9. The sub-tree is created in the same way as a GP individual of the initial population was created. This operator ensures genetic diversity within a population.

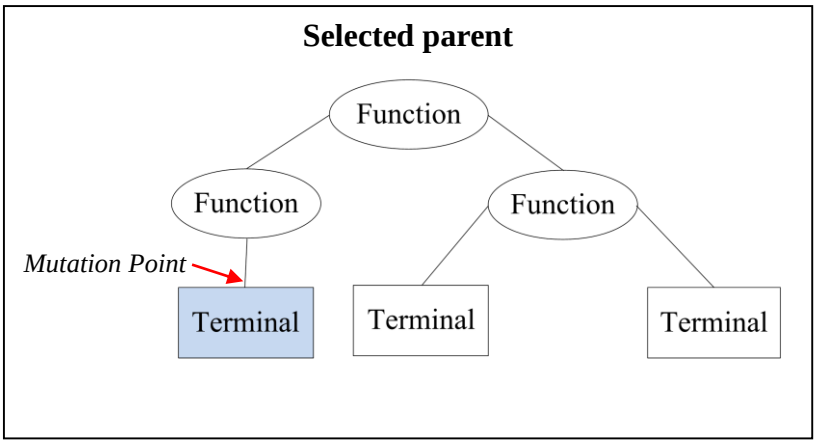


Figure 6.8. A single parent selected for mutation

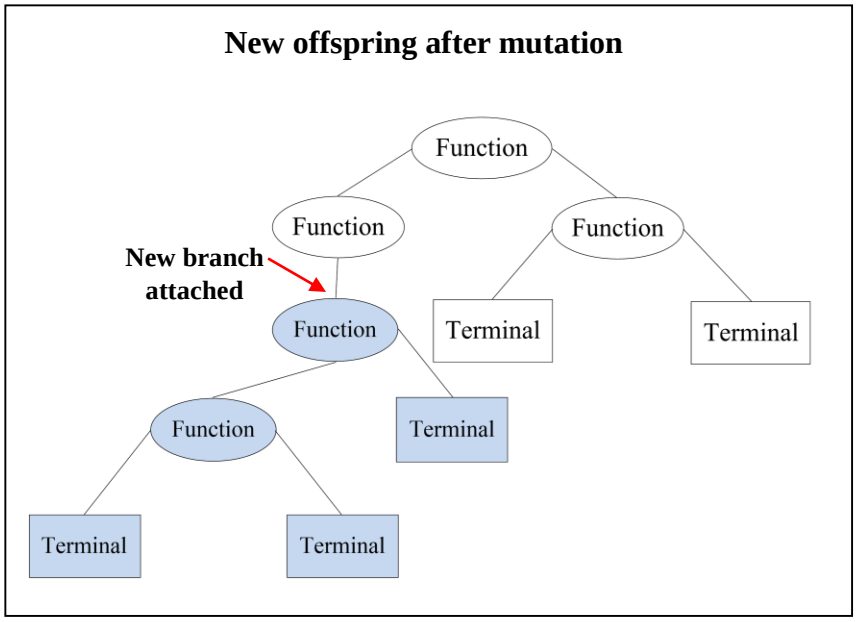


Figure 6.9. The resulting offspring after mutation

As with the crossover operator, a limit is usually placed on the size of the offspring created by the mutation operator. A high application rate of mutation may slow down convergence due to the tendency of this operator to direct the search to different areas of the search space.

6.9 GP Algorithm Parameters

The GP algorithm parameter values are problem dependent. It is thus impossible to make general recommendations for ideal value settings, as these depend entirely on the details of the problem. GP is in practice robust [110], and therefore it is possible that many different parameter values will work. Some of the important GP parameters include *population size, maximum depth, initial population generation method, selection method, genetic operator application rates* and *number of generations in a run*. Other details of the run, for example whether or not to select a terminal node for the crossover operation or an alternative *termination criterion* (section 6.10) can be included in the GP parameters.

- *Population size*: Both Poli *et al.* [110, 113] and Langdon *et a*l. [46] consider *population size* the most important parameter. Generally, the population size should be at least 500 or greater [45, 110]. There are however, GP algorithms that use much smaller populations. The main limitation on population size is computational power and often the time taken to evaluate the fitness of each individual. Langdon *et al.* suggest that as a rule one should have the largest population size that your system can handle [46, 110].
- *Maximum depth:* Depths of 4 to 6 are traditionally used for initial population generation but there are cases where depths of up to 10 have been used [116]. The initial tree sizes will depend upon the number of the functions, the number of terminals and the arities of each of the functions.
- *Selection method:* Although there are a number of selection methods available, Koza [45] mentions two in particularly, *tournament selection* and *fitness-proportionate* selection. In tournament selection, the number of individuals selected to for the tournament is usually 2 to 10 and referred to as *tournament size.* The value of this parameter dictates selection pressure (section 6.7). The advantage of tournament selection is that it requires less memory and less calculations than fitness-proportionate selection.
- *Genetic operator application rate:* Traditionally, 80 - 90% of children are created by subtree crossover [45, 110]. The application rates of the mutation and reproduction operators are usually problem specific. Reproduction inevitably occurs when an offspring is created that is identical to another individual in the population. As a result, the reproduction operator application rate is often kept low and it is not uncommon to exclude reproduction entirely [110].

- *Number of generations*: Typically, the number of generations is limited to between 10 and 50 as the most productive search is usually performed in those early generations. If the GP algorithm does not converge by then, it is unlikely that it will converge in a reasonable amount of time.

6.10 Termination and Solution Designation

A termination criterion ends a run [45]. Criteria may include a maximum number of generations to be evolved or a problem-specific success predicate. In certain cases where a perfect solution cannot be found, the GP algorithm can terminate after a certain number of generations or if a certain runtime is exceeded or a near-solution is found. An example of a success predicate, according to Koza [45], involves finding a 100% correct solution. The termination criteria of a GP algorithm are thus considered domain dependent [110]. Typically, the fittest individual is chosen and designated as the result of the run.

6.11 Chapter Summary

Genetic programming is a technique used for evolving programs and has applications in a wide-range of artificial intelligence domains. This chapter presented an overview of GP introducing the GP algorithm and described the representation of a GP program in terms of a parse tree. The parse tree is made up of terminal and function nodes. Creation of the initial population was discussed, describing the *full*, *grow* and *ramped half-and-half* methods for creating individuals of the population. The term *fitness* refers to the measure of how able an individual is in solving the problem at hand and was detailed in this chapter. Fitness is calculated using a fitness function by applying each individual of the population to a set of fitness cases. An overview of the selection process for selecting parents to produce offspring for future generations was presented. In an attempt to improve the fitness of the population, the population is evolved using various genetic operations. Three genetic operators were discussed, namely *reproduction*, *crossover* and *mutation*. The GP algorithm is controlled by parameters, the values of which are problem dependent. The parameters presented were population size, maximum depth, initial population generation method, selection method, genetic operator application rate and number of generations in a run. Finally, *termination criteria* and *solution designation* were briefly discussed. A termination criterion ends a run and the single fittest individual is designated as the result of the run. The following chapter,

Chapter 7, presents the application of GP in board game research for three different types of board games, namely, Othello, checkers and chess.

Chapter 7

Genetic Programming in Board Game Research

7.1 Introduction

Evolution through natural selection is primarily *nature's algorithm*, and as such has served as a source of many ideas for evolving AI agents for computerized systems. There is, however, no guarantee that concepts that work in nature will work in a computerized environment, but as to evolution [109] and the natural processes of selection [117] there is strong evidence that ideas taken from these processes are indeed feasible. In Chapters 3, 4 and 5 the creation of AI computer players produced through evolution and selection was presented in terms of neuroevolution and genetic algorithms. These methods were used to fine tune parameter values of the evaluation function. In this chapter, the use of genetic programming to construct the evaluation functions and evolve game playing strategies for AI computer players is discussed. In most cases, *strongly typed* GP is used so that syntactically correct programs are produced. Strongly typed GP is an enhanced version of genetic programming, which enforces data type constraints and is quite pertinent to the evolution of domain (game) specific evaluation functions [118].

The following presents the application of GP for evolving AI computer players for Othello, checkers and chess. Research into using genetic programming for game playing describes, in most studies, the use of GP to evolve board evaluation functions that are used in conjunction with game tree search methods to produce the AI computer players. The minimax search with alpha-beta pruning and the Monte-Carlo search are the two most commonly used methods for searching game trees (see figure 7.1). A game tree (Chapter 2, section 2.5.2) consists of nodes representing positions in a game and edges representing moves. These search algorithms are used to explore the game tree and are combined with evaluation function to make a move. In section 7.2, 7.3 and 7.4 studies on the application of GP for producing computer players for Othello, checkers and chess will be discussed. Each section will begin by introducing the application of GP for evolving computer players for that game type (Othello, checkers and chess) followed by a discussion of the contributing literature.

Each study is reported on using the following outline:

- Representation of the GP.
- The GP parameters.
- The evaluation function and parameters.
- Methods for determining the fitness of the evaluation function.
- Methods for determining the quality of the evolved players.
- Overall results.

7.2 The Application of GP for Evolving Computer Players for Othello

Genetic programming has been used primarily to produce board evaluation functions for Othello computer players. Eskin and Siegel [119] in 1999 presented the application of genetic programming to evolve Othello computer players as a teaching approach for students studying machine-learning techniques. The GP in this study was used to evolve *board evaluation functions*, which evaluate the configuration of the Othello board for each move of the game. A single evaluation function returns the heuristic values representing the quality of the board configurations at the leaf (terminal) nodes in the game tree.

Figure 7.1 represents an example of a game tree generated at each move during an Othello game to a depth of three. The root node represents the current board configuration. Each child node represents a possible future board configuration resulting from a move made by the parent node. The GP evolved function is applied to the leaf nodes in the game tree to produce a heuristic value for each node. These values are calculated from the resulting board configuration at that node. These values are then propagated back up the game tree, using the minimax algorithm to determine the best move to be made.

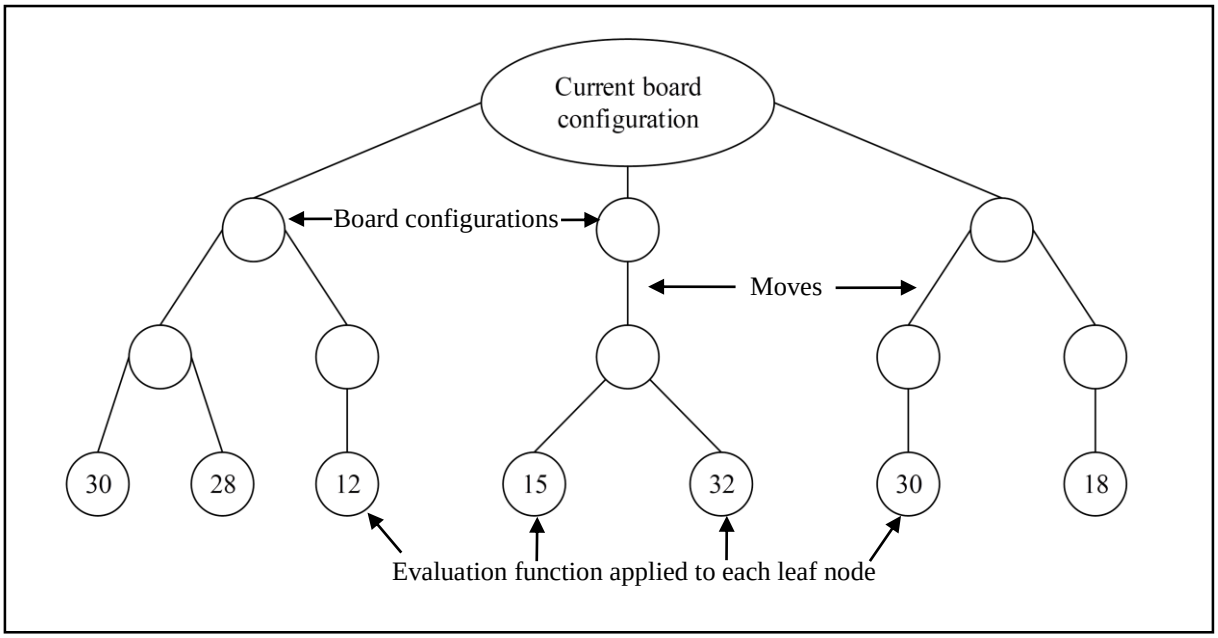


Figure 7.1. A typical game tree with each node representing a board configuration resulting from a move made from the parent node.

In this study, the GP evolved evaluation function is used in combination with the minimax search algorithm (with or without alpha-beta pruning) to determine the next move leading to the best position on the board. The evaluation function, in combination with the search algorithm, represents an Othello computer player. The GP terminal set was represented by ten terminals, each with an integer value depicting:

- Number of black discs on the board.
- Number of corners occupied by black discs.
- Number of black discs adjacent to the corners.
- Number of black discs on the edge of the board.
- Number of white discs on the board.
- Number of corners occupied by white discs.
- Number of white discs adjacent to the corners.
- Number of white discs on the edge of the board.
- An ephemeral constant value of 10.

The function set was made up of four arithmetic operators:

- Plus, minus, multiply and divide.

Figure 7.2 represents a typical parse tree representing an evaluation function evolved by GP from the above terminal and function sets.

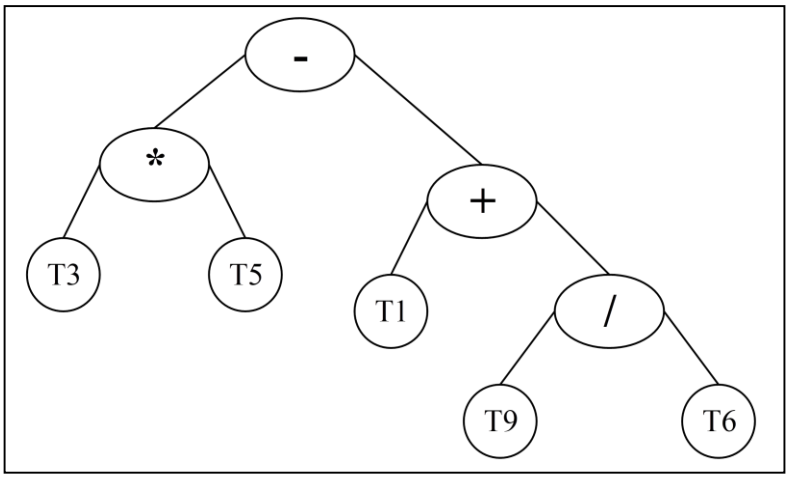


Figure 7.2. A typical evaluation function evolved by GP.

The selection method used was tournament selection with a size ranging from two to six. The standard genetic operators, *crossover, mutation and reproduction* presented by Koza [45] were used to evolve the evaluation functions. Several methods for determining the fitness of the evaluation functions during the evolutionary process were implemented. These methods involved tournament play, over a given number of games against:

- Random move players.
- Previously evolved players.
- Handcrafted players.
- Elements of the same population.
- Co-evolving population of GP players with different GP parameter values.

Several methods were used to determine the quality of GP evolved players:

- Tournament play against human players.
- Tournament play against online Othello players.
- Tournament play against other GP players.
- Tournament play against a handcrafted Othello program, *Edgar* [119].

- Games were recorded and replayed to analyze strategies produced by the computer players.

Although this study was presented as an introductory teaching guide for machine learning techniques, the application of GP in developing artificial computer players for Othello was successfully achieved. Several GP players were able to beat the handcrafted Othello program, *Edgar* [119]. The Othello-playing program, *Edgar* [119] was used in this study as a benchmark to evaluate the overall success of the students' Othello computer players.

Benbassat and Sipper [10, 120] presented the application of a strongly typed GP to evolve board evaluation functions for their Othello computer players. In this study, the minimax search algorithm with alpha-beta pruning is used in conjunction with the evolved evaluation function, representing the computer players, to determine moves. The research presented by these authors describes an approach that uses GP to evolve computer players for more than one type of board game, namely Othello, *lose-checkers*, 10x10 checkers, Nine Men's Morris and dodgem [10, 120].

The terminals in this study represented general game board states:

- The enemy pieces on the board. (Chapter 3 section 3.2.2.1 discusses the terms friendly and enemy discs).
- The number of friendly pieces on the board.
- The number of available move positions for the player. This represents the players's mobility.
- The number of available move positions for the opposing player. This represents the opponent's mobility.
- A value indicating if a position is occupied or not.
- A value indicating whether a friendly or enemy disc occupies a position.
- A value of TRUE.
- A value of FALSE.
- An ephemeral random constant (ERC) chosen at random from the range 5.0 to -5.0.

The function set was made up of several types of functions consisting of logical and arithmetic operators and domain specific functions.

Logical operators:

- or, nor, and, nand.

Arithmetic operators:

- Minus, Plus.
- MultERC. This function receives a value and returns the product of the value multiplied by a random number.

Domain specific functions:

- IfTrue. This function receives three arguments. If the first argument is TRUE it returns the second argument, otherwise it returns the third argument.
- LowerEqual. This function receives two arguments and returns TRUE if the first argument is less than or equal to the second.

Figure 7.3 depicts a parse tree representation of an evaluation function evolved by GP from the above terminal and function sets in this study.

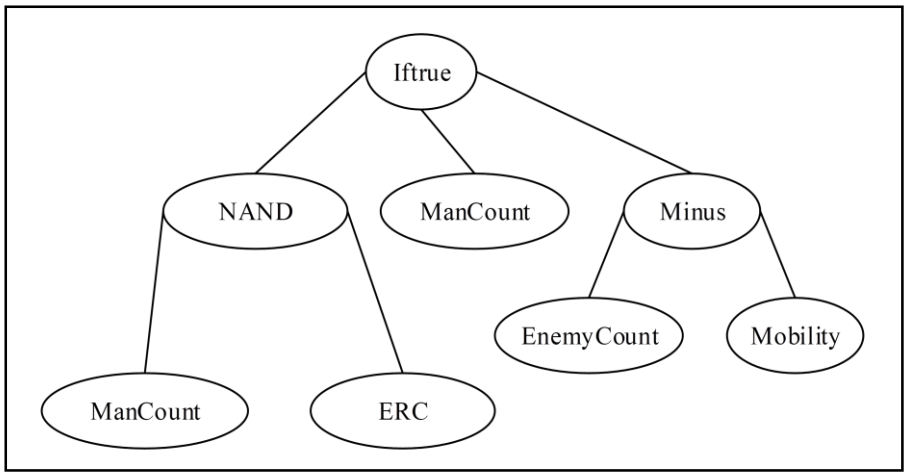


Figure 7.3. An example of a GP evolved evaluation function

Two methods were used to determine the fitness of the players during the evolutionary process. The players faced two types of opponents: external *guides* (described below) and their own counterparts in the population.

Firstly, in the *guide-play* rounds, individuals were played against two types of guide opponents. Random move players, i.e. players that chose a move at random and handcrafted players using the minimax search algorithm with alpha-beta pruning. The minimax player searched up to a preset depth in the game tree and used a handcrafted evaluation function to evaluate board states. One point was added to the fitness of the individual being evaluated for every win, and 0.5 points for every draw. Secondly, the population was then evaluated in a *co-evolution* tournament whereby each member of the population, in turn, played as black for pre-set number of games against randomly selected opponents from the population, playing white. One point was added to the fitness of the individual being evaluated for every win, and 0.5 points for every draw.

Tournament selection of size two [10, 120] was used to select parents to produce offspring for the next generation. Standard *mutation* and *crossover* genetic operators [45] and a modified version of the standard *crossover* operator termed *one-way crossover* were implemented to evolve board evaluation functions. For the modified method of crossover, part of the genome of the fitter parent is inserted into the other parent, without the donor receiving any genetic information in return. The reasoning is that individuals with higher fitness are more likely to contain better genes than the less-fit individuals and so this method of crossover is considered more likely to increase the frequency of the genes that lead to better fitness.

In this study, the guide players used in the fitness evaluation were also used to determine the performance of the evolved evaluation functions. The GP players were evaluated in matches of 10,000 games against the guide players as well as other players from co-evolved populations. The GP players scored an average of 85% wins against the random move players and 75% wins against the hand crafted minimax players.

In an attempt to improve the performance of the GP computer players the number of terminal and function elements making up the GP terminal and function sets was increased. Both basic as well as game specific terminals were introduced [121, 122, 123]

Additional basic terminals:

- A terminal with value of one.
- A terminal with value of zero.

Additional game specific terminals:

- FriendlyCornerCount. The number of corners occupied by friendly discs.
- EnemyCornerCount. The number of corners occupied by enemy discs.
- CornerCount. FriendlyCornerCount minus EnemyCornerCount.
- IsSquareEmpty. True if square is unoccupied.
- IsFriendlyPiece. True if square is occupied by a friendly disc.
- IsManPiece. True if square is occupied.
- Mobility. The number of move options available to the player.

Additional functions:

- Logical AND.
- Logical NAND.
- Logical OR.
- Logical NOR.

In an attempt to improve the speed of the game tree search, a selective search technique termed *forward pruning* was introduced to improve the speed of the minimax search. Two approaches were implemented for this selective search method. The first relied on a parameter value to determine the ratio of sibling states that are further expanded for search versus those that are pruned from the tree. The second approach chose *r* number of nodes, randomly selected in the tree, to be expanded for further search. The other nodes were simply ignored. In both cases the selected nodes are evaluated and sorted according to their heuristic values. Those nodes with the highest heuristic values are then expanded for further search. Sibling nodes with low heuristic values were cut from the game tree. This technique resulted in an increase in playing speed allowing for deeper searches.

To test the quality of their GP players, Benbassat and Sipper [121, 120] created handcrafted players that used the minimax algorithm with alpha-beta pruning to search the game tree to depths of 5 and 7. In these experiments the performance of the improved GP players was recorded to be far superior to that of the handcrafted players. GP players searching to a depth of 4 were able to beat the handcrafted players searching to a depth of 7 [121].

Extending their approach to evolving GP players for zero-sum, deterministic board games, Benbassat and Sipper [122, 123] replaced the minimax search algorithm and forward pruning method with the Monte-Carlo tree search (MCTS) algorithm (Chapter 2, section 2.4). The same experiments, as described above, were carried out using the MCTS algorithm for both the GP players as well as the guide players. All GP parameters and experimental setups were identical to above discussed experiments. The GP players were also evolved and evaluated in a similar way. These evolved GP players outperformed the handcrafted Othello computer players looking three and five moves ahead using the alpha-beta search algorithm. In a 1000 game tournament against the handcrafted players looking three moves ahead, the GP computer players won 94% of the games and won 88% of the games against the handcrafted players looking five moves ahead.

7.2.1 Summary of the use of GP in Othello board game research

The above studies on the use of GP for evolving AI computer players for Othello shown that GP is primarily used to evolve board evaluation functions. These evaluation functions are used in conjunction with game tree search algorithms, such as the minimax search with alpha-beta pruning and the Monte-Carlo search algorithm, to determine which moves are to be made. In addition to the standard minimax search method, a second method, termed *forward pruning* [121, 63], was implemented to improve the search speed of the game tree search. The combination of the evaluation function and the search algorithm constitutes a GP player.

In addition to the standard GP operators [45] used to evolve the GP players, a modified version of the crossover operator, termed *one-way crossover* [10, 120] was introduced. This genetic operator was implemented to improve the chances of good quality genetic material being carried through from one generation to the next. A number of methods were used to determine the fitness of the evaluation functions during the evolutionary process. These all involved tournament rounds between the evolved GP players and various evaluation guide players. These guide players included random move players, handcrafted evaluation functions combined with game tree search algorithms and tournament rounds between elements of the same or a co-evolving population.

The quality of the evolved GP players was assessed through tournament rounds involving different types of opponents. These opponents included human players, handcrafted Othello

players, online players and tournaments against other GP evolved players using different evaluation functions. These studies demonstrated that good quality Othello players can be produced with GP. The playing ability of these GP computer players can be further improved upon through modified genetic operators, improved search functionality and game specific board evaluation techniques such as mobility.

7.3 The Application of GP for Producing Computer Players for Checkers

In a similar way to that of Othello GP is used to evolve board evaluation functions for checkers. In addition, GP is also used for developing artificial ANNs using Cartesian Genetic Programming (CGP) (discussed below) to create checkers computer players. Benbassat and Sipper [124] presented their research on evolving computer players for checkers using GP on a variation of the standard 10x10 version of checkers [125] called *lose-checkers*. The basic rules of *lose-checkers* are the same as American checkers rules except that a player wins by losing all of its own pieces or is left with no legal moves.

This approach to evolving computer players for *lose-checkers* is very similar to the approach that was used for evolving computer players for Othello (discussed in section 7.2). The difference in this approach is that the Othello game specific GP terminals were replaced with checkers specific terminals:

- The number of friendly Kings minus the number of enemy Kings, namely the King count.
- A value representing a King piece (Chapter 4, section 4.2).
- A value representing a normal checkers piece.

Forward pruning and the implementation of the Monte-Carlo tree search method was not implemented for the *lose-checkers* experiments as in the Othello experiments. The quality of the GP evolved strategies were assessed through tournament play of a 1000 games against players that selected legal move positions at random to make a move, termed random move players, and handcrafted *lose-checkers* players. In these experiments, the GP evolved computer players performed better than the random move players and often better than the handcrafted checkers

players. Several of the GP players searching to a depth of three were able to beat the handcrafted players searching to a depth of six.

