A learning framework for zero-knowledge game playing agents

by

Willem H. Duminy

Submitted in partial fulfilment of the requirements for the degree

Master of Science (Computer Science)

in the Faculty of Engineering, Built-Environment

and Information Technology

University of Pretoria

South Africa

May 2006

To my wife Karen,

for her patience, and her loving support.

Abstract

The subjects of perfect information games, machine learning and computational intelligence combine in an experiment that investigates a method to build the skill of a game-playing agent from zero game knowledge. The skill of a playing agent is determined by two aspects, the first is the quantity and quality of the knowledge it uses and the second aspect is its search capacity. This thesis introduces a novel representation language that combines symbols and numeric elements to capture game knowledge. Insofar search is concerned, an extension to an existing knowledge-based search method is developed. Empirical tests show an improvement over alpha-beta, especially in learning conditions where the knowledge may be weak. Current machine learning techniques as applied to game agents is reviewed. From these techniques a learning framework is established. The data-mining algorithm, ID3, and the computational intelligence technique, Particle Swarm Optimisation (PSO), form the key learning components of this framework. The classification trees produced by ID3 is subjected to new post-pruning processes specifically defined for the mentioned representation language. Different combinations of these pruning processes are tested and a dominant combination is chosen for use in the learning framework. As an extension to PSO, tournaments are introduced as a relative fitness function. A variety of alternative tournament methods are described and some experiments are conducted to evaluate these. The final design decisions are incorporated into the learning framework configuration, and learning experiments are conducted on Checkers and some variations of Checkers. These experiments show that learning has occurred, but also highlights the need for further development and experimentation. Some ideas in this regard concludes the thesis.

Keywords: Machine Learning, Games, Knowledge Representation, Knowledge Discovery, Particle Swarm Optimisation, Checkers, Coevolution, Computational Intelligence, Classification, Game Tree Searching.

Supervisor: Prof. A. P. Engelbrecht (Department of Computer Science)

Degree: Magister Scientiae

Acknowledgements

The task of writing a thesis is a solitary activity. However, it is the attentive nods from friends and the unfailing support from family that kept me returning to my desk every day. For this, I am eternally