Kahn *et al*. [126] presented a hybrid system of Cartesian genetic programming (CGP) and ANNs. The term Cartesian is used as the genetic programs are represented as graphs using a two-dimensional grid of nodes [127]. In this study, neural networks were not evolved directly but CGP was used to build the ANNs. The ANNs were trained to play checkers and instead of having a predetermined structure, CGP was used to develop the structure of the ANNs in terms of the number of dendrites (inputs), axons (outputs) and neurons [128]. Encoded into the genome of the genetic programs were the instructions used to develop the ANN. This was achieved by evolving genetically encoded programs that when run, built the ANNs that were able to learn through interaction with the environment. The ANNs were used in conjunction with the minimax search algorithm to produce checkers players. The initial population comprised of five randomly generated chromosomes, which were evolved over 1000 generations producing five individually developed ANNs. Offspring for each generation of CGPs were created using the *reproduction* and *mutation* genetic operators [45].

Four separate experiments were conducted. The fitness of each of the five CGP produced chromosomes, at the end of each generation was determined as such. Firstly, five ANNs were produced by running the encoded CGP programs. For the first two experiments, the ANNs were evaluated through tournament play of 5 (experiment 1) and 10 games (experiment 2) respectively against a handcrafted evaluation function that was combined with the minimax search algorithm. For each of the experiments, at the end of each generation, the winning chromosome was selected and used as the parent to produce five offspring for the next generation. The selected chromosome was copied unaltered into the next generation. A further four offspring were produced by mutating the parent chromosome.

For experiments 3 and 4, a similar setup as in experiments 1 and 2 was used, except the CGPs were co-evolved instead of through tournament play against the handcrafted evaluation function. For each experiment, two similar CGP *systems* were implemented. Each starting with an initial population of five randomly generated chromosomes and evolved over 1000 generations. On each

generation the five CGPs from each system were evaluated by playing 5 (experiment 3) and 10 (experiment 4) games against the winning CGP of the previous generation of the alternate CGP system. As this evaluation method could not be used to evaluate the CGPs of the first generation, a randomly selected CGP was used as a parent. The five CGPs of the second generation of the alternate CGP system were evaluated against the randomly selected CGP of the first generation. After 1000 generations the winning GPC was selected to produce the ANN to be used as the evaluation function in the checkers playing program.

In all experiments, inputs to the ANNs were position occupation and the fitness of a CGP, over the five and 10 games, was calculated as value of pieces that remained on the board after a game:

> 1000 + (friendly Kings * 200 + friendly men * 100 + number of moves made) – (enemy Kings *200 + enemy men * 100).

For this study, the quality of the CGP checkers playing program was not determined through tournament play against any other checkers playing program but through tournament rounds between pre-evolved ANNs of the four experiments. This was achieved by selecting the best performing ANNs of the co-evolution experiments, at each 50 generation interval, and pitting them against the best performing ANNs of the minimax experiments, at similar 50 generation intervals in five game tournaments. The playing quality of the CGP produced ANNs evolved in experiment two and four (ten game tournaments) was far superior to those evolved in experiments one and three (five game tournaments). Further, the ANN produced through the co-evolved CGPs for both experiments, three and four, outperformed those ANN that were evolved using the handcrafted minimax search checkers program.

7.3.1 Summary of the use of GP in checkers board game research

The application of GP to evolve computer players for checkers playing computer players has been successfully demonstrated in two differing approaches. In the first approach, GP is used to evolve board state evaluation functions for a variation of the 10x10 checkers game know as *lose-checkers*. These evaluation functions are used in conjunction with the minimax game tree search algorithm to determine which moves are to be made. In a second approach, ANNs, used as board state evaluation functions, were not evolved directly but were produced by evolving genetically

encoded programs that when run, built the ANNs. These ANNs were used in conjunction with the minimax search algorithm to produce checkers playing programs.

7.4 The Application of GP for Producing Computer Players for Chess

The application of GP for evolving chess strategies for computer chess players includes its use to improve the alpha-beta search algorithm, evolve board evaluation functions in order to produce general purpose endgames strategies for different chess endgames and evolve pattern-based chess strategies for the classic King-Rook-King (*KRK*) chess endgame. A chess endgame is the stage of the game when only few pieces (12 or less) are left on the board [129].

Gross *et al*. [130] applied a hybrid GP and GA (Genetic Algorithm) in order to improve the evaluation function of their chess playing program and the search speed of the minimax search algorithm in an attempt to produce good middle game strategies for chess. No human expertise or chess strategy databases were used to guide the computer players in this study. The hybrid system consisted of three algorithms, two GAs and a GP algorithm. The first GA was used to determine the order in which the game tree nodes are to be expanded and the second to evolve the evaluation function that evaluated the given chess positions at the leaf nodes. The GP algorithm was use to evolve the function that determined to what depth each node (possible move) should be expanded. This was done in an attempt to improve the search speed of the minimax algorithm. Normally the depth of search for a game tree (Chapter 2, section 2.3) is a preset parameter value [104, 21]. In this way, at a certain depth, determined by the GP, the search is terminated.

The GP terminal set consisted of real number values that store the current configuration of the game tree in terms of:

- The current depth of game tree.
- The search depth limit of the alpha-beta algorithm.
- The current horizontal search position of the game tree.
- The number of nodes that can be expanded from the current move.
- The value of a chess piece at the node to be expanded.
    - Pawn = 1 point.

  - Bishop and Knight = 3 points.
  - Rook = 5 points.
  - Queen = 9 points.
- The position of next valid move.
- The value of the current move determined by the evaluation function.
- The total number of pieces on the board.

The GP function set consisted of:

- +, -, *, /.
- IF than ELSE.
- INC. Increments the search depth by one level.
- The sine function.
- The sigmoid function, $1/(1+e^{-\text{terminal value}})$.

A population of *p* individuals were evolved over *g* generations and the fitness of each individual, at the end of a generation, was evaluated though tournament play against a minimax search chess playing program set to a fixed search depth. The fitness of each individual was calculated as the number of wins, losses and draws against a simple chess playing program. For the evolution of the GP algorithm mutation [45] and two types of crossover were implemented. The first crossover operator exchanged genetic material between two parent individuals. In the second crossover operator, only one parent was used and two branches of the same parse tree were exchanged.

The quality of the computer chess players was assessed through tournament play of 20 games against the minimax search chess program set to levels of difficulty, 8, 10 and 12. Twelve being the hardest level. A total of sixty games were played between the evolved players and the minimax search chess playing program. The evolved computer chess player outperformed the minimax search program searching to an average depth of 40 - 50% less than the opponent's search depths. At a search depth of 5, the minimax search program needed to expand approximately 890 000 nodes per move, whereas the evolved chess players expanded an average of 120 000 nodes per

move. This in turn translated into the evolved chess players using only 6% of computing resources compared to the minimax search chess playing program.

Lassabe *et al*. [91] investigated the King-Rook-King (*KRK*) endgame in their research using GP to evolve pattern-based chess strategies. The GP terminal set was composed of thirteen Boolean values representing board configurations and eight Boolean values representing specific moves.

Board configuration terminals:

- Are the Kings opposite each other, separated by one square?
- Are the Kings almost opposite each other?
- Is the black King on the edge of the board?
- Is the only one move left for Black to play?
- Are the Kings separated by three squares?
- Can Black move?
- Can the white King block the black King?
- Can the white Rook check Black?
- Is the white Rook between the two Kings?
- Can the black King capture the white Rook?
- Is the white Rook protected?
- Can the white King move to protect the white Rook?
- Can the white King move between the white Rook and black King?

Move terminals:

- Move the white King to protect the Rook.
- Move the white King towards black King.
- Move the white King to oppose black King.
- Move the white Rook to limit black's moves.
- Move the white Rook away from the black King.
- Move the white Rook between the two Kings.

- Move the white Rook to check the black King.
- Move the white Rook to avoid stalemate.

The GP function set consisted of two functions:

- IF than ELSE.
- AND.

A population of 10 000 individuals were evolved over 300 generations. Offspring for each generation were created using the *mutation, crossover* and *reproduction* genetic operators. The fitness of an individual was determined through tournament play of 17 games each against a handcrafted program implementing 17 different *KRK* endgame board configurations (one configuration per game). Individuals of the population were exposed to the same 17 endgame board configurations. The fifty-move rule applied to games ending in a draw [92]. The fitness of an individual was calculated as the sum of bonus and penalty points achieved over the 17 games, namely:

- -5 points if Rook was taken.
- -3 points if a non-legal move was made.
- -2 points if a draw occurred through repetitive moves (50 move rule).
- -1 points if the game exceeded 50 moves.
- + 200 points if the game ended in a stalemate.
- +400 points for a win.

Finally, the GP chess players were evaluated in two experiments. In the first experiment the GP players were matched against the chess playing program, *CRAFTY*, in a 17 game *KRK* tournament. Each game consisted of a different endgame *KRK* board configuration. In the second experiment, the GP players played against a chess playing program that used a chess endgame database known as Nalimov's tables [131]. The same 17 *KRK* endgame configurations as in experiment 1 were used. Although the chess strategies evolved by GP evolved are specific to the *KRK* endgame, GP was able to evolve good endgame strategies. In both experiments, the optimal number of moves were achieved to checkmate the enemy King, i.e. eleven moves on average, and in a time of less than a second per move.

Hauptman and Sipper [19] applied GP to evolve board evaluation functions, used in conjunction with the minimax search algorithm, to produce general purpose endgames strategies for different chess endgames. The endgames that were investigated were the *KQRKQR*, *KQKQ*, *KRKR* and *KRRKRR* (White - King Queen Rook and Black - King Queen Rook) chess endgames. The board configurations resulting from a one move look ahead for all possible moves from the parent board configuration were evaluated and a real-value score returned indicating the quality of that move. The move resulting in the highest score was then used to make the game move.

The GP terminal set consisted of Boolean and real values and was divided into simple and complex terminals.

Simple terminals:

- Is the enemy King in check?
- Is the friendly Queen under attack?
- Is the enemy Queen under attack?
- Is the friendly piece creating a fork (two alternative positions to move to)?
- Is an enemy piece creating a fork?
- Are two or more friendly pieces protecting each other?
- Are two or more of the enemy pieces protecting each other?
- Is the enemy King protecting a piece?
- Is the friendly King protecting a piece?
- Distance of friendly King from the edge of the board.
- Distance of enemy King from the edge of the board.
- Number of friendly pieces that are not under attack.
- Number of enemy pieces that are under attack.
- The sum value of friendly pieces that are attacking (the values of chess pieces are mentioned in the Gross *et al*. study).
- The sum value of enemy pieces that are under attack.
- Number of legal moves available for the friendly King.
- Number of illegal moves for the enemy King.

- Distance between friendly King and Rook(s).
- Distance between enemy King and Rook(s).

Complex terminals:

- Did the player capture a piece?
- Is this a checkmate position?
- Can the player checkmate the enemy king after this move?
- Is it safe to capture the enemy piece?
- Is it safe to capture a friendly piece?
- Do the enemy King's legal moves bring it closer to the board edge?
- Do the friendly King's legal moves move it further from the board edge?
- Is the enemy King two or more positions behind a friendly piece?
- Is the friendly King two or more pieces behind an enemy piece?
- Is one or more enemy pieces pinned (cannot move without being taken)?
- Are all friendly pieces free to move?
- The sum value of piece on the board.

The function set consisted of:

- IF than ELSE.
- IF < THAN ELSE.
- OR, AND, NOT.

During the evolutionary process, the fitness of individuals was determined through tournament play. Each element in the population was selected to play against five randomly chosen elements of the population over a preset number of endgames starting with random board configurations. Standard genetic operators, namely, mutation, reproduction and crossover, were used to evolve the populations [45]. Fitness was calculated by the sum points accumulated over the five games. One point for a win, 0.5 for a draw and 0.75 for the sum of friendly pieces minus the sum of enemy pieces being greater than that of the opponents. The final fitness for each player was the sum of all

points earned throughout the entire tournament for that generation divided by the number of games played.

Three experiments were conducted to assess the quality of the GP evolved strategies. For the first experiment the best performing individual after 10 generations during the evolution process, was selected and tested against a handcrafted computer evaluation function. One hundred and fifty games were played, fifty games for each of the *KRKR*, *KRRKRR* and *KQRKQR* (White - King Queen Rook and Black - King Queen Rook) endgames. The quality of the GP evolving chess players improved rapidly after about 50 generations and stabilized at about 120 generations. An average of 6% games were won and 68% played to a draw. In a second experiment, the hand crafted evaluation function was replaced with the chess playing program *CRAFTY* (Chapter 5, section 5.2.2). An average of 2% games were won and 72 % played to a draw. For the third experiment, the evolved GP players were tested against *CRAFTY* in a 150-match tournament. Fifty games were played for each of the three endgame configurations. *CRAFTY* proved a formidable opponent, checkmating the GP players in nearly all of the games. However, there were occasions when the GP players were able to achieve a draw against *CRAFTY*.

7.4.1 Summary of the use of GP in chess board game research

The application of GP in chess to produce quality chess playing programs that play well in middle as well as endgames has been successfully demonstrated in the studies discussed above. Genetic programming was used in a hybridized system with genetic algorithms to evolve evaluation functions and to improve the game-tree search. This GP application was implemented specifically for chess middle games with no human expertise or chess databases to guide the computer players decision making. The application of GP resulted in a reduction in search computation, improving the game-tree search speed by 94%, thus allowing deeper searches in less time to make a move. The complex *KRK* endgame was highlighted in a study that used GP to evolve pattern-based chess strategies that use specific patterns and chess moves proposed by a chess expert. In this approach 17 *KRK* configurations were tested resulting in the GP players achieving the optimal number of moves, i.e. eleven, to checkmate the enemy King and in a time of less than a second per move. In a further study, GP was used to evolve the evaluation functions for general purpose endgames strategies for different chess endgames. The evolved evaluation

functions were tested against a handcrafted evaluation function and a world champion level chess playing program. Genetic programming evolved players performed moderately well against the handcrafted evaluation function, achieving an average 74% draws. The world champion level chess playing program proved to be quite formidable, beating the GP computer players in most games but occasionally the GP computer players were able to play to a draw.

7.5 Chapter Summary

Studies on game playing using GP have shown that GP, to varying degrees of success was able to produce computer players that could perform as well as, if not better than, handcrafted computer players. In this chapter the use of GP for evolving computer players for Othello, checkers and chess was presented.

For Othello, GP is used primarily to evolve evaluation functions that are used in conjunction with the minimax and Monte-Carlo game tree search algorithms to produce competent Othello computer players. A similar application of GP for evolving evaluation functions for checkers incorporating the minimax game tree search algorithms was also presented. In a further study, instructions encoded in the genome of a Cartesian genetic program was used to develop the structure of a checkers playing ANN. In chess, GP has successfully been used in a hybridized system with genetic algorithms evolving the evaluation functions and the implementation of GP to improve the speed of the game-tree search. In a further study, the classic KRK chess endgame configuration was used to determine the quality of the GP evolved game playing strategies. GP was used to evolve pattern-based chess strategies that use specific patterns and chess moves proposed by a chess expert for its computer players. A more generalized approach for evolving game playing strategies for different chess endgames was presented where GP was used to evolve board evaluation functions for the KQRKQR, KQKQ, KRKR and KRRKRR chess endgames. A combination of simple and complex GP terminals and non-game specific functions produced game playing strategies that were compatible to those produced by handcrafted chess playing program.

The following chapter, Chapter 8, will present a critical analysis of the use of genetic programming to evolve board game strategies and the methodology used to achieve the objectives of this dissertation outlined in Chapter 1.

Chapter 8

Methodology

8.1 Introduction

This chapter presents the methodology used to achieve the objectives of this dissertation that are outlined in Chapter 1. Section 8.2 presents an analysis of related literature. The aim and research methodology of this study is presented in sections 8.3 and 8.4. The board games used to test the approach are outlined in section 8.5 followed by a description of how this approach will be evaluated in section 8.6. The chapter is concluded with a technical specification of the hardware and software used to achieve the objectives in section 8.7, followed by the chapter summary in section 8.8.

8.2 Analysis of Genetic Programming in Board Game Research

From the survey on genetic programming applied to board games, it can be seen that the research has essentially focused on using GP for evolving evaluation functions to be used with other search techniques. There has not been sufficient research into using GP for evolving game playing strategies. Hence, the research in this thesis will investigate the use of GP for evolving heuristic based game playing strategies in real time. This will involve determining the GP representation, method of initial population generation, fitness function, selection methods, genetic operators, evaluation methods and the GP parameter values to be used.

The studies presented show that a number of methods were used to determine the fitness of an individual in the population at the end of each generation. Tournament play between elements of the same population was the most common method. Tournament play between coevolving populations, random move players and handcrafted computer players were used to a lesser extent [119, 10]. The short fall of using handcrafted players is that their quality of play will ultimately determine the quality of play of the evolving population. In most cases the handcrafted players were set to look no more than one or two moves ahead (search depth) due to the time it took to make moves using deeper searches. Furthermore, for most of the studies the computational and

time requirements necessary to evolve the computer players was considerably large [120, 126, 42]. Tournament play between coevolving populations proved successful [119, 10, 126] but proved time consuming. Therefore, for the purpose of this study, the fitness of individuals in the evolving population will be determined through tournament play between elements of the same population.

The studies also show that the playing quality of the evolved GP computer players was assessed through tournament rounds between different types of opponents and was met with varying degrees of success. The opponents included human players [119, 124], online players [119], pre-evolved computer players [10, 126], random move computer players [119, 120] and handcrafted players [119, 10, 120, 19]. For the purpose of this study, it was opted to use pseudo-random move players encoded with limited game playing strategies to assess the playing quality of the evolved strategies. The following section summarizes the aim and objectives of this study.

8.3 Aim of this Research

The main aim of this research is to investigate the use of genetic programming for evolving heuristic based game playing strategies for board games. This will be evaluated for games of varying levels of complexity to investigate the scalability of the approach. Three levels of complexity, namely, low, medium and high will be investigated. Othello will be used for the first level, checkers for the second and chess for the high level of complexity.

The objectives for this research, in summary are:

- Investigate a GP approach for evolving heuristic based game playing strategies for Othello.
- Investigate a GP approach for evolving heuristic based game playing strategies for checkers.
- Investigate a GP approach for evolving heuristic based game playing strategies for chess.

The following section presents the research methodology used to achieve these objectives.

8.4 Research Methodologies

There are four types of research methodologies that are most commonly used in Computer Science [132]. These are *proof by demonstration, proof techniques, mathematical, empiricism* and *hermeneutics and observational studies*. The objectives of this study will be achieved through the proof by demonstration methodology. This Computer Science methodology requires the development of a single approach which is iterated upon to achieve the stated objective(s). Firstly, an initial approach is developed based on an analysis of the literature. This initial approach is then tested and iteratively improved upon until a desired result is obtained or no further improvement can be made. At each iteration, the reason for failure must be identified and corrected. Changes to the approach are based on testing and these approaches should be produced to meet the objectives of the research question. For this research, the same methodology will be used for all three objectives.

As the aim of this study is to investigate the use of GP for evolving game playing strategies, the aspects that will examined and varied to obtain an improvement will be the representation and function and terminal sets, the method of initial population generation, the fitness function, the selection method and genetic operators and the GP parameter values. To achieve this an initial approach with sets of primitives and parameters will be implemented. These sets will be tested and iteratively refined until the objectives of this study are achieved. Thus, for this study the *proof by demonstration* methodology is considered the most suitable. Testing the approach will involve one run (Chapter 6, section 6.2) per move. If a suitable solution is not found, the possible reasons for this are to be identified. Refinement of the approach will then be performed. This involves correcting identified failures, changing primitives and varying the systems parameters and their value settings. In addition to the primitives and parameters of GP approach, it may also be necessary to change features of the GP algorithm such as the control model, the fitness function, selection method, genetic operators.

Aspects that will be varied in an attempt to improve performance are:

- The GP primitives.
- Representation.
- Fitness function.

- Genetic operators.
- The GP parameter values, namely:
    - Initial tree depth.
    - Selection method.
    - Selection method size.
    - Application rate of the genetic operators.
    - Maximum offspring depth.

The process of refining the GP algorithm and performance testing will continue until a desired result is obtained, i.e. good performing strategies or the strategies cannot be improved upon. The versions of the games that the GP approach will be applied to for testing are described in section 8.5. The measures used to assess the performance of the GP approach for each of the games is presented in section 8.6.

8.5 The Board Games Used For Testing

The approach will be tested on the board games Othello, checkers and chess.

- The version of Othello to be used for testing in this study is based on the international tournament rules of Othello [133] and is played on an 8x8 non-checkered board with 64 discs. All 64 discs are identical with one white side and one black side. The rules of Othello can be found on the following websites [133, 134].
- The version of checkers to be used in this study is based the standard British/American tournament rules [125] and is played on an 8x8 checkered board with 12 white and 12 black or red disc shaped pieces. The rules of checkers can be found on the following websites [125, 135].
- The version of chess to be used in this study is based on the standard international tournament rules [136] and is played on an 8x8 checkered board with 16 white and 16 black pieces. The rules of tournament chess can be found on the following website [136].

8.6 Performance Evaluation Measures

The performance of the approach will be evaluated for each game, namely, Othello, checkers and chess, through tournament bouts between the evolved GP game playing strategies and *pseudo-random* move computer players. Pseudo-random move players will for the most part randomly select a move position. There are however some pre-programmed game strategies that guide their move choices. For example, they are able to determine which moves are legal, which moves put them out of danger and which move will result in the capture opponent's pieces or to an extent gives them a strategic advantage, for example in Othello capturing a corner position. These move choices are given priority over a random move choice.

Objective 1: The approach will be evaluated for the game of Othello. The performance of the approach will be evaluated in 20 game tournaments against pseudo-random move Othello computer players. Performance will be calculated based on the number of points accumulated over the 20 game tournament bouts. Points will be allocated through games won or played to a draw. Two points will be given for each win and one point for each draw. For each game, the time taken for a move to be made will be recorded.

Objective 2: The approach will be evaluated for the game of checkers. The performance of the approach will be evaluated through tournaments of 10 games against pseudo-random move checkers computer players. The performance of the approach will be calculated based on the number of points accumulated over each 10 game tournament bout. Points will be allocated through games won or played to a draw. Two points will be given for each win and one point for each draw. For each game played the time taken to make a move will be recorded.

Objective 3: The approach will be evaluated for the game of chess. In previous studies [42, 130, 91] the authors chose to investigate chess endgames as opposed to a complete game and cited the complexity of analyzing the number of different strategies evolved during the opening, middle and endgame play as their reason. For this study, the approach will therefore be evaluated in chess endgames only. The performance of the approach will be evaluated through tournaments of 30 games against a pseudo-random move chess computer player encoded with limited chess playing strategies. Playing performance will be calculated based on the number of points accumulated over

each 30 game tournament bout. Points will be allocated through games won. Three points will be given for each win, 0 points for a draw and -3 points for a loss. In addition, 1 extra point will be given for each of the following, good strategies evolved, the capture of enemy pieces and the defense of own pieces. For each endgame, the time taken to make a move will be recorded.

8.7 Technical Specifications

The games, Othello, checkers and chess and the genetic programming approach were written in Java 8 (build 1.8.0_45-b15) using Netbeans 8.0.2. The software and hardware used were Windows 8.1 operating on a Toshiba© Intel© Core i7 4700MQ (2.40GHz) laptop with on-board Intel HD Graphics 4600 graphics card and 4GB DDR3 RAM. All simulations were run on the same hardware and software platforms.

8.8 Chapter Summary

This chapter provided an overview of the methodology used to achieve the objectives of this research as outlined in Chapter 1. An analysis of genetic programming in board game research was presented, highlighting the main focus area of previous work and how this study differs from the previous studies. The board games used for testing the approach were briefly outlined. Evaluating the performance of the GP evolved game playing strategies was presented followed by a description of the technical specifications used to develop and test the approach. The following chapter, Chapter 9, presents the genetic programming approach for evolving game playing strategies.

Chapter 9

Genetic Programming Approach for Evolving Game Playing Strategies

9.1 Introduction

This chapter presents the genetic programming approach for evolving heuristic based strategies for the board games Othello, checkers and chess. Section 9.2 presents an overview of the genetic programming algorithm. In this section the GP *representation*, *initial population generation*, *fitness evaluation* and *selection and regeneration* is detailed. Incorporating reinforcement learning and mobility strategies into the approach is presented in sections 9.3 and 9.4. The GP algorithm parameters and justification for their values is presented in section 9.5, followed by the chapter summary in section 9.6.

9.2 An Overview of the Genetic Programming Algorithm

An overview of the genetic programming algorithm used for evolving game playing strategies is presented in Figure 9.1. In this study, the approach makes use of a generational control model as described in Chapter 6, section 6.2. Each individual of the population is represented as a parse tree. To begin with, an initial population is created and then is iteratively refined through the process of evolution. This occurs by firstly evaluating each individual of the population and calculating its fitness. Parents are then selected and genetic operators are used to create offspring for the population of the next generation. The process of evaluation and regeneration continues until a terminating criterion is reached.

Algorithm 9.1. Genetic programming algorithm used for evolving game playing strategies

1. Create an initial population of candidate strategies.
2. Evaluate each strategy in the population and use a fitness function to calculate its fitness.
3. Update the heuristic values for each board position.
4. Create a new generation of strategies by:
    a. Selecting parents using tournament selection.
    b. Applying genetic operators to the selected parents.
5. Repeat from 2. until *g* generations have been reach
6. Return the best candidate strategy

The process of creating an initial population and evolving it over a set number of generations, as discussed in Chapter 6, section 6.2, is referred to as a run. For this study, a separate run is performed for each move in a game. Each game playing strategy represents a player and is composed of heuristics representing the strategic areas on the board for a particular game. A strategy will be applied by considering all valid moves and choosing the piece in the position with the highest heuristic value to make the move. The game playing strategies are evolved in real time during the game play.

Each run starts with initial population generation followed by *n* generations of evolution. For each generation, the reproduction, mutation and crossover operators are applied to create the population. Information from one run, namely a game move, is passed on to the next move by preserving the population and using it as the initial population of the next move. The best performing strategy, termed the *alpha player* for this study, is maintained throughout the run and is used to evaluate the population. The fitness of each game playing strategy, i.e. computer player, is evaluated by playing *n* games against the alpha player. Due to the stochastic nature of genetic programming, the pertinent number of tournament games to be played is determined through experimentation. The fitness of each player is calculated based on the number of points accumulated through wins and draws. The alpha player is randomly selected at the beginning of a run from the initial population or, if a move has already been made, the alpha player from the

previous run will be used. Checks are performed to ensure that each move is valid and when the game ends. The processes of *initial population generation*, *evaluation*, and *selection and regeneration* are described in the following subsections.

9.2.1 Initial Population Generation

One of three methods can be used by the genetic programming algorithm to create the initial population, namely, *grow*, *full* and *ramped half-and-half*. These methods have been described in Chapter 6, section 6.5. The preferred method for creating the initial population in this study is the ramped half-and-half method with a depth limit of six as this method produces a variety of tree shapes and sizes. Each element of the population represents a player and defines a game playing strategy in terms of heuristics. A game playing strategy in this case is defined by each position on the game board having a heuristic value. The strategy is implemented by selecting the position with the highest heuristic value to make the move. During the initial population generation, a board position and heuristic value from a predefined range are randomly generated.

Each heuristic value is mapped to its corresponding board positions before a move is made. Board positions are represented sequentially, starting at position 1 and ending at position *p* depending on the size of board. This is illustrated in Figure 9.1.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Figure 9.1. Board positions of a 8x8 game board

9.2.1.1 Terminal Set

The GP terminal set consists of one terminal that is comprised of two numeric values, the board position, designated *B*, as illustrated in Figure 9.3 and its heuristic value, designated *H*.

- Terminal Set: t ={(*B* = board position, *H* = heuristic value)}

For example, a terminal with heuristic values 3,20 depict that board position 3 will be assigned a value of 20.

The x and y co-ordinates of any board position *B*, on the 8x8 board (0,0 .. 7,7) can thus be calculated from the sequentially numbered positions, 1 to p:

- $y = (B-1)/8$
- $x = (B-1) - (y*8)$

During initial population generation, two values are randomly generated for each terminal in the terminal set. The first value is the board position and the second is the heuristic value. The range of values assigned to the board positions is selected from a preset range of values, namely, *heuristic value range* and is a parameter setting. This is depicted in table 9.1. Figures 9.3 and 9.4 illustrate how the terminal heuristics will be mapped to the game board before a game move is made. The board position with the highest heuristic value will be selected to make the move.

During the mapping process, board positions that are not represented in a strategy will be assigned a value of zero. If a board position appears more than once in a strategy with different heuristic values, the first heuristic value is chosen. By reducing the terminal set to a single terminal that accepts board positions and heuristic values, game boards of any size can be catered for. In the figure below, Figure 9.2, tBH represents a terminal *t* having a heuristic of two values, i.e. *B* the board position and *H* the heuristic value assigned to that board position. The board heuristic values for the strategy depicted in Figure 9.2 are shown in Figure 9.3.

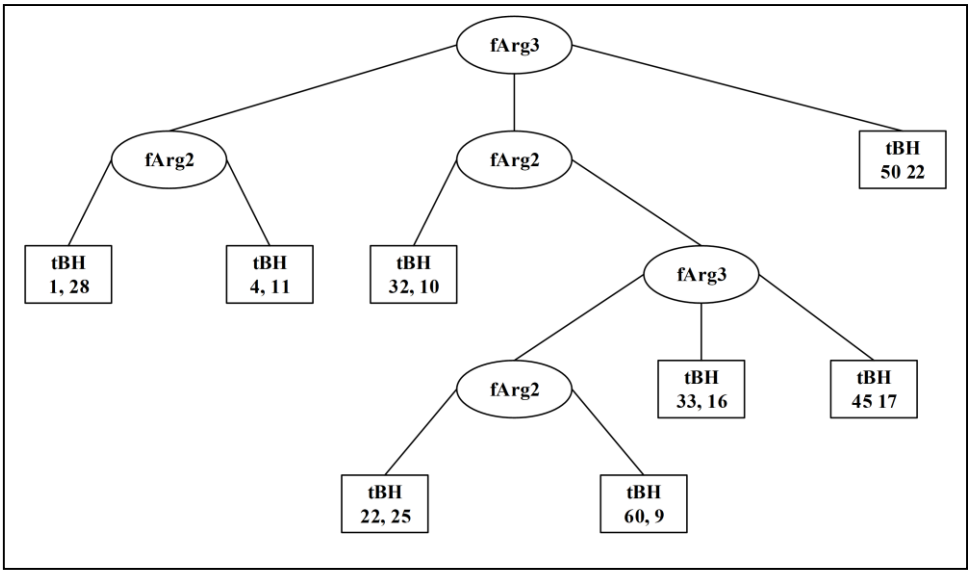


Figure 9.2. Example of an element in population representing one strategy

Board position 1 →

28	0	0	**11**	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	**25**	0	0
0	0	0	0	0	0	0	**10**
16	0	0	0	0	0	0	0
0	0	0	0	**17**	0	0	0
0	**22**	0	0	0	0	0	0
0	0	0	**9**	0	0	0	0

← Board position 64

Figure 9.3. Board heuristic values for the strategy depicted in Figure 9.2

9.2.1.2 Function Set

The elements of the *function set* are used to combine heuristics to form game playing strategies. The aim of these operators is to combine two or three primitives (Chapter 6, section 6.4)

respectively and are similar to the *block* operator used by Koza [45]. The function set contains two operators, namely, fArg2 and fArg3 that take 2 and 3 arguments respectively. This combination allowed parse trees of varying widths to be produced.

- Function Set: fs = {fArg2, fArg3}

Although each game playing strategy is composed of heuristics, not all the heuristics are necessarily contained in a single strategy. An example of a strategy composed of heuristics is illustrated in Figures 9.2. The board heuristic values allocated by the playing strategy depicted in Figure 9.2 are displayed in Figure 9.3. As mentioned above, the board position with the highest heuristic value is used to choose the next move. If there is more than one position with the highest heuristic value, a position to make the move is randomly chosen from the positions that share that value. This method did not encourage valid moves while excluding invalid moves as it is anticipated that evolutionary process will expunge invalid moves. During fitness evaluation (sub-section 9.2.2), invalid moves are assigned a fitness of zero.

9.2.2 Fitness Evaluation

The population is evaluated by each element playing against the alpha player over a preset number of games. Ten games proved to be sufficient for the purpose of fitness evaluation. This is illustrated in Figure 9.4. Each game in the evaluation tournament starts at the current board configuration and is played to the end of the game. The alpha player starts each alternative game in the evaluation tournament, thus giving both players an equal chance. For this study, it was determined that using the sum of wins and ties [10] to calculate fitness did not truly reflect the fitness of a player. It was found that this measure, to some extent, hides information as only the wins and losses are considered and hence does not truly reflect the fitness of each player. For example, a player could appear the strongest player after accomplishing only near wins. Similarly, a potentially stronger player would be considered a weaker player as a result of losing a few games by a small margin, even though the games won were achieved through outright wins. The measure, wins and losses, for fitness is acceptable if you are playing a round-robin tournament, i.e. the accumulative wins of an individual is a measure of its fitness. However, in this approach each player is matched against a single alpha player during the evaluation process. The problem arises in that the alpha player may beat the majority of the population and only get beaten by a very small

percentage. If two players each win, for example, 6 out of the 10 games played against the alpha player they each have a fitness value of 6. This does not reflect how well each player performed. Hence, a measure that was considered a better representation of the fitness of the player is the sum value of the player's pieces left on the board at the end of the game, accumulated after a preset number of games. This proved successful for Othello and checkers but not for chess. This is discussed in subsection 9.2.2.1. For Othello the points value for each disc is 1 point. For checkers the points value for each piece is 1point for a single disc and 2 points for a King.

To illustrate and using Othello as an example, the first strategy may have just beaten the alpha player 6 games out of a 10 game tournament and averaging 34 discs to the alpha player's 30 discs per game. Another strategy could win with an average 58 to 6 discs. In terms of wins both players have a fitness of 6 but in terms of disc count the first player will have a fitness of 6*34=204, whereas the second strategy will have a fitness of 6*58=348 a difference of 144.

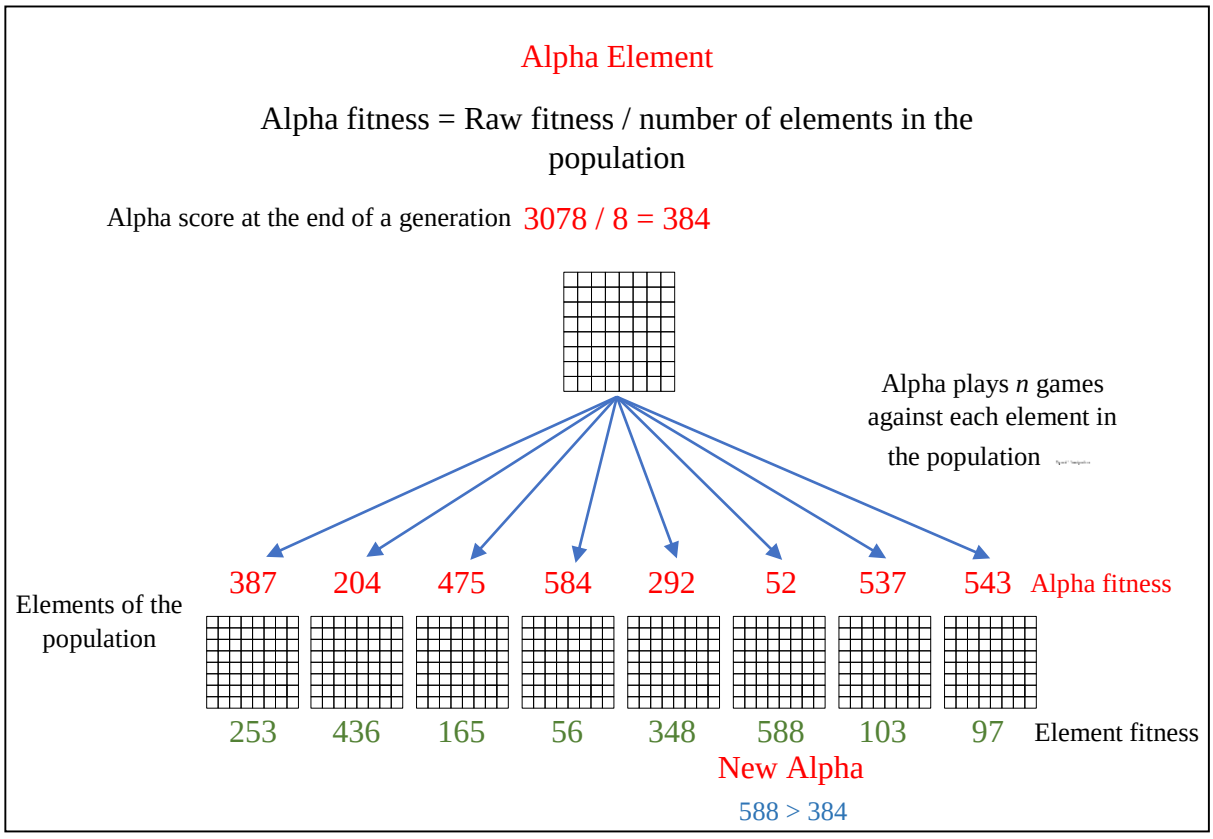


Figure 9.4. Evaluation tournament to determine fitness and select the alpha player

At the end of each generation, the fitness of each element in the population is compared to that of the alpha player's fitness. The alpha player's fitness is determined by dividing its total fitness by the number of elements of the population as a better reflection of its playing ability. This is because the alpha player plays as many game sets as the number of elements in the population, while each element of the population plays only one set. At the end of the evaluation process, the element that achieves the highest fitness is promoted to alpha status, replacing the alpha player in the next generation. This process is illustrated in Figure 9.4. In the event that more than one element has the same fitness but greater than that of the alpha player's, a new alpha player is created from those elements. The heuristic value for each board position of the new alpha player is calculated by applying the sum of heuristic values for that board position from each of the contributing elements, divided by the number of elements making up the new element. The new alpha player will then participate as the alpha player in the next generation if the parameter value *Preserve Alpha*, depicted in Table 9.1 is set to true.

The fitness evaluation method proposed above proved successful for Othello and checkers only but was not successful for chess. The reason being is that for Othello pieces are placed and not moved from their positions. For checkers pieces moved predominantly in one direction only (the checkers king piece being the exception). In chess however, the major pieces (non-pawns) move in multiple directions throughout the game. The following describes the shortfall of using the above fitness evaluation method of evaluating the performance of an element against a single alpha player for chess.

Figure 9.5 shows a single white move of a candidate strategy within a population. Figure 9.6 shows the alpha player making a single move. The alpha player is playing black. Board position heuristic values beyond the fourth row of the colour that is to move, i.e. white in Figure 9.5 and black in Figure 9.6, has not been shown in this example for the purpose of clarity. In figure 9.5, these are rows 1 to 4 (white to move) and in figure 9.6 these are rows 5 to 8 (black to move).

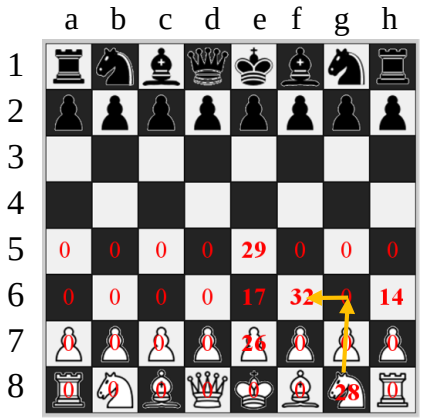


Figure 9.5. A single white move of a candidate strategy

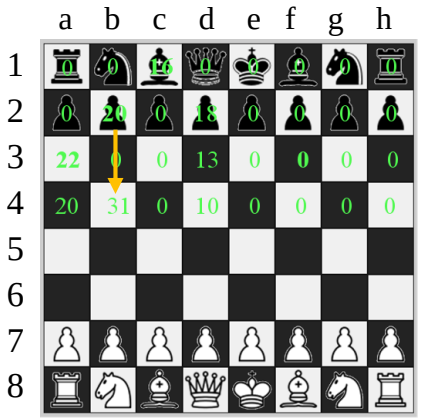


Figure 9.6. A single move of the alpha player

The following sequence takes place:

- White moves first. The King's Knight has the highest heuristic value so will be selected. This piece will move to the position having the highest heuristic value, i.e. 32 at position f6. Black will move its Queen's Knight-Pawn having the highest heuristic value, i.e. 20, two squares forward to position b4 which has the highest heuristic value, i.e. 31.
- White will move its King's Knight back to original position g8 as g8 now has the highest heuristic value.
- Black will move Queen's Bishop to a3.
- White will simply continue moving between the two positions f6 and g8.
- Black falls into the same trap moving between positions a3 and b4.
- The game will be terminated by the 50 move no take rule. This rule states that if no piece is captured and no checkmate is eminent within 50 moves after the last piece capture, the game is ended and declared a draw [18].

As a result of this, during the evaluation process, games do not play out to the end as pieces are often trapped in this infinite loop and all elements of the population end up with equal fitness values. An alternative method for calculating the fitness value of each element was thus investigated for chess. This following subsection details this method.

9.2.2.1 Alternative Method for Chess Fitness Evaluation

A potential solution to the problem described above is to evaluate the strategy of each element in the population, based on two consecutive moves only and not through an entire game as is implemented for Othello and checkers. The fitness value of each element is thus calculated by each strategy in the population making two moves against *r* number of pseudo-random move players of the opposite colour. This approach generates a sub-set of moves from the total moves available to a strategy at a particular board configuration. The pseudo-random move players will for the purpose of this study be termed *training players*. The board is reset to its original state after each pair of moves. The fitness value of a strategy is rewarded or penalized depending on the resultant board configuration and on its performance against the training players. This in principle can be viewed as a process of *real-time learning*. In this study 1, 5, 10, 15, 20 and 25 training players were used to generate the sub-sets of moves. A two-move look ahead was determined to be sufficient for this study as a number greater than 2 often resulted in a piece moving to and fro between two positions having the highest heuristic values.

The following describes the reward and penalty point scores used to calculate the fitness value of a strategy. The total value of white or black pieces on a chessboard is 39 points based on the British and American international tournament rules [92]. This value is calculated as the sum of 8 Pawns worth 1 point each, 2 Knights and 2 Bishops worth 3 points each, 2 Rooks worth 5 points and a Queen worth 9 points. The value of 39 is used as a reference value for choosing the reward and penalty values. The term *friendly* refers to the GP computer player or piece colour of the GP player. The term *enemy* refers to the player or colour opposing the GP computer player.

Rewards:

- The sum of positions made available to the player that has just moved (this is also in essence a simple measure of its mobility).
- The sum of the points value for each piece on the board after a move (this is termed *piece score*).
    - Pawn = 1 point.
    - Bishop and Knight = 3 points.
    - Rook = 5 points.

- Queen = 9 points.

- If the enemy's piece score before the move is less than after the move, this means an enemy piece has been captured and the friendly player is rewarded 20 extra points.
- 50 points are awarded if the enemy King is checked as a result of that move.
- 1000 points are awarded if that move results in the checkmate of the enemy player.

Penalties:

- If the friendly piece score before the move is less than after the move, a friendly piece has been captured and the friendly player is penalized 50 points.
- The friendly player is penalized 100 points if a move by the enemy results in the friendly King being checked.

The overall fitness value is calculated by the sum of points gained through rewards minus the sum of points lost through penalties, accumulated after the *evaluation tournament*, i.e. two moves against *r* training players. It was determined through experimentation that the first of the two moves should have a greater score value weighting than the second move, as the first move contributes significantly more to the immediate or local strategy. Thus 80% of the points accumulated for the first of the two moves and 20% of the points accumulated for the second of the two moves is used to calculate the fitness value of the element.

9.2.3 Selection and Regeneration

For this study, tournament selection [45] with a tournament size of four is used to choose parents to create offspring of successive generations. In summary, this method randomly chooses a preset number *t* elements from the population. The fittest element is returned as the winner of the tournament and a parent. Selection is with replacement, so an element of the population can serve as a parent more than once. The reproduction, mutation and crossover operators are used for regeneration [45]. The reproduction operator copies the parent chosen through tournament selection, unchanged, into the next generation. For the mutation operator, in this study, only terminal nodes are selected as single mutation points. This is done to preserve the structure of the parse tree. The mutation operator randomly selects a single terminal node and replaces the node with a randomly generated terminal. Hence, mutation is the only operator that can change the heuristic value of a terminal. The crossover operator randomly selects a crossover point in each of

the parents and swaps the subtrees rooted at the crossover points to create two offspring. For this study, any node, i.e. a function or terminal node, except for the root node, is selected as a crossover point. In this way, terminal nodes can replace function nodes and *visa versa*. The crossover operator alters the structure of the parent chromosomes to create offspring but preserves parts of the genetic makeup from each parent.

Genetic operators are applied according to application rates. For example, for 0.6 crossover, 0.2 mutation and 0.2 reproduction with a population size of 100, will result in 60 offspring through crossover, 20 offspring through mutation and 20 through reproduction in the next generation. Offspring that exceed the preset size limit, that is the number of nodes making up the pares tree, are pruned by replacing the rightmost function nodes with terminal nodes in the subtree of which the function node is the root. This is illustrated in Figures 9.7 and 9.8. No limit is placed on the depth of the parse tree.

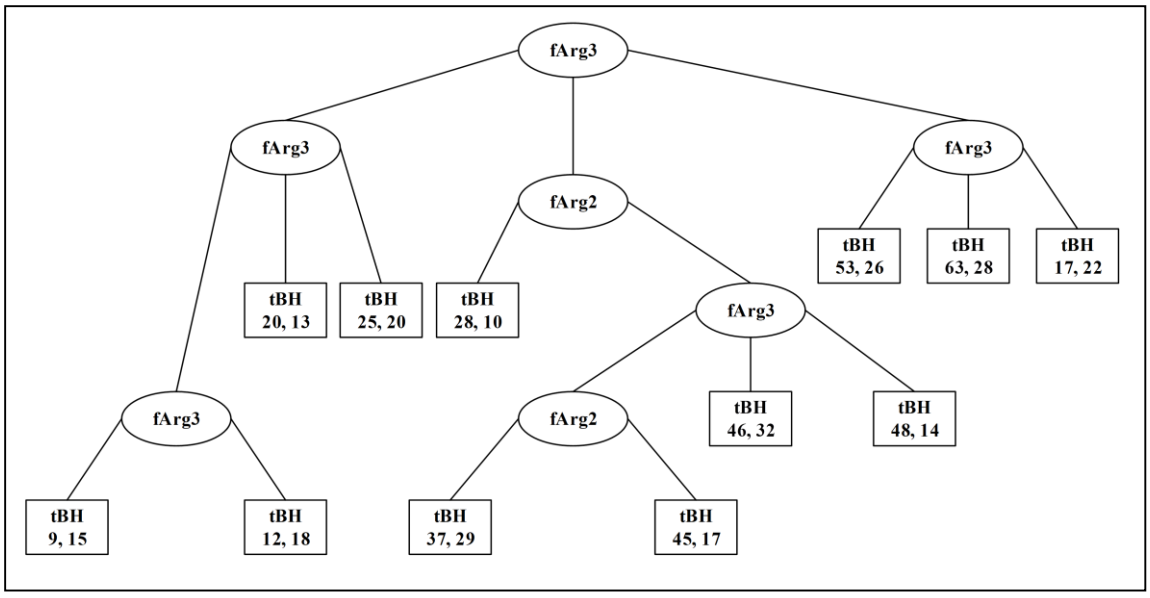


Figure 9.7. Parse tree before pruning

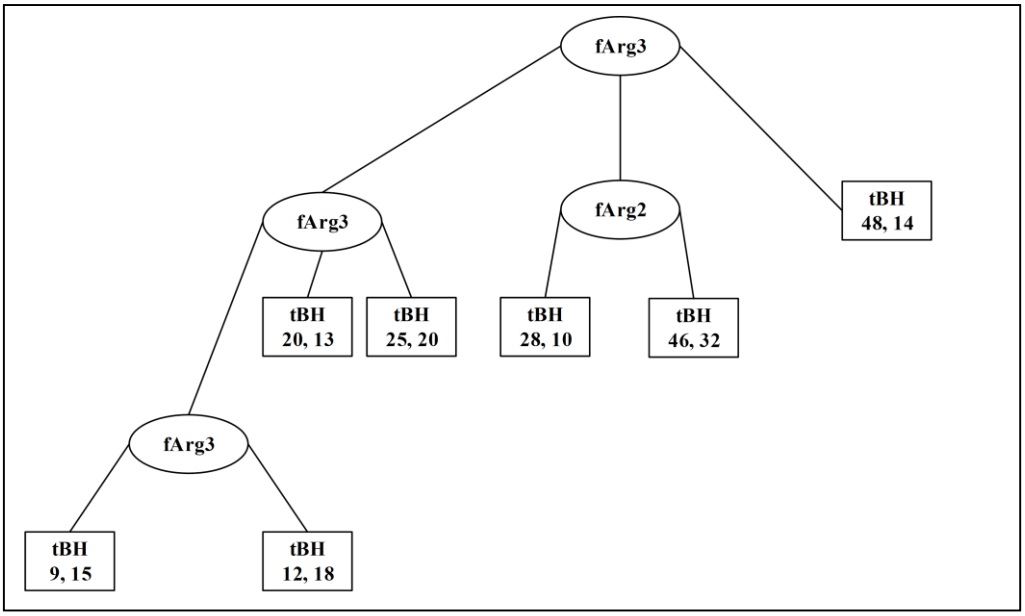


Figure 9.8. Parse tree after pruning

The alpha player from a generation can be used as a parent in the successive generation if the parameter value controlling this, namely, *Alpha Elite*, is set to true. In this case, for crossover, the alpha player and a parent are selected from the population are chosen to produce two offspring. In the case of mutation, it is randomly decided whether to mutate the alpha player or not.

The following sections outline two methods incorporated into the approach in an attempt to improve the performance of the GP evolved strategies.

9.3 Reinforcement Learning

Sutton and Barto [71] describe *reinforcement learning* as learning to map certain actions to situations in order to maximize reward. In this way, a learning computer player is not told which strategies to execute (brute force programming) but must instead discover which moves result in the most reward by trying them out. Methods of reinforcement learning are often incorporated into computational intelligence techniques [87, 88]. Hence, correct actions result in rewards and incorrect actions are penalized [36]. The reasoning behind the reinforcement learning method that is used in this research is to capture the performance of the heuristics in good and weak strategies during a run and update the heuristic values of the alpha player accordingly at the end of a run.

The heuristics of those strategies that perform well are selected and used as rewards to update the heuristics of the alpha player. The heuristics of those strategies that perform poorly are also selected but their heuristics are used as penalties when updating the heuristics of the alpha player. The term *matrix* in this discussion refers to the *x*,*y* coordinate mapping of a strategy onto a game board.

The following outlines the reinforcement learning method used in this study in an attempt to improve strategies and thus improve the performance of the alpha player. In addition to the matrix resulting from the strategy represented by the alpha player, two additional matrices are produced. One matrix stores the average heuristic value of each board position for all the strongest elements, i.e. those elements that are promoted to alpha status at each generation. A second matrix stores the average heuristic values of each board position for all the weakest elements, i.e. those elements that result in a fitness of less than a preset percentage of the alpha player's fitness. This percentage value was determined through experimentation and set to be 30%. Learning is achieved by producing a single heuristic value, based on rewards and penalties, for each board position by means of the following formula:

- Learned matrix (x,y) = Alpha matrix (x,y) * Strongest matrix (x,y) / Weakest matrix (x,y)

Figure 9.9 illustrates this method.

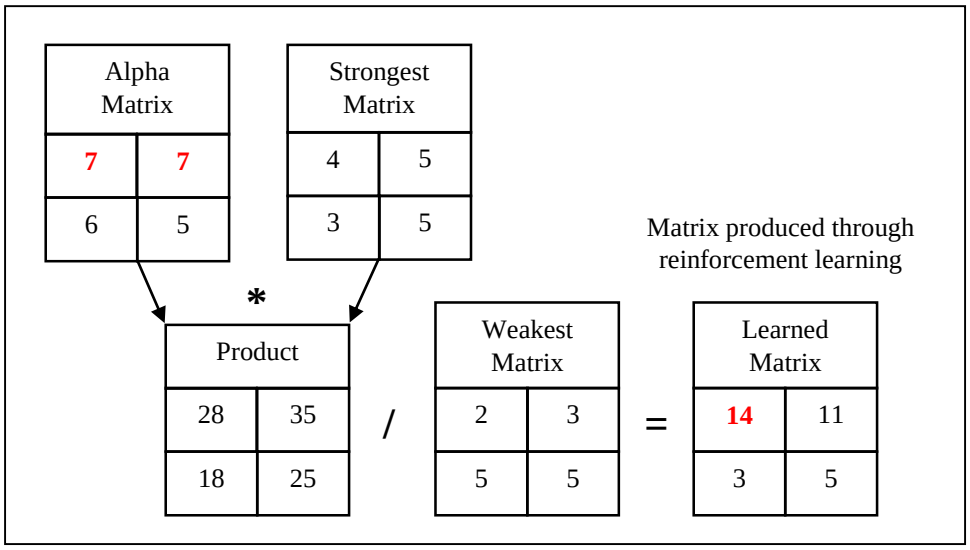


Figure 9.9. Reinforcement learning method

In the following example, the matrix resulting from the strategy represented by the alpha player depicts two board positions that have equal heuristic values. Both positions have a heuristic value of seven. By applying the above formula to each of the alpha player's board position heuristic values, certain values are improved upon while others are worsened. As a result, in this example, the position having the heuristic value of 14 is selected to make the move.

In this study, a different method for evaluating the strategy of each element in a population is used for chess. This is discussed in the above sub-section 9.2.2.1. Unlike the evaluation method used for Othello and checkers, this method evaluates each strategy based on two consecutive moves against *r* number of pseudo-random move players of the opposite colour and not through an entire game against an alpha player. As a result, the method to calculate the average heuristic value of each board position for all the strongest and weakest elements could not be achieved. Hence, the reinforcement learning technique implemented for Othello and checkers could not be implemented for chess.

9.4 Mobility Strategies

In addition to the reinforcement learning method implemented in this research a second method, mobility strategies (Chapter 3, section 3.2.2.2) is investigated in an attempt to improve the evolved game playing strategies. This method is incorporated into the approach after reinforcement learning and before a game move is made.

For this study, a one-move look ahead mobility strategy for Othello is implemented. Hence, when selecting the position to make a move, all valid positions are evaluated, by looking one move ahead. This determines which move will result in the least moves available to the opponent. The mobility value is calculated as the best heuristic value on the board resulting from making the move that results in the least moves for the opponent in the next move. When applying a game playing strategy the heuristic values assigned by the strategy are used in conjunction with the mobility strategy as follows.

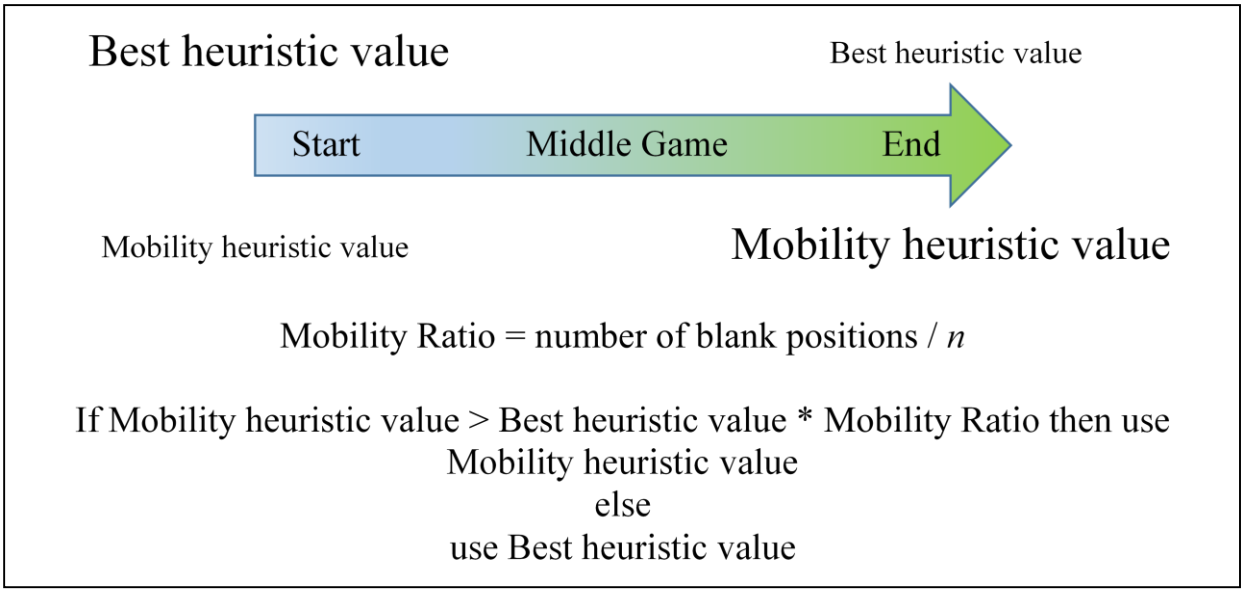


Figure 9.10. Heuristics incorporating mobility strategies

There is a subtle play between moves based on the best heuristic values and mobility heuristic values to determine the next move. Thus, a ratio between heuristic values and the mobility values are used to decide on which move to make. This ratio is a measure of the amount of game play that has taken place. This calculation and the sliding scale for the mobility method implemented for Othello is depicted in Figure 9.10.

Test runs of the approach indicated that it is more effective to use heuristic values at the start of a game, i.e. < 20 moves made, with mobility strategies becoming more important in the mid-game and increasing towards the end of the game, i.e. > 45 moves made. Limiting your opponent's mobility in the endgame severely restricts play, often forcing that player to forfeit moves or make moves to your advantage. This is a vital strategy to ensure the advantage is kept, leading to game success [137]. In the middle-game, the use of mobility strategies and the best heuristic value is balanced. The following rule is used to achieve this; *if the mobility value is less than the product of the best heuristic value and mobility ratio then the mobility value is used to decide which move to make otherwise the heuristic value is used to make this decision*.

The value of *n* in Figure 9.10 represents the number of moves into game-play and is essentially a weight used to control the effect of mobility. Tests performed indicated a value of 42 was

appropriate for this study. If *n* is set to 32, i.e. 64 board positions divided by 2, mobility comes into play half way through the game. The smaller the value the earlier on in the game mobility comes into play. Mobility strategies become more and more important in the last third, or towards the endgame. Thus with *n* set to 42, mobility is evaluated only over the last 18 free squares, representing the endgame.

The one-move look ahead mobility strategy method implemented for Othello could not be extended to the more complex games of checkers and chess. The reason being, Othello is a disc-placing game with all discs being equal and unlike checkers and chess, pieces are not moved about on the board during game play. Mobility strategies for Othello can thus be determined by looking one move ahead to establish all valid positions that will result in the least moves available to the opponent. For games that have multiple pieces of different types involved in a strategy, each piece on the board requires its own mobility strategy to be calculated [49, 138]. This can only be accomplished through game specific instructions.

9.5 Parameters

Table 9.1 presents the parameters and their values used for the genetic programming algorithm for each of the three games. For this study, the parameter values were determined through experimentation.

The following outlines the population size and number of generations evolved for each run (game move):

- For Othello a population size of 25 individuals evolved over 25 generations, referred to in this study as *version 25/25* and a population size 50 evolved over 50 generations, referred to as *version 50/50* are used. A population size greater than 50 increased the time it took the GP to find a solution and did not improve performance. Fifty generations allowed for sufficient convergence without creating high runtimes for real time play. Tests with runs with more generations did not produce fitter individuals.
- For checkers a population size of 50 individuals are evolved over 25 generations. A population size greater than 50 increased the time it took the GP to find a solution and did

not improve performance. Test runs of greater than 25 generations increased the time it took the GP to find a solution but did not produce fitter individuals.

- For chess a population size of 50 individuals are evolved over 10 generations. A population size greater than 50 increased the time it took the GP to find a solution and did not improve performance. Test runs of greater than 10 generations did not produce fitter individuals but increased the time it took the GP to find a solution.

The ramped half-and-half method with a maximum initial depth of 6 is used to generate the initial population for all three games. These values catered for a more general representation (Chapter 6, section 6.3). The tournament size was set to 4 individuals, thus promoting selection of better performing individuals. The genetic operator rates were decided during the study, through observation of different combinations and selecting those rates that produced the best results. In this study, mutation is given a typically low application rate of 0.2 as this genetic operator is a global search operator resulting in high genetic diversity during the evolutionary process. Crossover on the other hand is a local search operator, which limits variety thus preserving genes and was set to 0.8, higher than the standard value of 0.6 suggested by Koza [45]. This is important as one of the main objectives of the experiment was to evolve a population that was highly fit and combines several individuals for a final result. By sharing more genetic code through increased crossover, it was anticipated that, as search is focused on a particular area of the search space, this would result in better individuals at the end of each run. The probability used for reproduction was set to 0.0, as the actions of both the mutation and crossover operators could result in reproduction if the resultant offspring is identical to the parent. The *maximum offspring node limit* is set to 1000 nodes to cater for a more general representation.

The *heuristic value range* for Othello is set at 1 to 32. The top value 32 was chosen simply on the basis that this is half the number of squares on an 8x8 Othello board. For checkers and chess it was found that a range of 1 to 64 produced better results as this range catered for a larger difference between board position heuristic values. Ten game tournaments proved sufficient to determine the fitness of individuals in a population and so the *fitness evaluation games* parameter is set to 10. The parameter values *Alpha Elite* and *Preserve Alpha* are set to true for all three game experiments as it is anticipated that these would preserve good genes by passing some of the alpha player's

genes to offspring making up the next generation and to the offspring of the initial generation of the next run (game move).

The number of game moves to end a game was decided during the study by testing different values. Othello has a finite number of game moves to the end of a game, i.e. 60 moves or less therefore the *game moves* parameter is not applicable to Othello. This is not the case for checkers and chess. In certain cases where equally matched opponents are able to continually move pieces on the board without changing the status of the game, a draw is declared. In this case the game may be terminated after *n* number of moves have been made without a win or stalemate. For this study, 150 game moves proved a sufficient number of moves before a game is declared a draw and terminated. The parameter *game moves* is therefore set to 150 for checkers and chess.

Table 9.1. Genetic programming algorithm parameter values

	Othello	**Checkers**	**Chess**
Parameter	*Value*	*Value*	*Value*
Population Size	25 or 50	50	50
Maximum Depth	6	6	6
Generation Method	Ramped half-and-half	Ramped half-and-half	Ramped half-and-half
Selection Method	Tournament	Tournament	Tournament
Tournament Size	4	4	4
Reproduction	0.0	0.0	0.0
Crossover	0.8	0.8	0.8
Mutation	0.2	0.2	0.2
Generations	25 or 50	25	10
Maximum offspring node limit	1000	1000	1000
Heuristic value range	1 to 32	1 to 64	1 to 64
Fitness evaluation games	10	10	10
Alpha Elite	True	True	True
Preserve Alpha	True	True	True
Game Moves	N/A	150	150

9.6 Chapter Summary

This chapter provided an overview of the general GP approach used to evolve heuristic based game playing strategies for Othello, checkers and chess. Heuristic values, combined into the strategies, are randomly generated and mapped to specific board positions. Through the process of evolution these strategies are improved upon. The position with the highest heuristic value is selected to make the game move. Reinforcement learning and mobility methods are incorporated into the approach in an attempt to improve the performance of the evolved strategies. Finally, the chapter presented the parameters used by the genetic programming algorithm for evolving game playing strategies. The following chapter, Chapter 10, will detail the results of this study and present a discussion of the approach.

Chapter 10

Results and Discussion

10.1 Introduction

In this chapter, the results of the GP approach for evolving heuristic based game playing strategies for Othello, checkers and chess is presented. Section 10.2 deals with the results and discussion for Othello, presenting a performance comparison between Othello players in subsection 10.2.1. Subsection 10.2.2 presents an analysis of the Othello game playing strategies evolved by the GP approach. Section 10.3 covers the results and discussion for checkers, presenting a performance comparison of checkers GP players and an analysis of the checkers game playing strategies evolved by the GP approach. The results and discussion for chess is presented in section 10.4. The performance of the GP approach is outlined in subsection 10.4.1. The analysis of chess game playing strategies evolved by the GP approach is presented in subsection 10.4.2. The chapter summary is presented in section 10.5.

10.2 Results of the GP Approach for Othello

This section reports on the performance of the heuristic based GP approach in inducing game playing strategies for Othello and compares the performance with reinforcement learning and mobility incorporated. To determine the quality of the GP evolved strategies and the effectiveness of incorporating reinforcement learning and mobility into the genetic programming approach for Othello, four experiments were conducted:

Experiment 1: This experiment compares the performance of the heuristic based GP approach presented in this study, designated *HGP*, without reinforcement learning and mobility, to a pseudo-random move player encoded with limited Othello game playing strategies. These encoded strategies prioritize moves that result in corner and key edge positions being captured (Chapter 3, subsection 3.2.2.1). The pseudo-random move player (Chapter 8, section 8.6) is referred to as

RPlayer. The performance of the *HGP* will be compared to the *RPlayer* simply to determine whether the *HGP* can demonstrate improved levels of playability.

Experiment 2: This experiment compares the performance of *HGP* to the GP approach incorporating reinforcement learning and is designated *RHGP*.

Experiment 3: This experiment compares the performance of the *HGP* to the GP approach using mobility and is designated *MHGP*.

Experiment 4: This experiment compares the performance of the *HGP* to the GP approach incorporating both reinforcement learning and mobility and is designated *RMHGP*.

Each of the four experiments were repeated twice, firstly with a population size of 25 evolved over 25 generations (version 25/25) thereafter with a population size of 50 evolved over 50 generations (version 50/50). A population size of 25 evolved over 25 generations was initially chosen for the experiments. It was determined through trial runs that an increase in population size evolved over a greater number of generations improved the performance of GP evolved players. However, population sizes greater than 50 individuals evolved over more than 50 generations did not show improved performance of GP evolved players.

Tournament rounds of 20 games per experiment were conducted with each player alternating between black and white per game. For example, in experiment one, the players evolved by *HGP* played against pseudo-random move players. *HGP* played ten games as white and ten games as black. Twenty games per experiment proved to be sufficient as a greater number did not produce improved results. The number of wins were recorded for each experiment as well as the time taken for the GP player to make a move. Each experiment was carried out using random seed values (Chapter 6, section 6.2), with the GP algorithm parameters depicted in Chapter 9, Table 9.1. A comparison of the performance of *HGP* and *RPlayer*, *RHGP*, *MHGP* and *RMHGP* is firstly presented in subsection 10.2.1, followed by an analysis of some game playing strategies evolved by *RMHGP* in subsection 10.2.2.

10.2.1 Performance Comparison

The experiments have revealed that game playing strategies were improved through the implementation of reinforcement learning techniques and to some extent mobility strategies.

Experiment 1 – *HGP* vs. *RPlayer*: *HGP* was found to outperform the *RPlayer* for both 25/25 and 50/50. The 25/25 version of *HGP* won 18 of 20 games and the 50/50 version of *HGP* won all 20 games. The *RPlayer* was able to marginally beat the *HGP* in two games due to being encoded with the ability to prioritize capture of corner, key edge positions. However, an increase in the number of generations to allow further learning for the *HGP* allowed the *HGP* version 50/50 to win all 20 games.

Experiment 2 – *HGP* vs. *RHGP*: *RHGP* was found to perform better than *HGP* with the 25/25 version winning 15 of the 20 games and the 50/50 version winning all 20 games. These results show that the incorporation of reinforcement learning improves the decision making process.

Experiment 3 – *HGP* vs. *MHGP*: *MHGP* performed better than *HGP* but not as well as *RHGP*. For the 25/25 version of *MHGP*, 11 of the 20 games were won and for the 50/50 version 15 of the 20 games were won. The game play was observed in order to find an explanation for this performance as it was hypothesized that *MHGP* would perform better. It was found that the use of a mobility strategy resulted in one of two scenarios. Limiting moves by means of mobility forcing the *HGP* to either forfeit moves or make bad move choices resulting in a win for *MHGP*. Alternatively, limiting the choice of moves can also force *HGP* to immediately make moves to the take corner or key edge positions or positions of high importance and gain advantage over *MHGP*.

Experiment 4 – *HGP* vs. *RMHGP*: *RMHGP* performed better than *HGP* and *MHGP* but not as well as *RHGP*. The 25/25 version of *RMHGP* won 13 of the 20 games and the 50/50 version won 17 of the 20 games. The incorporation of reinforcement learning into *RMHGP* resulted in the improved performance over *MHGP*.

Figure 10.1 and Figure 10.2 provide an overview of the results obtained by *HGP*, *RPlayer*, *RHGP*, *MHG* and *RMHGP* for the 25/25 and the 50/50 versions. The experiments have also revealed that that the 50/50 version performed better than the 25/25 version. This could possibly

be attributed to the larger population size representing more of the search space and a larger number of generations that is needed for the genetic programming algorithm to converge.

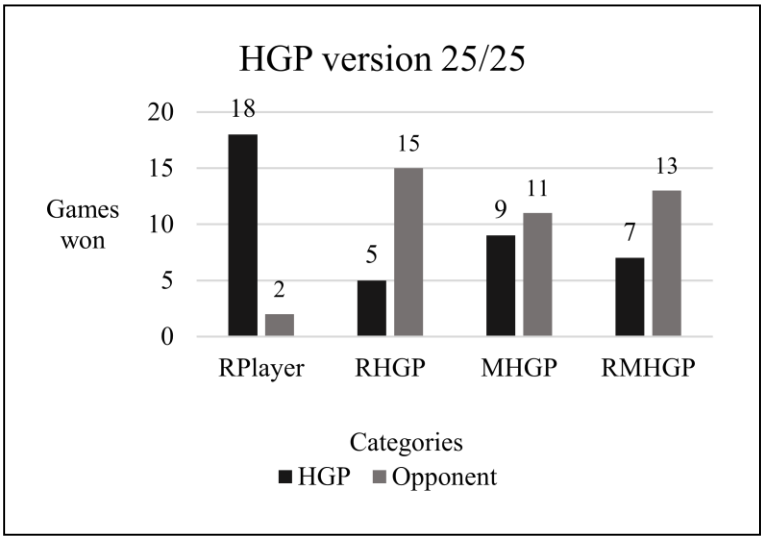


Figure 10.1. Games won with a population size of 25 evolved over 25 generations

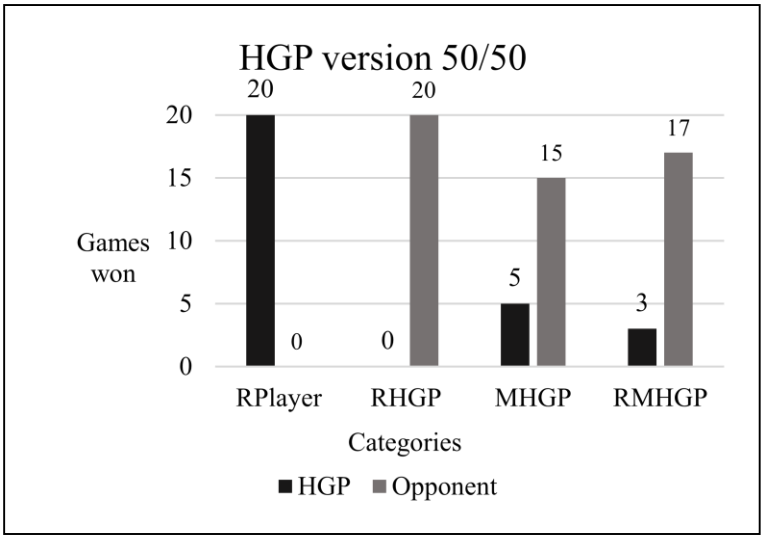


Figure 10.2. Games won with a population size of 50 evolved over 50 generations

One of the differences between this study and similar work in this field is that strategies are evolved in real time during game play instead of offline. As is evident from Figure 10.3 and Figure 10.4 the GP approach was able to evolve winning strategies in under a minute.

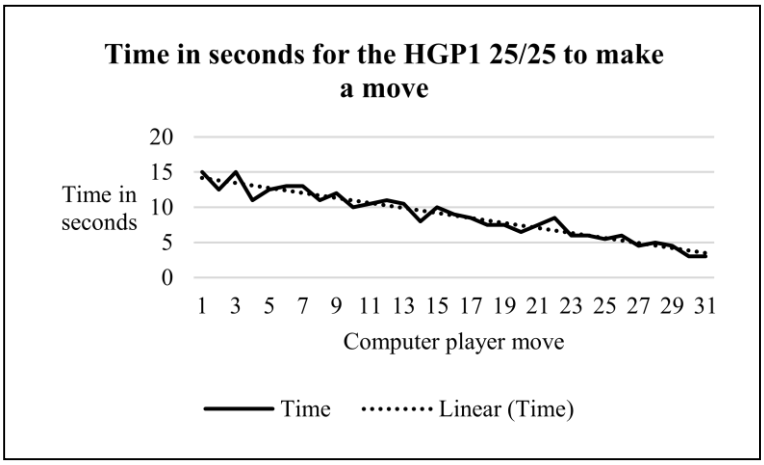


Figure 10.3. Population size of 25 evolved over 25 generations

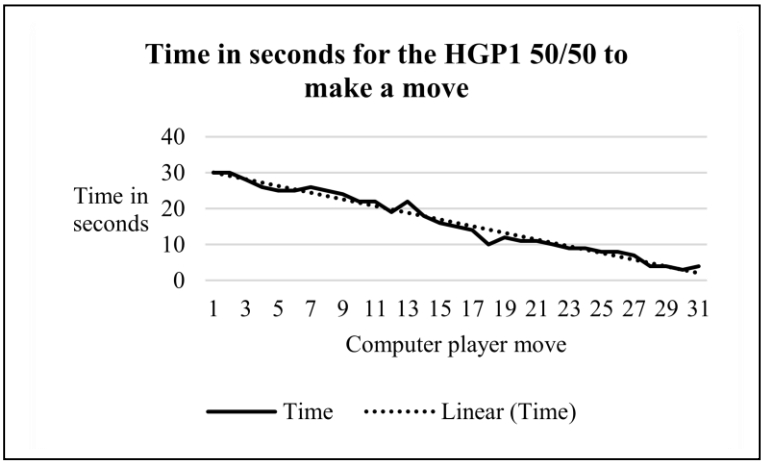


Figure 10.4. Population size of 50 evolved over 50 generations

10.2.2 Analysis of Othello Game Playing Strategy

As the 50/50 version of *RHGP* was found to perform the best, it was decided to study the general game playing strategies evolved by it. The following example is used to illustrate the best performing strategies of the *RHGP*. The *RHGP* player is playing black and the *HGP* player is playing white. The learned values, i.e. potential moves, calculated from the evolved heuristics incorporating reinforcement learning are indicated by the numerical values. The arrows indicate the best move to be made based on these values.

Figure 10.5 depicts the opening game. At the start of the game the corners and edges are considered less important and are assigned low heuristic values. Positions that could lead to the capture of strategic edge positions and corners are assigned high heuristic values as shown in Figure 10.5. It is important to note that the inner board positions are already considered high priority positions to be captured by black. After black has made the first move the board status will be re-evaluated and new heuristic values assigned.

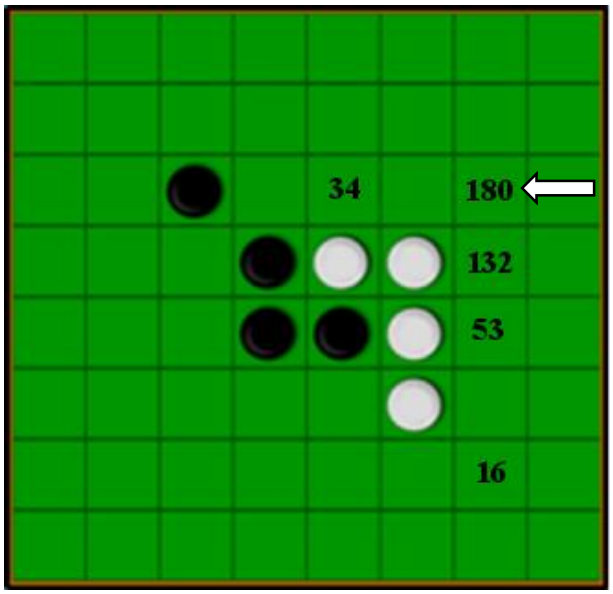


Figure 10.5. Othello opening game

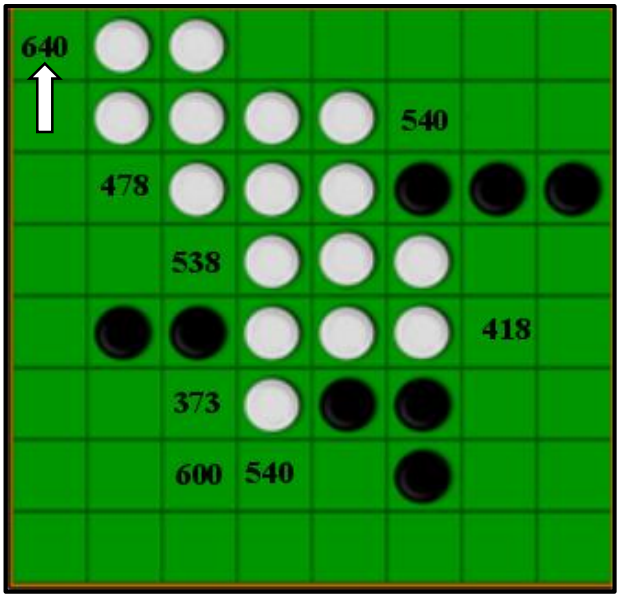


Figure 10.6. Othello middle game

In the middle-game, Figure 10.6, the corners, diagonal and bounding edge positions become more important and higher heuristic values are assigned to them. During the middle to endgame, the *RHGP* player secures the top left quadrant by taking the top left corner and immediately begins strategizing within the bottom right quadrant. As play converges towards the bottom right of the board, the inner edge and middle diagonal positions are considered high priority positions and are assigned high heuristic values.

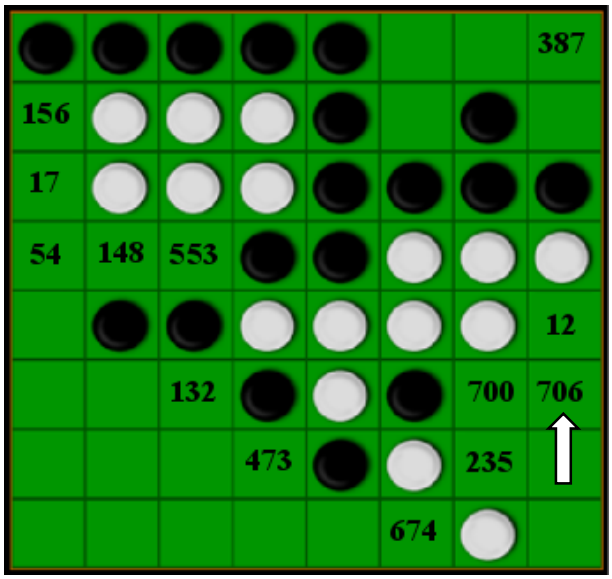


Figure 10.7. Othello corner defense

This is crucial for black as capturing any of those positions will eventually force white to capture positions directly next to the corner square as shown in Figure 10.7. This will allow the *RHGP* player to capture the bottom right corner thus gaining a distinct advantage over white. By securing strategic inner edge positions black is able to dominate the edges and corner positions, winning three of the corner positions as well as isolating white's disks in the top right quadrant.

At the endgame, black has secured the top left quadrant of the board as well as securing three of the four corners, leaving white no option but to play the weaker board positions. The *RHGP* computer player defeats the *HGP* player by 49 discs to 15 as shown in Figure 10.8.

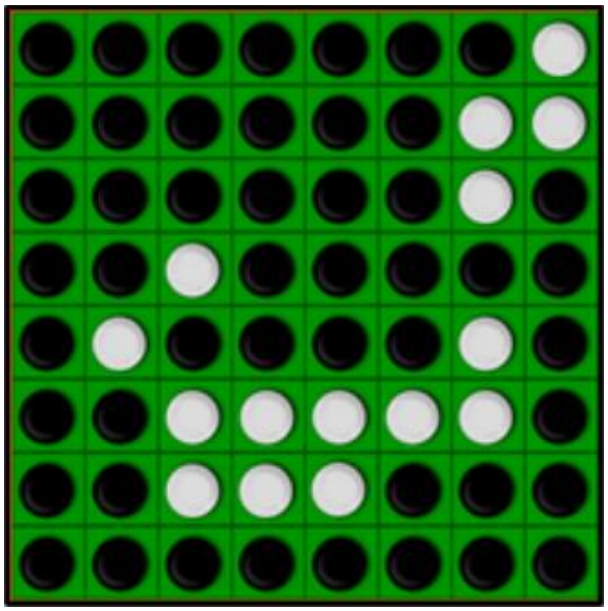


Figure 10.8. Black wins 49 discs to whites 15

The *RHGP* player, playing black, was able to strategically defend the corners and play diagonal and edge tactics to the disadvantage of the *HGP* player, playing white. Black fragmented white's defense through clustering and good middle-game tactics. This illustration shows that the *RHGP* is able to evolve improved Othello strategies over the *HGP*. A noteworthy observation in many of the games was that for most of the beginning and middle-games the *RHGP* trailed in the number of discs, often sacrificing discs gained in order to occupy strategic positions. Available corner positions were often assigned lower weight values while priority was given to other key strategic positions in defense of the corners and edges. This strategy inevitably led to the *RHGP*'s victory.

10.2.3 Othello Conclusion

The research presented in this section investigates the use of reinforcement learning and mobility strategies in order to improve the Othello game playing strategies evolved by GP. The performance of the GP approach is compared to pseudo-random move players, the GP approach incorporating reinforcement learning, the GP approach incorporating mobility and the GP approach incorporating both reinforcement learning and mobility. Both reinforcement learning and mobility strategies were found to improve the performance of the evolved game playing strategies, with reinforcement learning producing better results than the mobility. Reinforcement learning was found to further strengthen the evolved players' game playing ability. The single

move look ahead mobility technique worked well in most cases but in others worked against the GP. This study has revealed the potential of this innovative GP approach for evolving good game playing strategies for games of low complexity such as Othello.

10.3 Results of the GP Approach for Checkers

This section reports on the performance of the heuristic based GP approach in inducing game playing strategies for checkers and compares the performance with reinforcement learning incorporated into the approach. To determine the quality of the GP evolved strategies and ascertain the effectiveness of incorporating reinforcement learning into the genetic programming approach for checkers, two experiments were performed.

Experiment 1: This experiment compares the performance of the heuristic based genetic programming approach, without reinforcement learning, to a pseudo-random move player encoded with limited checkers game playing strategies. These encoded strategies prioritize moves that result in single pieces being promoted to King status and selecting pieces that are in a position to capture enemy pieces. The pseudo-random move player in this experiment is referred to as *RPlayer*. The heuristic based genetic programming approach is referred to as *HGP*. The performance of the *HGP* will be compared to the *RPlayer* simply to determine whether the *HGP* can demonstrate improved levels of playability.

Experiment 2: In this experiment, the performance of the *HGP* is compared to the heuristic based GP approach incorporating reinforcement learning. The GP approach incorporating reinforcement learning is designated *RHGP*.

In both experiments, 10 games were played with each player alternating between black and white per game. Five games as black, thus moving first, and five games as white. Ten games proved to be sufficient for purposes of evaluation. The score, 1 point for a single piece and 2 points for a King, and the number of single pieces promoted to Kings during play, for each game was recorded for each experiment. A game was determined to be at an end when one of the following conditions was reached:

- No more pieces of one colour were left on the board.

- One colour was unable to move.
- No pieces were taken or promoted to Kings in the preceding 50 moves.
- The number of moves in the game, set by the parameter value *game moves*, exceeded the set value.

Each experiment was carried out using random seed values, with the GP and algorithm parameters depicted in Chapter 9, Table 9.1. The time taken for the GP player to make a move was recorded for each game. A comparison of the performance of *HGP* and *RHGP* is firstly presented in subsection 10.3.1, followed by an analysis of some game playing strategies evolved by *RHGP* in subsection 10.3.2.

10.3.1 Performance Comparison

The proposed GP approach was able to successfully evolve sound game playing strategies for checkers in under a minute, i.e. less than 1 minute per game move. The HGP outperformed the pseudo-random move players in Experiment 1, winning all of the games. Experiment 2 revealed that game playing strategies were improved through the incorporation of reinforcement learning.

Experiment 1 – *HGP* vs. *RPlayer*: *HGP* was found to outperform the pseudo-random move players, winning 10 out of the 10 games as shown in Figure 10.9. All of the games resulted in the *HGP* capturing every *RPlayer* piece. This is anticipated as the heuristic representation used for *HGP* is aimed at providing sound strategies for the checkers player. *HGP* promoted a significantly greater number of single pieces to Kings in each game. This is depicted in Figure 10.10. *HGP* promoted 39 single pieces to Kings whereas *RPlayer* promoted 6 single pieces to Kings. Each game lasted no more than 50 moves.

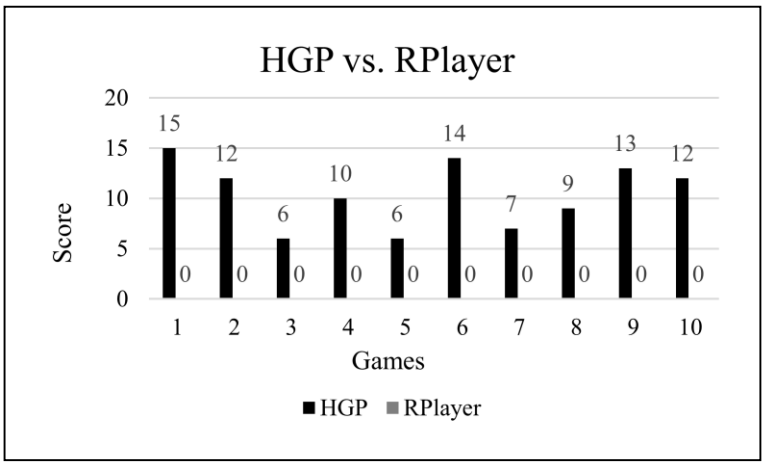


Figure 10.9. Games won

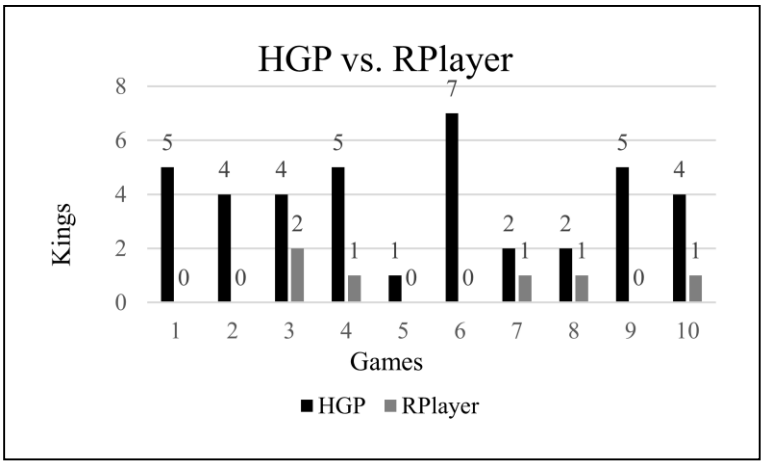


Figure 10.10. Kings achieved

Experiment 2 –*HGP* vs. *RHGP*: *RHGP* performed better than *HGP* winning 7 of the 10 games and drew 1 game as shown in Figure 10.11. *RHGP* promoted 33 single pieces to Kings and *HG*P promoted 19 single pieces to Kings as shown in Figure 10.12.

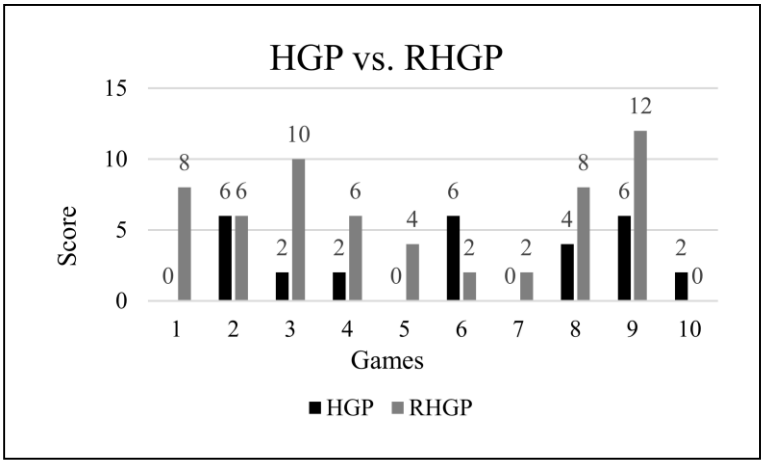


Figure 10.11. Games won

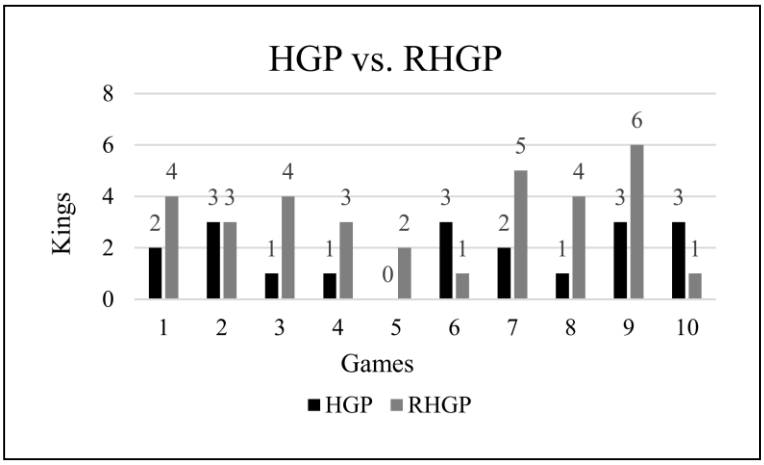


Figure 10.12. Kings achieved

It can be noted that the second game ended in a tie, that is before one colour captured all of the pieces of the other colour (*no take rule*) [125], indicating that both the *RHGP* and *HGP* were able to evolve defensive strategies, preserving their Kings. The *RHGP* gained a sum of 58 points over 10 games and the *HGP* gained 28 points.

10.3.2 Analysis of Checkers Game Playing Strategies

This section outlines four important strategies that were evolved by the *RHGP* playing against the *HGP*. In these examples, the *RHGP* is playing black and the *HGP* is playing white. The arrows indicate the direction in which a selected piece will move.

Strategy 1: This noteworthy strategy was evolved by *RHGP* during the start game creating a solid defense against *HGP*. The start game in checkers is usually considered *the battle for mobility* [124]. Advancing pieces into strategic positions from the start is often guaranteed to result in the promotion of single pieces to Kings and the prevention of your opponent gaining Kings. Figure 10.13 illustrates a typical example of a strategy evolved by the *RHGP* in which each rank (row of four pieces) is sequentially advanced in unison. The black back rank (B) in particular is noted to form part of the maneuver, unlike in the case of the *HGP*, in which the white back rank (W) has remained in position resulting in fragmentation of the front ranks.

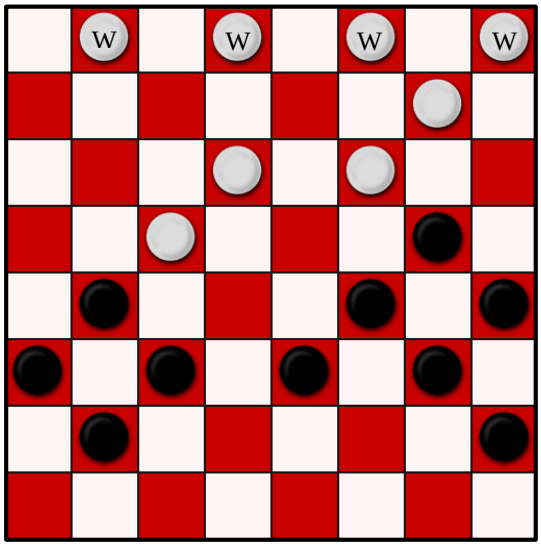


Figure 10.13. RHGP evolved strategy 1

Strategy 2: This second strategy was evolved by *RHGP* during the early stages of the middle game against the HGP. The *RHGP* is noted to adopt a powerful strategy defending territory and pieces as shown in Figure10.14. The X's form central points around which the *RHG*P strategy builds a four man formation. In the game of checkers, this is an almost impenetrable formation,

and if successfully maneuvered to the King side of the board, will result in the promotion of single pieces to Kings.

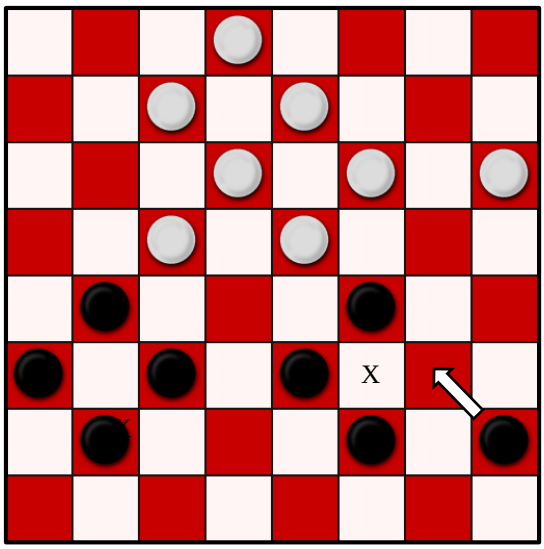


Figure 10.14. RHGP evolved strategy 2

Strategy 3: A third strategy evolved by the RHGP favoured the promotion of single pieces to Kings. It was noted, however, that Kings often remained on the *King Rank*, and only moved to allow passage for a single piece into the rank. This is illustrated in Figures 10.15. and 10.16.

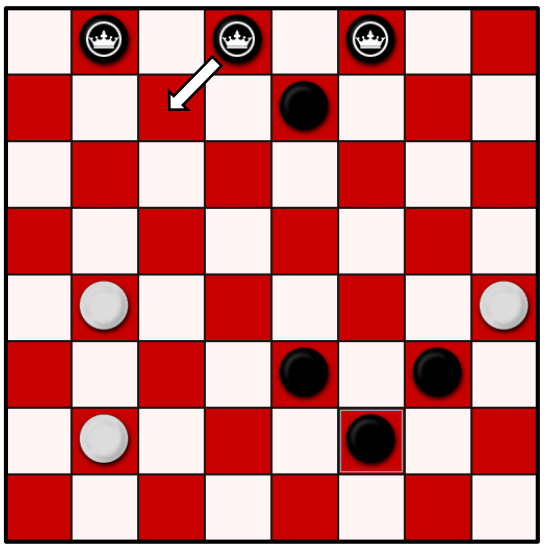


Figure 10.15. RHGP evolved strategy 3

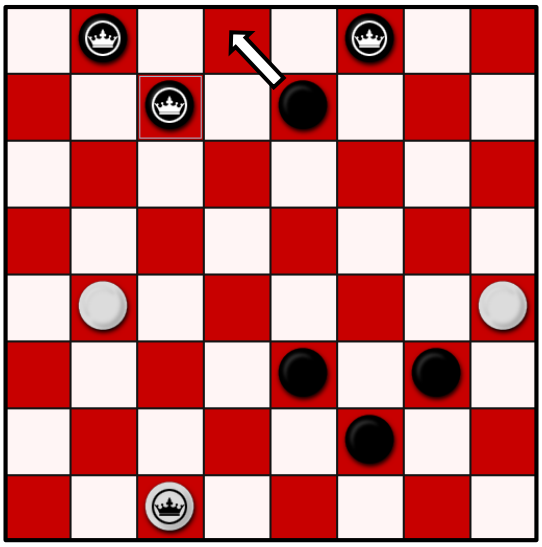


Figure 10.16. RHGP evolved strategy 3

Strategy 4: A fourth strategy evolved by the *RHGP* and in this case also evolved by the *HGP* resulted in Kings from both sides simply moving backwards and forwards between ranks as shown in Figure10.17. This was observed to occur in the second game played between the *RHGP* and the *HGP*, which resulted in a draw. Unlike strategies adopted by humans, in which the aim is to gain the upper hand resulting in a potential win, this defensive strategy seems intuitive of the evolved learning process to preserve Kings disregarding the fundamental aim of winning. In this case, games were ended based on the *no take rule* [125].

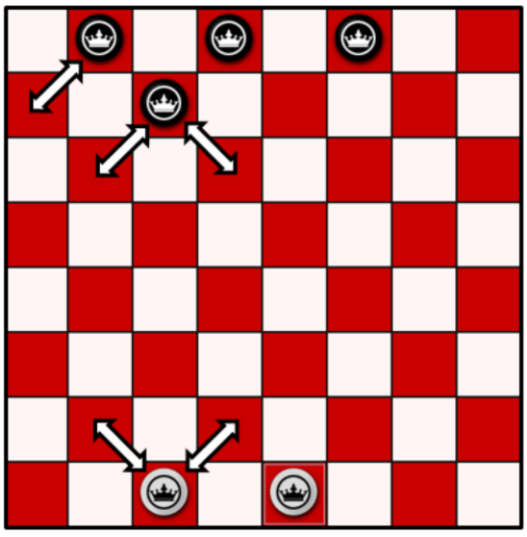


Figure 10.17. RHGP evolved strategy 4

10.3.3 Checkers Conclusion

The aim of the study presented in this section for checkers is twofold. Firstly, the heuristic based GP approach for evolving game playing strategies for checkers is evaluated. Secondly, the effectiveness of improving the evolved strategies using reinforcement learning is investigated. The evolved strategies were tested against pseudo-random players and were found to outperform these players. Furthermore, the use of reinforcement learning to improve the evolved strategies resulted in the generation of better game playing strategies, which performed better than the evolved players applied as is. This study has revealed the potential of both the heuristic based approach and the incorporation of reinforcement learning for inducing good game playing strategies for medium complexity games such as checkers.

10.4 Results of the GP Approach for Chess

This section reports on the performance of the heuristic based GP approach in inducing game playing strategies for chess endgames and compares the performance of the GP evolved chess strategies with increased numbers of training players. Training players in this study are used to generate sub-sets of potential moves from which the best move is selected by the GP player to make a game move. This is described in Chapter 9, section 9.2.2.1. A discussion of the performance of the approach is presented firstly followed by an analysis of some of the game playing strategies typically evolved by the GP approach.

To determine the quality of the GP evolved strategies and the effectiveness of incorporating real-time learning into the genetic programming approach for chess, three experiments using three different endgame configurations were conducted. The GP player played white for all three experiments against a pseudo-random move player encoded with limited chess playing strategies. These encoded strategies allowed the pseudo-random move player to prioritize moves that would result in the opposing King being checked or checkmated, movement of pieces out of danger and taking of enemy pieces.

Experiment 1: This endgame configuration is a middle to endgame configuration comprising several chess pieces of differing values. This configuration is illustrated in Figure 10.18.

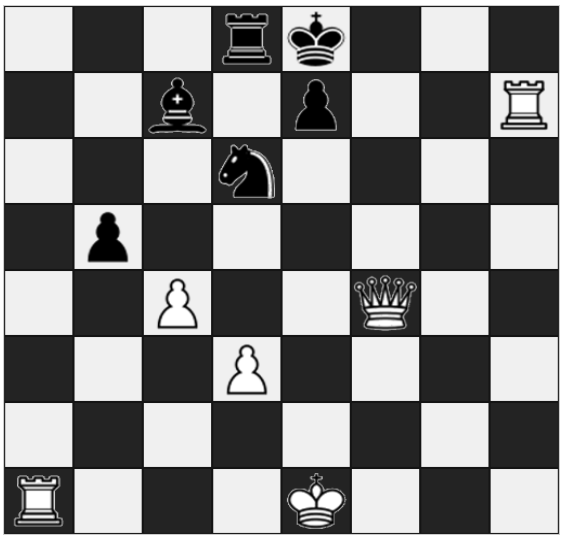


Figure 10.18. Experiment 1 endgame configuration

Experiment 2: This endgame is a classical endgame configuration [91, 50, 139]. A black Bishop was added to the endgame configuration for experimental purposes in order to increase the playout challenge. This configuration is illustrated in Figure 10.19.

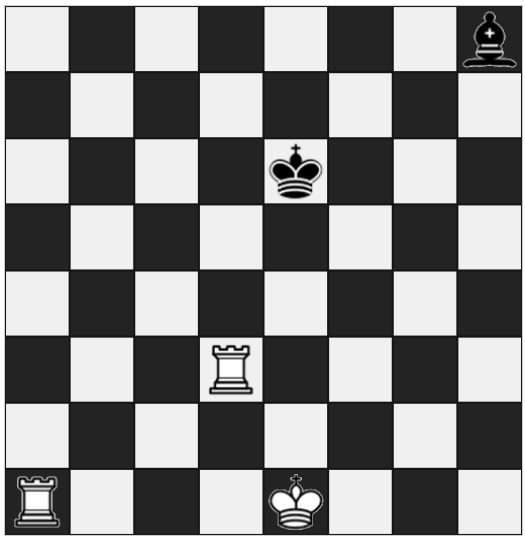


Figure 10.19. Experiment 2 endgame configuration

Experiment 3: This endgame configuration is the classic Rook-King-Rook endgame and is considered one of the most difficult for an AI agent to accomplish [50, 139]. This configuration is illustrated in Figure 10.20.

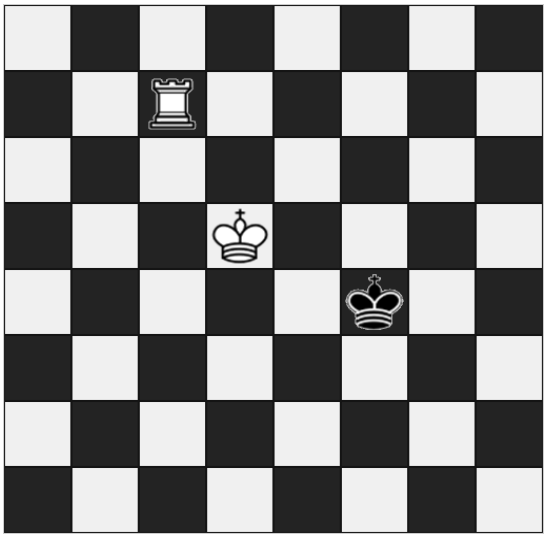


Figure 10.20. Experiment 3 endgame configuration. A classic endgame configuration and one of the more difficult for an AI player to accomplish

To determine the effect of incorporating an increased number of training players, each of the three experiments were carried out applying 1, 5, 10, 15, 20 and 25 training players (Chapter 9, section 9.2.2.1). For each experiment, tournaments of five games per training player application were played out between the GP player and a pseudo-random move player. In total, 90 endgame playouts were analyzed, that is 30 playouts per experiment. The successes of the GP player for each number of training players was determined by the sum points, depicted in Table 10.1. In addition to scoring the GP's playing performance by means of the number of games won, it was determined through experimentation, that a better measure could be achieved by adding additional points to the performance score based on key chess strategies. These scores were then used to assess the best performing players evolved. Each experiment was carried out using random seed values, with the GP algorithm parameters depicted in Chapter 9, Table 9.1.

Table 10.1. Point system to calculate the strength of the GP player

(A) Game result			**(B) Good strategy evolved** *			**(C) Friendly pieces captured**		**(D) Enemy pieces captured**		**(E) Defense of major or moved pieces** *		
win	draw	loose	yes	?	no	yes	no	yes	no	yes	?	no
1	0	-3	1	0	-1	-1	1	1	0	1	0	-1

* *Although these attributes are subjective, their values can easily be determined through careful analysis of the endgame.*

10.4.1 Performance of the GP Approach

The results of the three experiments, each conducted with different endgame configurations, have revealed that the GP evolved game playing strategies are improved through an increase in number of training players. The number of games won by the GP players increased as the number of training players increased. All other games that were not won by the GP players were played to a draw applying the fifty-move rule, which is discussed in Chapter 7, section 7.4 or a stalemate. A stalemate occurs when a King is maneuvered into a position where it is not in check but cannot be moved. All other pieces of that colour are in positions where they cannot be moved. This concludes the game and for the purpose of this study, a draw was assigned to the match.

The following presents the results and discussion of the results for each of the three experiments.

Experiment 1: For this experiment, the performance strength of GP evolved players was observed to increased as the number of training players increased. These results are shown in Table 10.2. Points allocation, (A) – (E), are indicated in Table 10.1.

Table 10. 2. Experiment 1: Performance strength of GP player with increasing number of training players

Training Players	**(A)**	**(B)**	**(C)**	**(D)**	**(E)**	**Performance Strength of GP Player**
Experiment 1						
1	2	-12	-1	5	-5	**-9**
5	3	-6	1	3	-5	**-4**
10	5	5	3	1	-2	**12**
15	5	5	5	4	3	**22**
20	3	6	2	5	1	**17**
25	5	12	5	4	5	**31**

As the number of training players increased, the GP players were able to accomplish this endgame configuration through an increasing number of wins. This is illustrated in Figure 10.21.

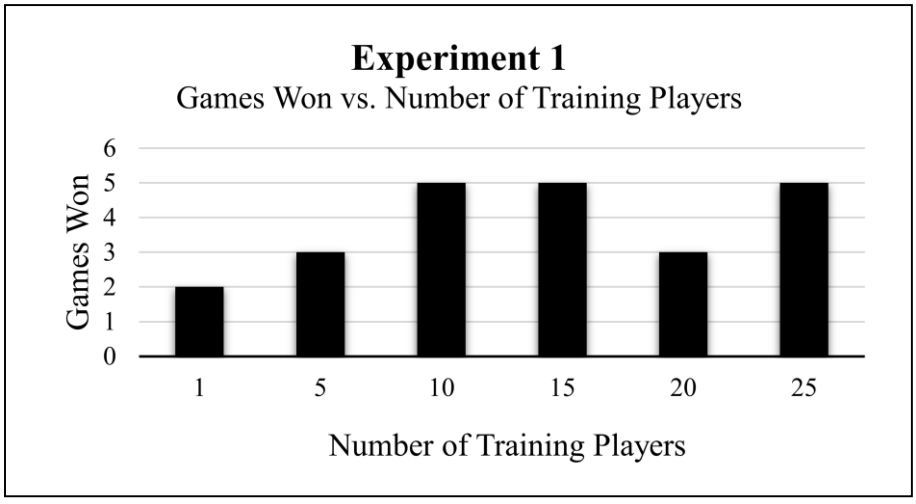


Figure 10.21. Games won

With 1 to 5 training players, the GP players moved randomly about the board, demonstrating limited strategy. With 10, 15 and 25 training players, the GP players were able to win all 15 games by checkmating the black King demonstrating good game playing strategies. The drop in

performance of the GP player's from 22 to 17 with 20 training players, shown in Figure 10.22, is due to two games ending in a stalemate for Black.

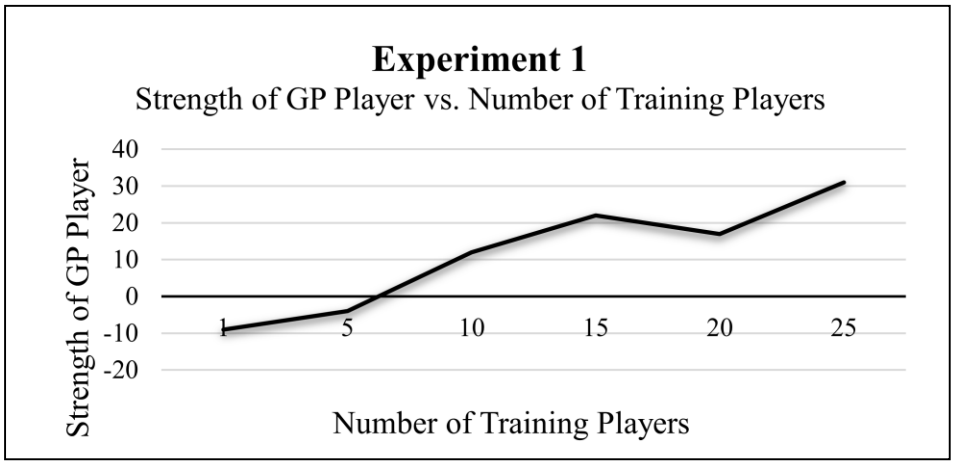


Figure 10.22. Experiment 1: Performance strength of GP player with increasing number of training players

The following subsection gives an analysis of the best strategy evolved by GP for endgame configuration one.

10.4.1.1 Analysis of the best evolved strategy for endgame configuration one

This configuration, comprising several chess pieces of differing values, is a middle to endgame configuration and can be easily accomplished by an AI player. This is the best strategy evolved by GP with 25 training players. The GP player in this example is white and moves first. The yellow arrows indicate the movement of a white piece from one position to the next. A red arrow indicates the movement of a black piece. A blue arrow indicates positions that are blocked by an opposing piece, in this case the white Queen show in Figure 10.28. In this board configuration, the black King is protected by two major pieces, the Rook and Bishop, and any direct attack on the King will force it to move to safety opposite the black Bishop. However, the GP player, playing white in this example, performs an indirect attack on the black King, executing a Rook-Rook maneuver to checkmate the black King. This maneuver is initiated by the white Rook attacking the black Bishop and trapping the King on the black King's rank, Figure 10.24.

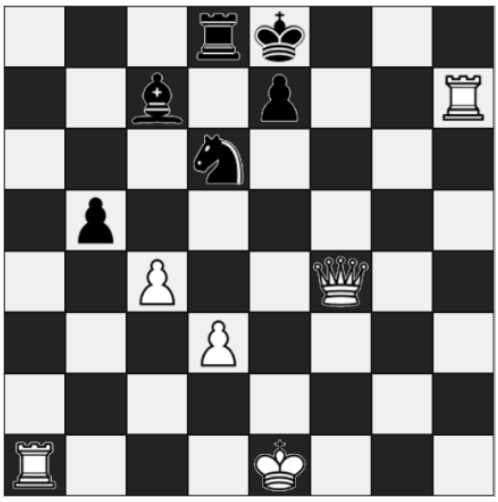


Figure 10.23. Endgame configuration one

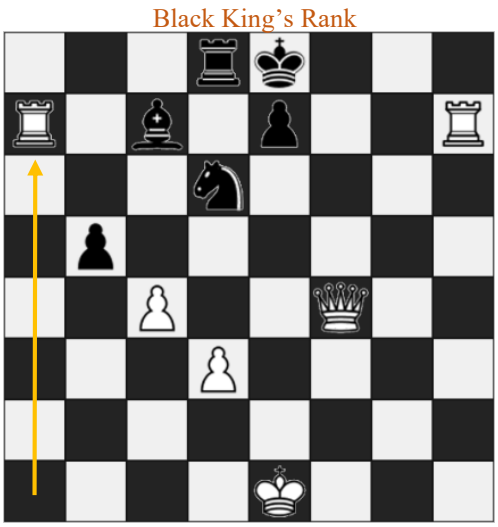


Figure 10.24. White move its Rook to position an attack on the black Bishop.

The black player responds to this move by attacking the white King, Figure 10.25. The white Rook is in a strong position and does not take the black Bishop. Instead, the GP player simply moves the white King out of danger, Figure 10.26.

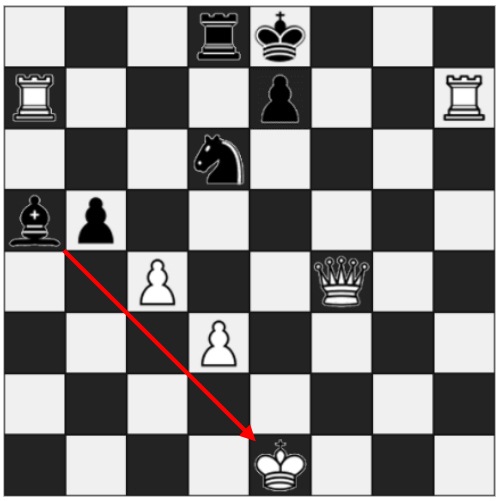


Figure 10.25. Black moves out of danger attacking the white King

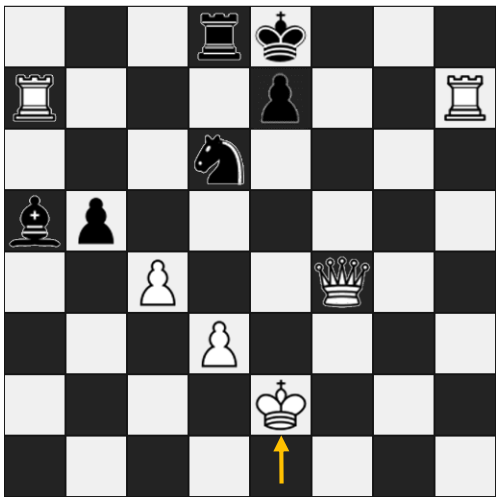


Figure 10.26. The white King moves out of danger

The black Bishop moves out of danger from attack by the white Rook, setting white up to checkmate the black King, Figures 10.27 and 10.28.

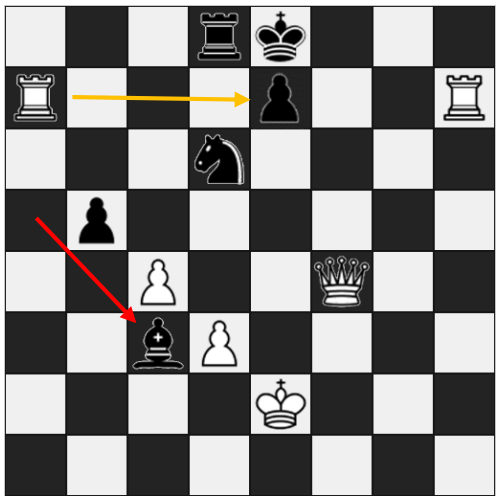


Figure 10.27. The black Bishop's offensive move sets white up to checkmate the black King

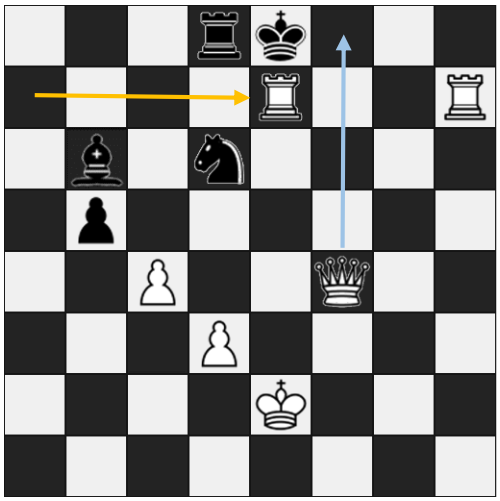


Figure 10.28. White accomplishes this endgame through a combined Rook-Rook strategy

It can be noted the throughout this maneuver the white Queen is in a very strong position and is not selected to move at all. The GP player accomplishes this endgame by executing a *Rook-Rook* maneuver to checkmate the black King. Figures 10.24 to 10.28 demonstrate that GP is able to evolve complex strategies that allow pieces to be maneuvered into key positions in order to checkmate the King.

Experiment 2: For this experiment, the performance strength of the GP players increased as the number of training players increased. This is illustrated in Figure 10.29. The results of increased number of training players is shown in Table 10.3. Points allocation, (A) – (E), are indicated in Table 10.1.

Table 10.3. Experiment 2: Performance strength of GP player with increasing number of training players

Training Players	**(A)**	**(B)**	**(C)**	**(D)**	**(E)**	**Performance Strength of GP Player**
Experiment 2						
1	0	-9	4	2	-5	**-8**
5	1	-5	3	5	-4	**0**
10	1	-1	5	5	-2	**8**
15	2	1	6	5	-2	**12**
20	3	4	5	2	3	**17**
25	4	5	5	4	4	**22**

Results show that as the number of training players increased, the GP players were able to accomplish this endgame configuration through an increasing number of wins, illustrated in Figure 10.30.

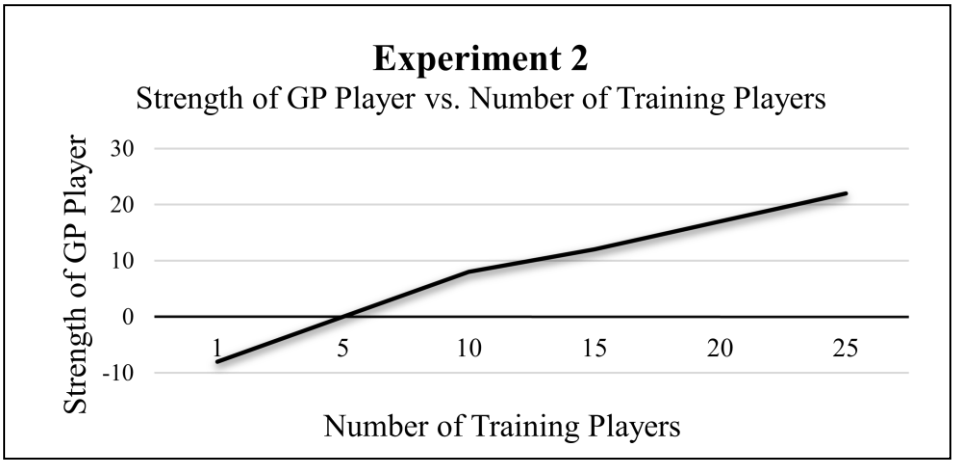


Figure 10.29. Experiment 2: Performance strength of GP player with increasing number of training players

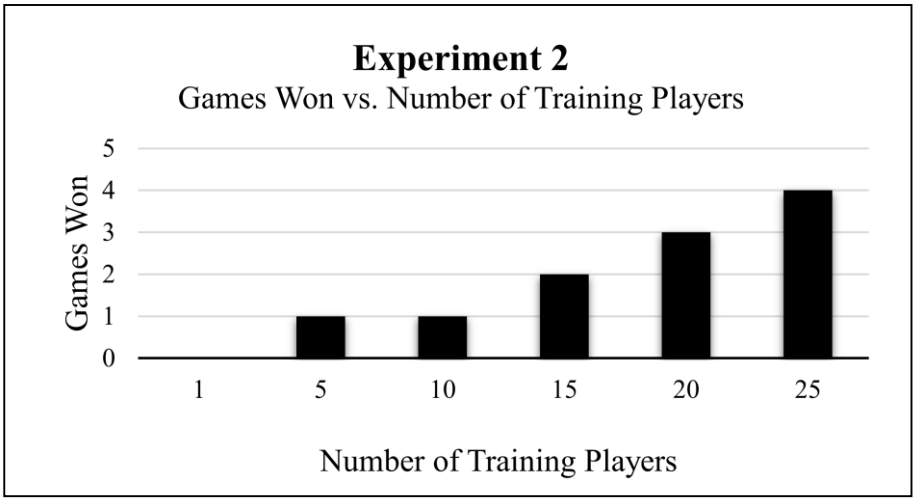


Figure 10.30. Games won

When using 10 to 15 training players, the GP players were always able to defend their Rooks and King but did not accomplish checkmating the black King in all of the games. With 20 to 25 training players, the GP players were increasingly able to checkmate the black King by successfully maneuvering it to the edge of the board using the classic *KRK* maneuver. With 25 training players the GP players won 4 out of the 5 games.

The following subsection gives an analysis the best strategy evolved by GP for endgame configuration two.

10.4.1.2 Analysis of the best evolved strategy for endgame configuration two

This is a difficult endgame configuration as the GP player has to determine the value of capturing either the rank in front or behind the defending King in order to maneuver the King into checkmate. These ranks are indicated by the yellow lines illustrated in Figure 10.31. Furthermore, the white Rook is in danger of being captured by the black Bishop. The following example is the best strategy evolved by GP with 25 training players. The GP player in this example is white and moves first. The white Rook moves out of danger and immediatly checks the black King, Figure 10.32.

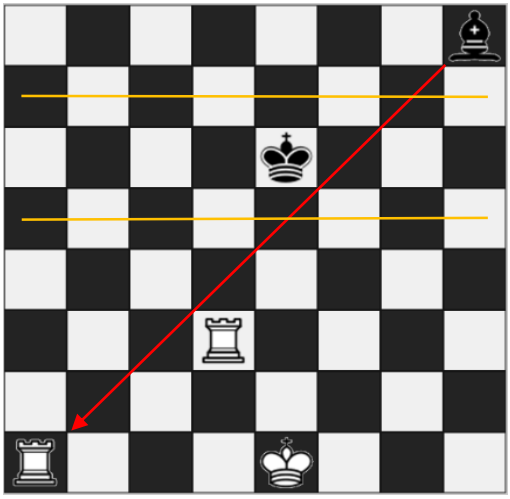


Figure 10.31. Endgame configuration two

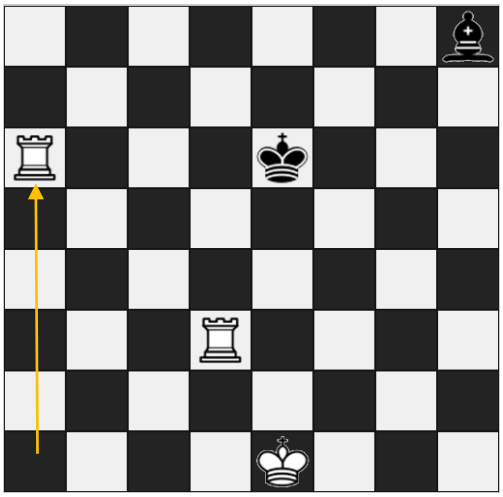


Figure 10.32. The white Rook moves out of danger to check the black King

The black King moves out of check, Figure 10.33. In the next two moves, the GP player moves to check the black King. The black player chooses to move the King downwards out of check but becomes trapped between the two white Rooks, Figures 10.32 and 10.33. In the next move, the black King attacks a white Rook, Figure 10.34. The white Rook immediately moves out of danger and attacks the black Bishop.

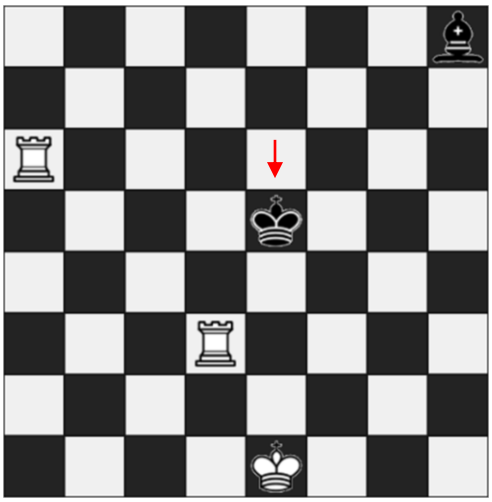


Figure 10.33. The black King moves out of check

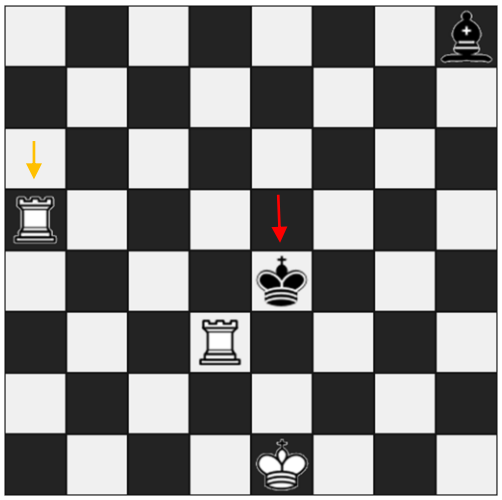


Figure 10.34. The black King attacks the white Rook

The black Bishop moves out of danger allowing the white Rook to move onto the same rank as the black King, forcing the King to move one rank down towards the white King, Figures 10.35 and 10.36. This move is important and immediately give the GP player an upper hand for the black King cannot move out of the Rooks trap, Figure 10.36.

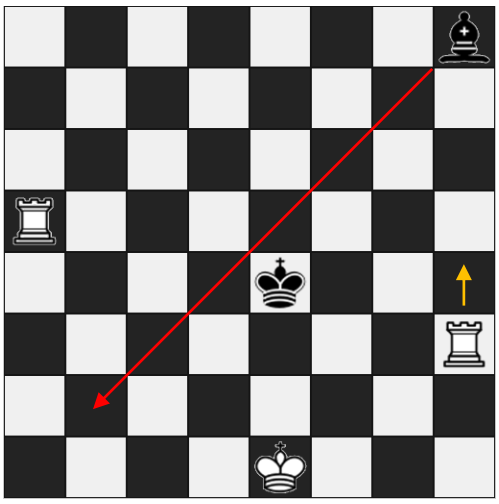


Figure 10.35. The white Rook traps the black King as the black Bishop moves into an offensive position

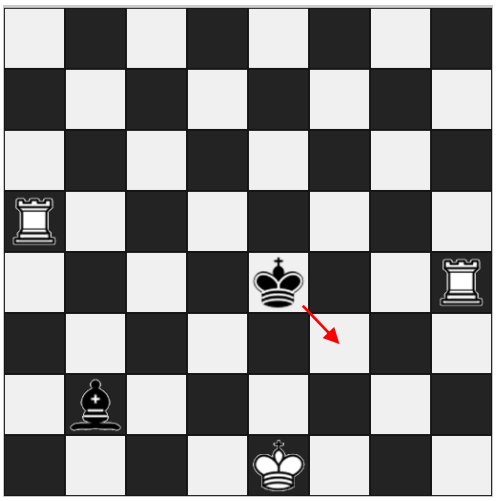


Figure 10.36. The black King is forced to move towards the white King's rank

The left hand side white Rook moves into a strong position blocking the enemy King from attacking the second white Rook, Figure 10.37.

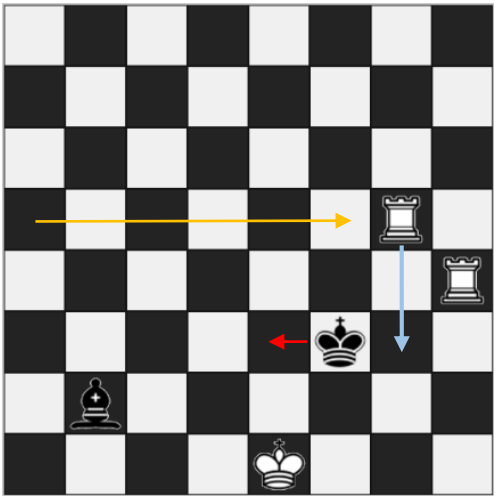


Figure 10.37. The black King is setup to be checkmated

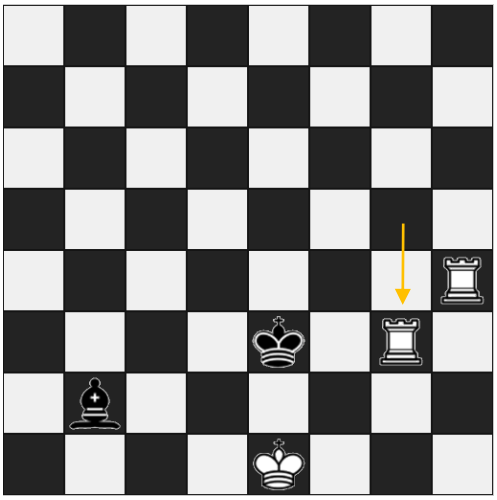


Figure10.38. The white Rook checkmates the black King

Checkmate is eminent and the black King has no other move option but to move away from the white Rooks, Figure 10.37. The black Bishop cannot prevent this checkmate. The GP player checkmates the black King by moving the white Rook onto the same rank as the enemy King. Figures 10.32 to 10.38 demonstrate somewhat complex strategies evolved by GP in order to maneuver the defending King into checkmate, as well as strategies to maneuver friendly pieces out of danger. Furthermore, it may be noted, that strategies to attack the black Bishop did not play a part in the GP player's overall game strategy as attempting to check or checkmate the black King dominated the game play.

Experiment 3: This endgame configuration proved to be the most challenging for GP evolved players as they only succeeded in checkmating the black King twice in 30 endgames, once with 20 training players and once with 25 training players. The number of games won is depicted in Figure 10.39.

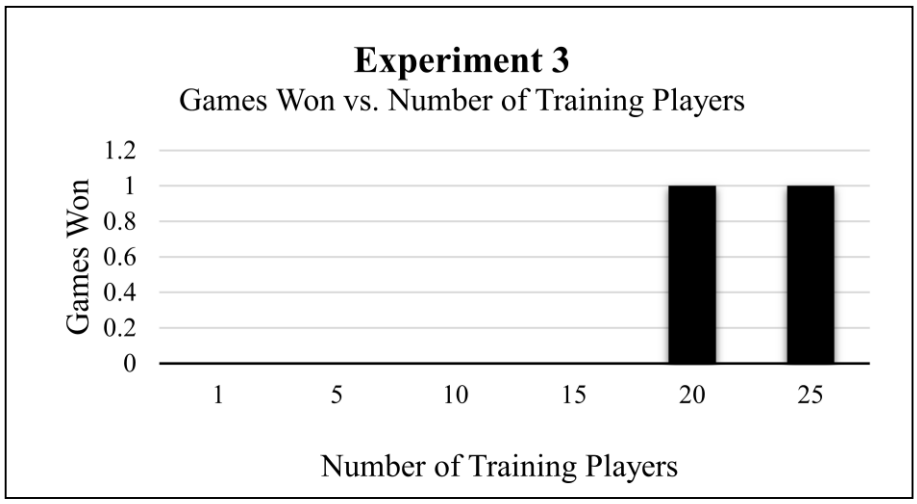


Figure 10.39. Games won

When using 1 to 15 training players GP was unable to evolve strategies that allowed the GP players to checkmate the black King. These results are shown in Table 10.4. Points allocation, (A) – (E), are indicated in Table 10.1.

Table 10.4. Experiment 3: Performance strength of GP player with increasing number of training players

Training Players	**(A)**	**(B)**	**(C)**	**(D)**	**(E)**	**Performance Strength of GP Player**
Experiment 3						
1	0	0	N/A	0	0	**0**
5	0	0	N/A	0	0	**0**
10	0	0	N/A	0	0	**0**
15	0	0	N/A	0	0	**0**
20	1	2	N/A	0	0	**3**
25	1	2	N/A	0	0	**3**

The performance strengths of the GP players for this endgame configuration only increased with 20 and 25 number of training players. This is illustrated in Figure 10.40.

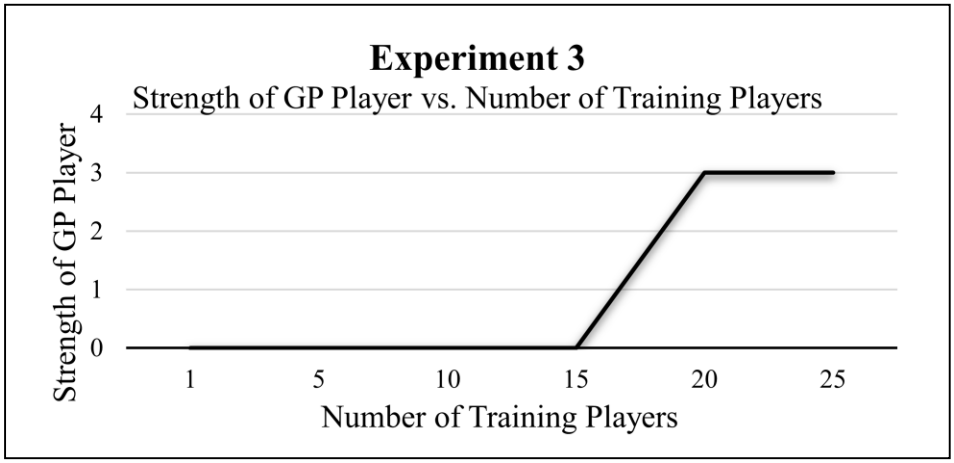


Figure 10.40. Experiment 3: Performance strength of GP player with increasing number of training players

The following subsection gives an analysis of the two winning strategies evolved by the GP for endgame configuration three.

10.4.1.3 The best strategy for endgame configuration three

The GP player in this sub-section is white and moves first. Upon careful analysis of this endgame configuration playout, it was observed that the white Rook, being the only other major piece on the board apart from the King, was always selected to make the move. This is anticipated as the white Rook, in most cases, can be moved to a position resulting in the black King's check. This move is to white's advantage. However, due to the small number of pieces on the board, the black King is able to simply move out of check. This resulted in a perpetual "dog fight" between the white Rook attacking the black King and the black King moving out of danger.

Analysis of the two endgames playouts that resulted in a checkmate revealed that strategies involving moving the white King were those that resulted from the white Rook not being able to check the black King directly. This is illustrated in Figure 10.41.

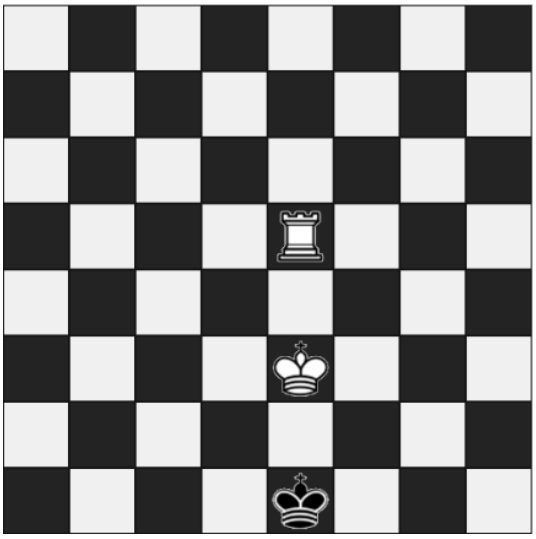


Figure 10.41. The white Rook unable to check the black King directly

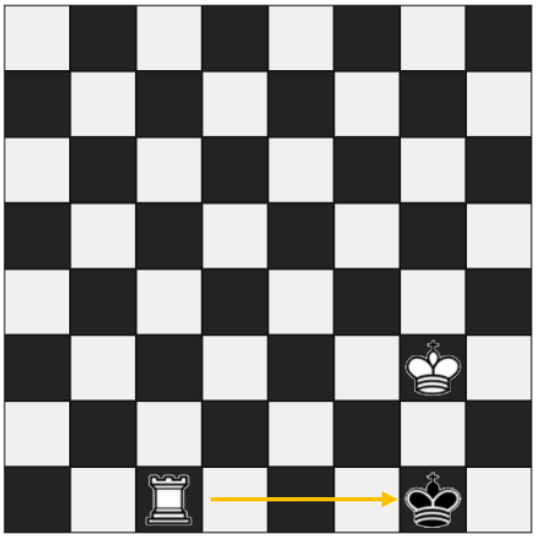


Figure 10.42. A combined King-Rook strategy to checkmate the black King

In this instance, the white King was able to keep the black King on the edge of the board. A combination of King and Rook maneuvers eventually resulted in the checkmate of the black King as shown in Figure 10.42.

10.4.2 Performance Comparison

The time taken for the GP players to make a move in all three experiments increases as the number of training players increased. With one training player the GP took under 30 seconds to make a move. With 25 training players, the GP players took up to 4 minutes to make a move, which is twice the time required to make a move in tournament chess. The average performance strength of GP evolved players for all three experiments, illustrated in Figure 10.43, shows that performance strength increased with an increase in the number of training players. This increase in performance strength produced strategies that were good enough to checkmate the black King. For experiments 1 and 2, the number of wins achieved by the GP players increased with an increase in the number of training players. The GP players won only two games in experiment 3, one with 20 training players and the other with 25 training players.

Figure 10.43 shows the average strengths of GP evolved players with increased numbers of training players for all three experiments. Figure 10.44 shows the average number of games won by GP evolved players with increased numbers of training players for all three experiments.

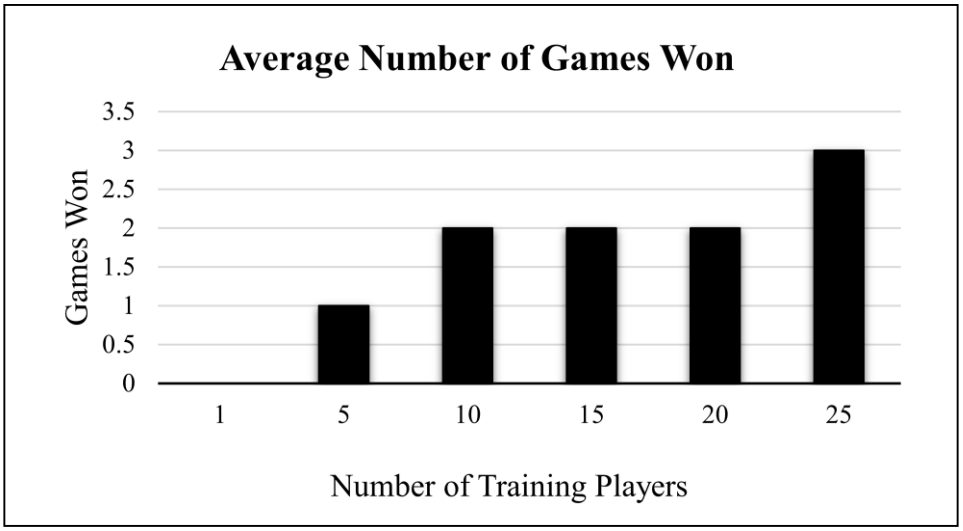


Figure 10.43. Average number of games won with increasing numbers of training players

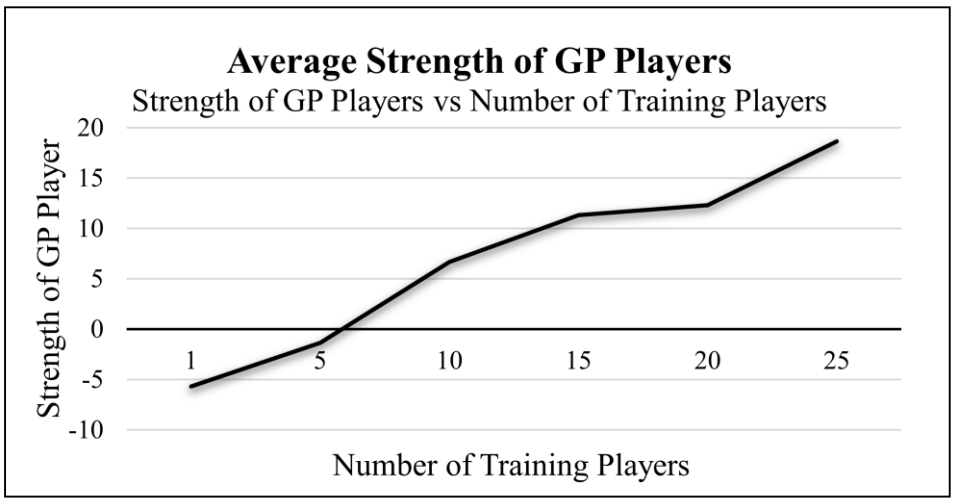


Figure 10.44. Average performance strength of the GP players with increasing numbers of training players

These results show that for all three experiments, the performance of the GP players increased with an increase in the number of training players resulting in an increase in the number of games won.

10.4.3 Chess Conclusion

The research presented in this section investigates the use of GP for evolving game playing strategies for chess endgames. This work differs from other research in the field in that; firstly, the strategies are induced during real time game play. Secondly, the approach is heuristic based and combines these heuristics with *real time learning* in an attempt to enhance the evolved game playing strategies. From the results it can be concluded that with increased numbers of training players, the performance of the GP players subsequently improve. Furthermore, the strategies that are induced by the GP are compatible with classic endgame strategies used to checkmate the opposing King. The three experiments did however show that the strength of the GP players decreased as the number of pieces on the board decreased. Experiment 3, which comprised of an endgame having only the Kings and a Rook on the board, resulted in a "dog fight" between the white Rook and black King. Only two out of the thirty endgame playouts resulted in a win for white. This strongly suggests that a dynamic fitness function, i.e. one that can evolve or adapt during various stages of game play, depending on the number and status of pieces left on the board, should be implemented. Alternatively, different fitness functions could be implemented for game

play during the opening, middle and endgames. This study has revealed the potential of the GP approach for evolving good game playing strategies for games of high complexity such as chess.

10.5 Chapter Summary

This chapter presented the results of the GP approach for evolving game playing strategies for Othello, checkers and chess. An analysis of some of the best evolved game playing strategies for all three games was presented. From the results, it can be concluded that the incorporation of reinforcement learning greatly improves the playing performance of GP evolved players. The GP players for Othello and checkers with reinforcement learning were able to beat their opponent players and make moves in less than a minute. For chess, with training players of 20 and 25 the GP evolved chess players were able to accomplish all three endgame configurations by checkmating the opposing King. The time taken to make moves, however, increased as the number of training players increased. With 25 training players, the GP was taking up to four minutes to make a move. The following chapter, Chapter 11, presents a conclusion to this dissertation and propose future extensions of this research.

Chapter 11

Conclusion and Future Research

11.1 Introduction

This chapter summarizes the findings of this dissertation and provides a conclusion to each of the objectives presented in Chapter 1. Section 11.2 presents each objective and conclusions based on the results presented in Chapter 10. The chapter ends by discussing future extensions to the research presented in this thesis, in section 11.3 and the chapter summary in section 11.4.

11.2 Objectives and Conclusions

The primary aim of this research was to *evaluate genetic programming for evolving heuristic based strategies for board games*. In attaining this aim, the following objectives were met:

- Objective 1: Develop and evaluate the GP approach for Othello.
- Objective 2: Develop and evaluate the GP approach for checkers.
- Objective 3: Develop and evaluate the GP approach for chess.

Based on a critical analysis of previous literature using GP to evolve computer players for various board games (discussed in Chapter 9), it was determined that GP has not been used to evolve heuristic based strategies for board games. These studies show that the primary focus of the research is to implement GP approaches that evolve board evaluation functions to be used in combination with other search techniques to produce intelligent game-playing agents. In addition, game playing strategies are generally created offline in conjunction with large game specific knowledge bases to guide the intelligence of the computer players.

A GP approach was implemented and evaluated for three board games of different complexities, namely, Othello, checkers and chess. Othello representing a game of low complexity, checkers representing medium complexity and chess a game of high complexity. The results presented in Chapter 10 clearly demonstrate that the use of GP for evolving heuristic based game playing strategies in real time for different types of board games was successful. The dissertation objectives and conclusion to each objective is provided in the following subsections.

11.2.1 Objective 1 – Develop and evaluate the GP approach for Othello.

The GP approach was developed and evaluated for Othello. In an attempt to improve the performance of the induced game playing strategies, reinforcement learning and mobility were incorporated into the approach. The performance of the GP approach was compared to pseudo-random move players, the GP approach incorporating reinforcement learning, the GP approach incorporating mobility and the GP approach incorporating both reinforcement learning and mobility. Two versions of the approach were implemented. One using a population size of 25 individuals evolved over 25 generations and the other using a population size of 50 individuals evolved over 50 generations. The latter version outperformed the former version. Increasing the population size produced a greater variety of individuals. Increasing the number of generations allowed the GP algorithm to converge to better candidate solutions. In both cases, the GP approach outperformed the pseudo-random move players demonstrating that the GP approach induced good game playing strategies. Both reinforcement learning and mobility strategies were found to improve the performance of the approach, with reinforcement learning producing better results than the mobility strategies. The single move look ahead mobility method worked well in only some cases. These outcomes demonstrated that good game playing strategies for Othello were produced by the GP approach. Furthermore, the performance of the strategies were greatly improved through reinforcement learning. This study has shown the potential of the GP approach for evolving good game playing strategies for low complexity games such as Othello.

11.2.2 Objective 2 – Develop and evaluate the GP approach for Checkers.

The GP approach was developed and evaluated for checkers. To improve the performance of the induced game playing strategies, reinforcement learning was incorporated into the GP approach. The performance of the approach was compared to pseudo-random move players and the GP approach incorporating reinforcement learning. A population of 50 individuals evolved over 25 generations was used to induce the game playing strategies. The GP approach outperformed the pseudo-random players, winning all ten of the games and promoted 39 single pieces to Kings compared to the 6 Kings achieved by the pseudo-random move players. In addition, the use of reinforcement learning to improve the evolved strategies resulted in the induction of game playing strategies which performed better than the evolved players applied as

is. The GP approach with reinforcement learning won 7 of the 10 games and achieved 33 Kings compared the 19 Kings achieved by the GP approach without reinforcement learning. Analysis of games also revealed that both the GP approach and GP approach incorporating reinforcement learning produced strategies to preserve Kings disregarding the fundamental aim of winning the game. These results have revealed the potential of both the heuristic based approach and the incorporation of reinforcement learning for producing computer players for medium complex board games such as checkers.

11.2.3 Objective 3 – Develop and evaluate the GP approach for Chess.

In previous studies for using GP to evolve strategies for computer chess players the authors chose to investigate chess endgames as opposed to a complete game [19, 140, 130, 91]. Based on these literature reviews, the GP approach was developed and evaluated for chess in three different chess endgame configurations. Two endgames, namely, the Rook-King-Rook and Rook-King-King are classic endgames. The Rook-King-King endgame is considered the hardest endgame configuration for an artificial intelligent agent to accomplish [50, 139]. The initial fitness evaluation method used in this GP approach proved successful Othello and checkers but was unsuccessful for chess so an alternative method was developed. This method implemented training players to generate a sub-set of moves from the total moves available to a strategy at a particular board configuration. The results presented in Chapter 10 clearly show that with increased numbers of training players, the performance of the GP players subsequently improved and the induced strategies are compatible with classic endgame strategies used to checkmate the opposing King. With 20 to 25 training players, for experiment 1 and 2, the GP players were able to win 14 of the 20 games by checkmating the black King demonstrating good game playing strategies. Two of the 20 games were played to a stalemate and four games were played to a draw. For experiment 3, the classic Rook-King-King endgame, the GP players managed to checkmate the opposing King twice, one with 20 training players and the other with 25 training players. In both cases, the King-Rook maneuver was used to force the opposing King to the edge of the board in order to checkmating it. This demonstrated that good game playing strategies were induced with an increase in the number of training players. The other 28 games were played out to a draw. Furthermore, the GP players for all three experiments did not lose a game.

The results of the three experiments did however show that the strength of the GP players decreased as the number of pieces on the board decreased. This suggests that the fitness function used to evaluate the chess computer players during game play could be improved by implementing a dynamic fitness function, alternatively different fitness functions could be implemented for game play during the opening, middle and endgames. In conclusion, this study has shown the potential of the GP approach for evolving good game playing strategies for board games of high complexity such as chess.

11.3 Future Research

Based on the research presented in this dissertation, future extensions of this work will include the following:

- Previous studies have presented the induction of game playing strategies offline using larger populations. Future research will examine combining offline and online induction of game playing strategies as a means of obtaining a balance between deriving strategies appropriate for the current scenario of game play as well facilitating longer GP runs.
- Incorporating reinforcement learning into the approach proved successful for improving the performance of GP evolved strategies. Future work will investigate other options for incorporating reinforcement learning into the evolutionary process as well as developing general game players capable of playing well in more than one type of game.
- For chess, further development will examine implementing the GP approach for entire games as well implementing a dynamic fitness function to improve the performance of GP evolved players.

11.4 Chapter Summary

This chapter provided a summary of the findings of this research, the outcomes of the objectives and how they were fulfilled. Finally, future extensions of this work based on the observations made during this research was presented.

Bibliography

[1] C. Smith, B. McGuire, T. Huang and G. Yang, "The History of Artificial Intelligence", CSEP 590A, Washington, 2006.

[2] E. A. Feigenbaum, "Some challenges and grand challenges for computational intelligence", *Journal of the ACM (JACM)*, vol. 50, no. 1, pp. 32-40, 2003.

[3] J. McCarthy, "What is Artificial Intellegence", Personal website, www.formal. stanford.edu/jmc/index.htmlStanford, 2007.

[4] P. Brey, "Hubert Dreyfus - Human versus Machine", *American Philosophy of Technology: The Empirical Turn*, pp. 37-63, 2001.

[5] H. L. Dreyfus, "Alchemy and Artificial Intelligence", The Rand Corporation, Santa Monica, California, 1965.

[6] S. J. Russell and P. I. Norvig, "Artificial Intelligence: A modern Approach", New Jersy: Alan Apt, 1995.

[7] C. Shannon, "Programming a computer for playing chess", *Philosophical Magazine*, vol. 14, pp. 256-275, 1950.

[8] A. M. Turing, "Computing Machinery and Intelligence", *Mind*, vol. 49, pp. 433-460, 1950.

[9] J. Schaeffer, J. and J. van den Herik, "Games, computers, and artificial intelligence", *Artificial Intelligence*, vol. 134, pp. 1-7, 2002.

[10] A. Benbassat and M. Sipper, "Evolving Board-Game Players with Genetic Programming", *In proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO '11)*, 2011.

[11] O. David-Tabibi, M. Koppel and N. Netanyahu, "Expert-Driven Genetic Algorithms for Simulating Evaluation Functions", *Genetic Programming and Evolvable Machines,* vol. 12, no. 1, pp. 5-22, 2011.

[12] A. Samuel, "Some studies in machine learning using the game of checkers", *IBM Journal. Recent Development,* vol. 3, pp. 210-229, 1959.

[13] A. Samuel, "Some studies in machine learning using the game of checkers: Recent progress", *IBM Journal. Recent Development,* vol. 11, pp. 601-617, 1967.

[14] C. Stachey, "Logical or non-mathematical programmes", *In: Proceedings of Association for Computing Machinery Meeting*, Toronto, 1952.

[15] J. Kister, P. Stein, S. Ulam, W. Walden and M. Wells, "Experiments in chess", *Journal ACM,* vol. 4, pp. 174-177, 1957.

[16] B. Jacobs, "Learning Othello using Cooperative and Competitive Neuroevolution", Netherlands, 2008.

[17] S. Lucas, "Computational Intelligence and AI in Games", *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 1, pp. 1-3, 2009.

[18] S. a. K. G. Lucas, "Evolutionary Computation and Games," *IEEE Computational Intelligence Magazine,* 2006.

[19] A. Hauptman and M. Sipper, "GP-EndChess: Using Genetic Programming to Evolve Chess Endgame Players", *In: Proceedings of the 8th European Conference on Genetic Programming (EuroGP '05)*, pp. 120-131, 2005.

[20] T. Cunningham, "Using the Genetic Algorithm to Evolve a Winning Strategy for Othello. Genetic Programming and Genetic Algorithms", Stanfort University, Department of Computer Science, California, 2003.

[21] R. E. Korf, "Multi-player alpha-beta pruning", *Artificial Intelligence,* vol. 48, pp. 99-111, 1991.

[22] D. E. Knuth and R.E. Moore, "An analysis of alpha-beta pruning", *Artificial Intelligence,* vol. 6, pp. 293-326, 1975.

[23] C.A. Luckhardt and K. B.Irani, "An algorithmic solution of N-person game", *In: Proceedings AAAI-86,* Philadelphia, 1986.

[24] A. Plaat, "A Minimax Algorithm faster than NegaScout," Rotterdam, 1997.

[25] M. Campbell, J. Hoane Jr and H. Feng-hsiung Hsu, "Deep Blue", *Artificial Intelligence,* vol. 134, pp. 57-83, 2002.

[26] M. Magnuson, "Monte Carlo Tree Search and Its Application", *In: UMM Computer Science Seminar Conference,* Minnesota, 2015.

[27] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI", *In: Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference,* Menlo Park, 2008.

[28] C. Brown, S. Lucus and P. I. Cowling, "A Survey of Monte-Carlo Tree Search Methods", *EEE Trancactions on Computational Intelligence and AI in Games,* vol. 4, no. 1, 2012.

[29] B. Brugmann, "Monte Carlo Go", 1993, [Online]:Available: http://www.ideanest.com/vegos/MonteCarloGo.pdf.

[30] B. Bruno, "Old-fashioned Computer GO vs Monte-Carlo GO", *In: IEEE Symposium on Computational Intelligence and Games*, Honolulu, Hawaii, 2007.

[31] D. Silver, A. Huang and C. J. Maddison, "Mastering the Game of Go with Deep Neural Networks and Tree Search", *Nature,* vol. 529, pp. 484-489, 2016.

[32] S. Nair, P. Kundzicz and V. Kumar, [Online]: Available: http://www.yisongyue.com/courses/cs159/lectures/MCTS.pdf

[33] G. D. Team, "DeepMind Mastering Go", 2016. [Online]. Available: http://airesearch.com/wp-content/uploads/2016/01/deepmind-mastering-go.pdf.

[34] A. Al Sallab and M. Rashwan, "USA Adaptation of a deep learning machine to real world data", *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 5, pp. 216-226, 2013.

[35] D. B. Fogel, "Evolutionary Computation: Toward a New Philosophy of Machine Intelligence", Piscataway: NJ: IEEE Press, 1995.

[36] K. Chellapilla and D. Fogel, "Evolving an expert checkers playing program without using human expertise", *In: IEEE Transactions on Evolutionary Computation*, 2001.

[37] D. Heger, "An Introduction to Artificial Neural Networks (ANN) - Methods, Abstraction, and Usage", 2001.

[38] M. Hajek, "Neural Networks", UKZN, Pietermaritzburg, South Africa, 2005. [Online]. Available: http://www.e-booksdirectory.com/details.php?ebook=10055

[39] C. Shannon, "A Chess-Playing Machine", *In: Computer Games*, New York, Springer New York, pp. 81-88, 1988.

[40] S. Y. Chong, M. K. Tan and J. D. White, "Evolved Neural Networks Learning Othello Strategies", *In: IEEE Transactions on Evolutionary Computation*, Canberra, Australia, 2003.

[41] D. Fogel, "Blondie 24: Playing at the edge of AI", Morgan Kaufmann, San Francisco, 2002.

[42] B. Pell, "Strategy Generation and Evaluation for Meta-Game Playing", Cambridge, 1993.

[43] M. Melanie, "An Introduction to Genetic Algorithms", London: MIT Press, 1998.

[44] J. H. Holland, "Adaptation in Natural and Artificial Systems", Michigan: MIT Press, 1992.

[45] J. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992.

[46] W. B. Langdon, R. I. McKay and L. Spector, "Genetic Programming: Handbook of Metaheuristics", *International Series in Operations Research & Management Science,* vol. 146, pp. 185-225, 2010.

[47] D. E. Moriarty and R. Miikkulainen, "Discovering Complex Othello Strategies Through Evolutionary Neural Networks", *Connection Science,* vol. 7, pp. 195-209, 1995.

[48] O. David, J. van den Herik, M. Koppel and N. Netanyahu, N, "Genetic Algorithms for Evolving Computer Chess Programs", *In: IEEE Transactions on Evolutionary Computation*, 2014.

[49] J. E. Clune, "Heuristic Evaluation Functions for General Game Playing", *In: Künstliche Intelligenz,* vol. 25, no. 73, pp. 73-74, 2011.

[50] D. Gleich, "Machine Learning in Computer Chess: Genetic Programming and KRK", Claremont, Calif, USA, 2003.

[51] D. E. Moriarty and R. Miikkulainen, "Efficient reinforcement learning through symbiotic evolution", *Machine Learning,* vol. 22, pp. 11-32, 1996.

[52] D. E. Moriarty and R. Miikkulainen, "Forming Neural Networks Through Efficient and Adaptive Coevolution", *Evolutionary Computation,* vol. 5, pp. 373-399, 1997.

[53] F. J. Gomez and R. MiiKKulainen, "Solving Non-Markovian Control Tasks with Neuroevolution", *In: Proceedings of the International Joint Conference on Artificial Intelligence,* Stockholm, 1999.

[54] P. Rosenbloom, "A World-Championship-Level Othello Program", *Artificial Intellegence,* vol. 19, no. 3, pp. 279-320, 1981.

[55] D, Billman and D. Shaman, "Strategy knowledge and strategy change in skilled performance: A study of the game Othello", *American Journal of Psychology,* vol. 103, pp. 145-166, 1990.

[56] U. S. O. A, "United States Othello Association", 2016. [Online]. Available: http://usothello.org/misc/USOA_Tourn_Rules.pdf.

[57] V. Oberg, "Diva-portal", 2015. [Online]. Available: http://www.diva-portal.org/smash/get/diva2:823737/FULLTEXT01.pdf.

[58] N. J. Nilsson, "Problem-Solving Methods in Artificial Intelligence", New York: McGraw-Hill, 1971.

[59] Y. Lai, S. Chang and T. Ko, "Applying Genetic Algorithm and Self-Learning on Computer Othello", Taiwan, 2006.

[60] K. Chen, "Computer Go: Knowledge, Search, and Move Decision", *ICCA*, vol. 24, no. 4, pp. 203-216, 2001.

[61] V. Sannidhanam and M. Annamalai, "An Analysis of Heuristics in Othello", 2015. [Online]. Available: http://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf., Russia.

[62] J. Cirasella and D. Kopec, "Academic Works", 2006. [Online]. Available: http://academicworks.cuny.edu/cgi/viewcontent.cgi?article=1181&context=gc_pubs.

[63] A. Benbassat, A. Elyasaf and M. Sipper, "More or Less? Two Approaches to Evolving Game-Playing Strategies", *Genetic Programming Theory and Practice X*, pp. 171-185, 2013.

[64] A. Leouski, "Learning of Position Evaluation in the Game of Othello", Massachusetts, 1995.

[65] F. Lee and S. Mahajan, "The Development of a World Class Othello Program", *Artificial Intelligence*, vol. 43, pp. 21-36, 1990.

[66] F. Lee and S. Mahajan, "A Pattern Classification Approach to Evaluation Function Learning", *Artificial Intelligence*, vol. 36, pp. 1-25, 1988.

[67] R. M. Neal, "Bayesian Learning for Neural Networks", Toronto, 1995.

[68] M. Buro, "The Evolution of Strong Othello Programs", *In: Entertainment Computing*, Springer, Springer US, 2003, pp. 81-88.

[69] G. Andersson, "WZebra Othello", 2006-2016. [Online]. Available: http://radagast.se/othello/download.html.

[70] K. Tournavitis, "Herakles Othello", 2004 - 2016. [Online]. Available: http://www.tournavitis.de/herakles/.

[71] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction", London England: MIT Press, 2012.

[72] J. Clouse, "Learning Application Coefficients with a Sigma-Pi Unit", Massachusetts, 1992.

[73] D. E. Moriarty and R. Miikkulainen, "Evolving Complex Othello Strategies Using Marker-Based Genetic Encoding of Neural Networks", Texas, 1993.

[74] J. Alliot and N. Durand, "A genetic algorithm to improve an Othello program", *Evolution Artificielle*, vol. 1063, pp. 307-319, 1995.

[75] LO4D, "Deep Green Reversi", 2008-2016. [Online]. Available: http://deep-green-reversi.en.lo4d.com/.

[76] FOG, "Online Games", 2008-2016. [Online]. Available: http://www.freeonlinegames.com/game/3d-reversi.

[77] A. C. F., "American Checker Federation", 2016. [Online]. Available: http://www.usacheckers.com/.

[78] I. D. F., "International Draughts Federation", 2016. [Online]. Available: https://idf64.org/a-history-of-draughts/.

[79] MTG, "Masters Traditional Game", 1999-2016. [Online]. Available: http://www.mastersgames.com/rules/draughts-rules.htm.

[80] R. E. Mayer, "Rote Versus Meaningful Learning", *Theory into Practice,* vol. 41, no. 4, 2002.

[81] A. Singh and K. Deep, "Use of Evolutionary Algorithms to Play the Game of Checkers: Historical Developments, Challenges and Future Prospects", *In: Proceedings of the Third International Conference on Soft Computing for Problem Solving, Advances in Intelligent Systems and Computing 259*, Springer India, 2014.

[82] M. Fiertz, "A brief history of computer checkers", 2009. [Online]. Available: http://www.fierz.ch/history.htm.

[83] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu and D. Szafron, "A World Championship Caliber Checkers Program", *Artificial Intelligence,* vol. 53, no. 2-3, pp. 273-289, 1992.

[84] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu and S. Sutphen, "Solving Checkers", *Science,* vol. 317, no. 5844, pp. 1518-1522, 2007.

[85] J. Schaeffer and R. Lake, "Solving the Game of Checkers", *Games of No Chance,* vol. 29, 1996.

[86] Microsoft, "MSN Games", 2001-2016. [Online]. Available: http://zone.msn.com/en-us/home.

[87] K. Waledzik and J. Mandziuk, "The Layered Learning method and its application to generation of evaluation functions for the game of checkers", *In: 11th International Conference*, Kraków, Poland, 2010.

[88] M. Kusiak, K. Waledzik and J. Mandziuk, "Evolutionary approach to the game of checkers", *In: Adaptive and Natural Computing Algorithms, 8th International Conference, ICANNGA 2007*, Warsaw, Poland, 2007.

[89] H. Bird, "Chess History and Reminiscences", 2002. [Online]. Available: http://www.cs.unibo.it/~cianca/wwwpages/chesssite/bird.pdf.

[90] H. Nasreddine, H. Poh and G. Kendall, "Using an Evolutionary Algorithm for the Tuning of a Chess Evaluation Function Based on a Dynamic Boundary Strategy", *In: IEEE Conference on Cybernetics and Intelligent Systems*, Bangkok, 2006.

[91] N. Lassabe, S. Sanchez, H. Luga, and Y. Duthen, "Genetically programmed strategies for chess endgame", *In: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006.

[92] FIDE, "World Champion Match Rules and Regulations for Tournament Chess", [Online]. Available: https://www.fide.com/component/handbook.

[93] D. Rasskin-Gutman, "Chess Metaphors - Artificial Intelligence and the Human Mind", MIT Press, 2009.

[94] W. G. Chase and H. A. Simon, "Perception in Chess", *Cognitive Psychology*, vol. 4, pp. 55-81, 1973.

[95] A. Newell, J. C. Shaw and N. Simon, "Elements of a theory of human problem solving", *Psychological Review*, vol. 65, pp. 151-166, 1958.

[96] R. Korf, "Does deep-blue use AI?", *In: Proceedings of the 4th AAAI Conference on Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, 1997.

[97] S. Thrun, "Learning to Play the Game of Chess", *In: Advances in Neural Information Processing Systems 7*, MIT Press, 1995, pp. 1069-1076.

[98] B. Oshri and N. Khandwala, "Predicting Moves in Chess using Convolutional Neural Networks", 2005. [Online]. Available: http://cs231n.stanford.edu/reports/ConvChess.pdf.

[99] Free Software Foundation, "GNU Chess", [Online]. Available: https://www.gnu.org/software/chess.

[100] D. Sharma and U. Chakraborty, "An Improved Chess Machine based on Artificial Neural Networks", *International Journal of Computer Applications® (IJCA)*, 2014.

[101] O.E. David, N.S. Netanyahu and L. Wolf, "DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess", *In: Artificial Neural Networks and Machine Learning*, Barcelona, Spain, Springer International Publishing, 2016, pp. 88-96.

[102] M. Webster, "Merriam Webster Dictionary", 2016. [Online]. Available: http://www.merriam-webster.com/dictionary/a%20priori.

[103] CPW, "Chess Programming WIKI", 2016. [Online]. Available: https://chessprogramming.wikispaces.com/Falcon.

[104] G. Kendall and G. Whitwell, "An evolutionary approach for the tuning of a chess evaluation function using population dynamics", *In: Proceedings of the 2001 Congress on Evolutionary Computation*, Seoul, Korea, 2001.

[105] M. Block, M. Bader, E. Tapia, M. Ramirez, K. Gunnarsson, E. Cuevas, D. Zaldivar and R. Rojas, " Using reinforcement learning in chess engines", *Research in Computing Science,* vol. 35, pp. 31-40, 2008.

[106] J. Baxter, A. Tridgell and L. Weaver, "Learning to Play Chess Using Temporal Differences", *Machine Learning* , vol. 40, no. 3, pp. 243-263, 2000.

[107] Software Toolworks, "The Fidelity Chessmaster 2100", 1989. [Online]. Available: https://archive.org/details/msdos_The_Fidelity_Chessmaster_2100_1989.

[108] A. M. Turing, "Intelligent machinery: Collected Works of A. M. Turing", 1992.

[109] C. Darwin, "The Origin of Species", John Murray, penguin classics, 1985 edition, 191859, 1985.

[110] R. Poli, W. B. Langdon and N. F. McPhee, "A Field Guide to Genetic Programming", Essex, 2008.

[111] M. Walker, "Introduction to Genetic Programming", Québec City, 2001.

[112] W. Banzhaf, P. Nordin,R. E. Keller and F. D.Francone, "Genetic Programming: An Introduction", The Morgan Kaufmann Series in Artificial Intelligence, 1997.

[113] R. Poli, W.B. Langdon and N. F. McPhee, "On the search properties of different crossover operators in genetic programming", *Genetic Programming,* pp. 293-301, 1998.

[114] J. H. Holland, "Adaptation in Natural and Artificial Systems", Michigan: University of Michigan Press, 1975.

[115] L. Beadle and C. G. Johnson, "Semantically driven crossover in genetic programming", *In: Proceedings of the IEEE World Congress on Computational Intelligence*, 2008.

[116] E. F. Crane and N. F. McPhee, "Genetic Programming Theory and Practice III", *In: The Effects of Size and Depth Limits on Tree Based Genetic Programming*, Springer US, Springer, 2006, pp. 223-240.

[117] G. Williams, "Adaptation and Natural Selection", Princeton, USA: Princeton University Press, 1966.

[118] D. Montana, "Strongly typed genetic programming", *Evolutionary Computation,* vol. 3, no. 2, pp. 199-230, 1995.

[119] E. Eskin and E. Siegel, "Genetic Programming Applied to Othello: Introducing Students to Machine Learning Researc", *In: 30th Technical Symposium of the ACM Special Interest Group in Computer Science Education*, New York, 1999.

[120] A. Benbassat, "Finding Methods for Evolving Competent Agents in Multiple Domains", Negev, 2014.

[121] A. Benbassat and M. Sipper, "Evolving Both Search and Strategy for Reversi Players Using Genetic Programming" *In: IEEE Conference on Computational Intelligence and GAmes(CIG)*, Granada, 2012.

[122] A. Benbassat and M. Sipper, "EvoMCTS:Enhancing MCTS-Based Players Through Genetic Programming", *In: Proceedings of the 2013 IEEE Symposium on Computational Intelligence and Games (CIG)*, Niagra Falls.

[123] A. Benbassat and M. Sipper, "EvoMCTS: A Scalable Approach for General Game Learning", *IEEE Transactions in Compututer. Intellegence AI Games*, vol. 6, no. 4, 2014.

[124] A. Benbassat and M. Sipper, "Evolving Lose-Checkers Players Using Genetic Programming", *In: Proceedings of the 2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, Dublin, 2010.

[125] W.C.D.F., "World Checkers and Draughts Federation", [Online]. Available: http://www.wcdf.net/rules/rules_of_checkers_english.pdf.

[126] G. M. Khan, J. F. Miller and D. M. Halliday, "Developing Neural Structure of Two Agents that Play Checkers Using Cartesian Genetic Programming". *In: GECCO '08 Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, New York, USA, 2008.

[127] J. F. Miller and P. Thomson, "Cartesian Genetic Programming", *EuroGP*, vol. 1802, pp. 121-1321, 2000.

[128] B. M. McCarver, "Structure and Function of Neurons", 2000. [Online]. Available: http://www.universitypsychiatry.com/clientuploads/stahl/Stahl_3rd_ch01.pdf.

[129] Chess.com, "Endgame Practice", 2017. Online]. Available: https://www.chess.com/drills/endgame-practice.

[130] R. Gross, K. Albrecht, W. Kantschik and W. Banzhaf, "Evolving Chess Playing Programs", *In: GECCO*, San Francisco, 2002.

[131] Chess Programming WIKI, "Nalimov Tablebases", 2005 - 215. [Online]. Available: https://chessprogramming.wikispaces.com/Nalimov+Tablebases.

[132] C. Johnson, "Basic Research Skills in Computing Science", United Kingdom, 2001.

[133] U. O. F., "United States Othello Association", 2016. [Online]. Available: http://usothello.org/.

[134] O. H. M., "Othello History Musium", [Online]. Available: http://www.beppi.it/ public/ OthelloMuseum/ pages/ history.php/.

[135] Darkfish Games, "Checkers Rules", [Online]. Available: http://www.darkfish.com/checkers/rules.html.

[136] World Chess Federation, "Word Champion Match Rules and Regulation for Chess Tournaments", [Online]. Available: https://www.fide.com/component/handbook/?id=3&view=section.

[137] M. Onkar, P. Vishal and S. Satyendra, "Evaluation Function to predict move in Othello", *ISSN International Journal of Emerging Trend in Engineering and Basic Sciences*, vol. 2, no. 1, pp. 481-487, 2015.

[138] D. Farren, D. Templeton and M. Wang, "Analysis of networks in chess", Stanford University, Stanford, U.S.A., 2013.

[139] W. Iba, "Searching for Better Performance on the King-Rook-King Chess Endgame Problem", *In: Proceedings of the Twenty-fifth International FLAIRS Conference*, Marco Island, Florida, USA, 2012.

[140] A. Hauptman and M. Sipper, "Emergence Of Complex Strategies in the Evolution of Chess Endgame Players", *Advances in Complex Systems*, vol. 10, pp. 35-59, 2007.

Appendix A - User Manual

This appendix describes the integrated testing environments for the board games Othello, checkers and chess.

A.1 Program Requirements

Java must be installed on your computer in order to run the three Java executable files. The latest version of Java can be down loaded from the following website, http://java.com/en/download. In the GP Games folder, there are three folders, each containing the integrated test environments for one of the three games. These folders are GP Othello, GP Checkers and GP Chess. Copy the GP Game folder onto your computer hard drive. The GP control parameters are stored in the Config.txt file. If required, the random number seed value can stored in the Seed.txt file. If the value is set to 0 the system will generate a seed value.

A.2 Othello

The GP_Othello.jar is the Java executable file for the Othello program. Run the Othello.bat file to invoke the GP_Othello.jar executable. The Othello integrated testing environment user interface is shown in Figure A1.

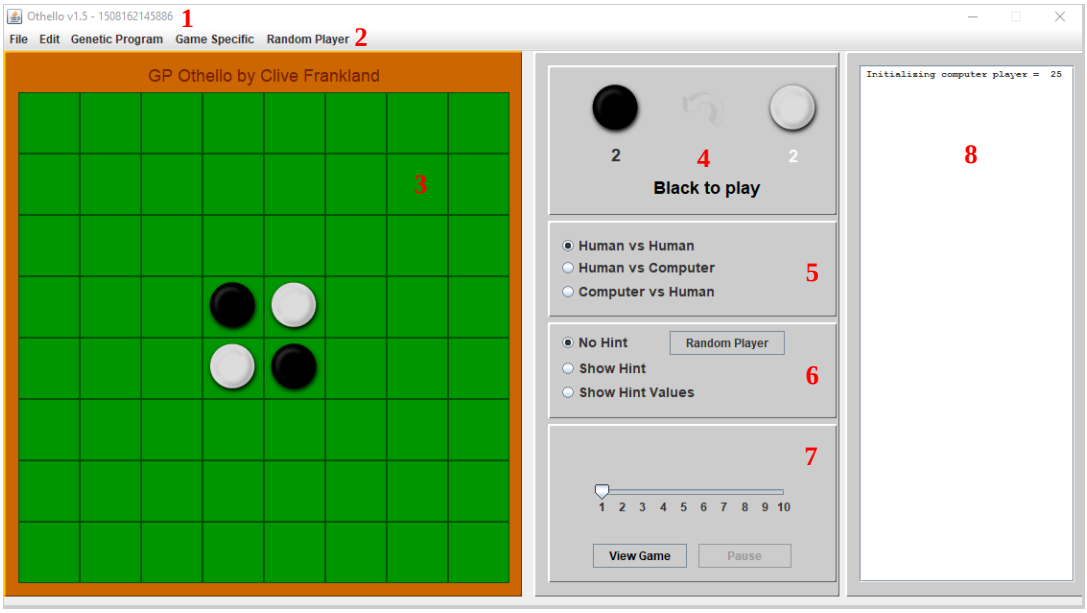


Figure A1. Othello integrated testing environment user interface

1. Current seed value
2. Menu
    a. File
        i. New – Start a new game
        ii. Open – Load a saved board configuration
        iii. Save – Current board configuration
        iv. Load Alpha
        v. Save Alpha
        vi. Exit
    b. Edit
        i. Flip Board – Alternate start positions
        ii. Setup Board
        iii. Show Board Heuristics
        iv. Run Code Snippet – Test code snippets
    c. Genetic Program
        i. Preserve Alpha
        ii. Preserve Population
        iii. Mobility – Evaluate with mobility strategies
    d. Game Specific
        i. Evaluate Corners – Force corner strategy evaluation
        ii. Heuristic Filter – Used for edge strategy evaluation
    e. Random player
        i. Set computer player to pseudo-random move player
        ii. Let pseudo-random move player evaluate corner and edge strategies
3. Othello game board
4. Score and board setup dashboard
5. Select opponents
6. Display dashboard
7. Game controller – Game speed, view and pause game play
8. Debug information panel

A.3 Checkers

The GP_Checkers.jar is the Java executable file for the checkers program. Run the Checkers.bat file to invoke the GP_Checkers.jar executable. The checkers integrated testing environment user interface is shown in Figure A2.

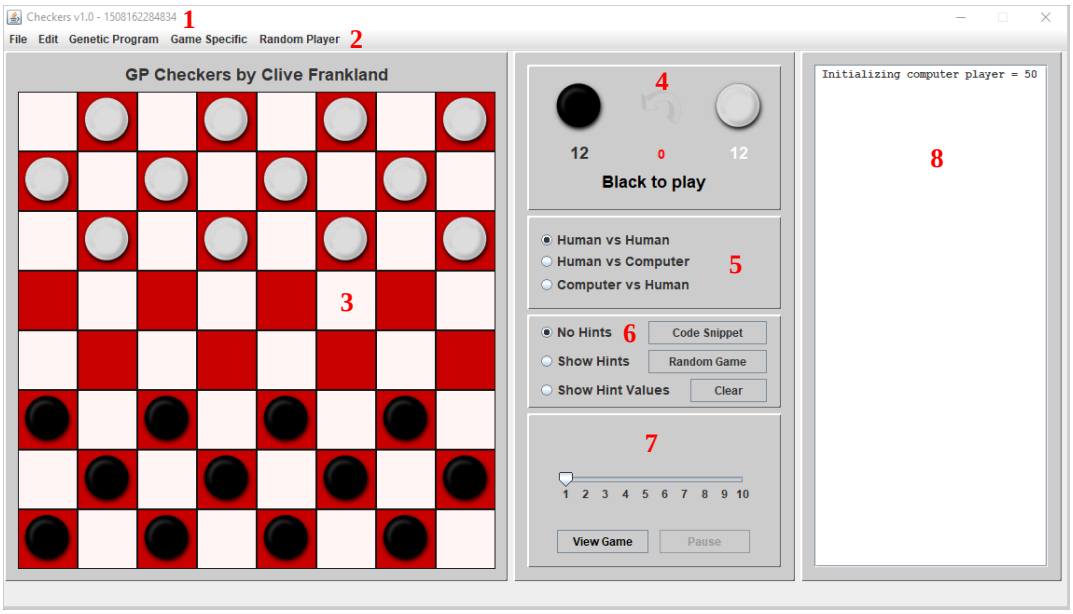


Figure A2. Checkers integrated testing environment user interface

1. Current seed value
2. Menu
    a. File
        i. New – Start a new game
        ii. Open – Load a saved board configuration
        iii. Save – Current board configuration
        iv. Load Alpha
        v. Save Alpha
        vi. Exit
    b. Edit
        i. Flip Board – Rotate board 180°
        ii. Setup Board
        iii. Show Board Heuristics

iv. Show Hints
v. Print Hint Matrix
vi. Run code snippet – Test code snippets
vii. Refresh GUI

c. Genetic Program
i. Preserve Alpha
ii. Preserve Population
iii. Mobility – Evaluate with mobility strategies

d. Game Specific
i. Heuristic Filter – Force game play strategies

e. Random player
i. Set computer player to pseudo-random move player

3. Checkers game board
4. Score and board setup dashboard
5. Select opponents
6. Display and Control dashboard
7. Game controller – Game speed, view and pause game play
8. Debug information panel

A.4 Chess

The GP_Chess.jar is the Java executable file for the chess program. Run the Chess.bat file to invoke the GP_Chess.jar executable. The chess integrated testing environment user interface is shown in Figure A3.

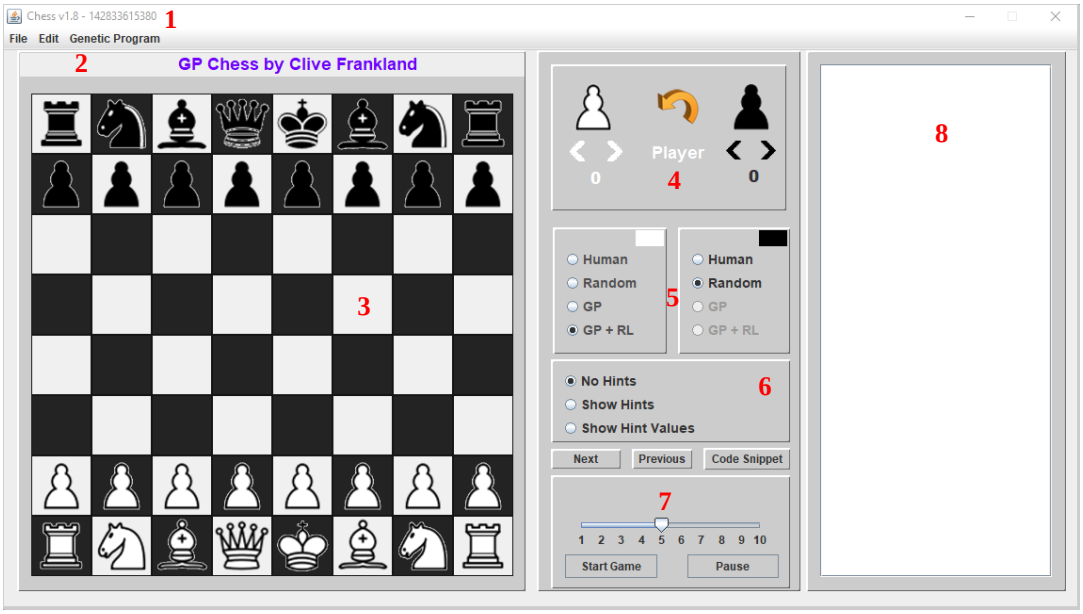


Figure A3. Chess integrated testing environment user interface

1. Current seed value
2. Menu
    a. File
        i. Initialize Board
        ii. New Game
        iii. Open Board – Load saved board configuration
        iv. Save Board – Save current board configuration
        v. Open Game – Play a recorded game
        vi. Record Game
        vii. Exit
    b. Edit
        i. Setup Board
        ii. Clear Board

iii. Show Board Heuristics
iv. Show hints
v. Print Hint matrix
vi. Run Code Snippet – Test code snippets
vii. Refresh GUI

c. Genetic Program
i. Preserve alpha
ii. Preserve population

3. Chess game board
4. Score and board setup dashboard
5. Select opponents
6. Display and Control dashboard
7. Game controller – Game speed, view and pause game play
8. Debug information panel

- End -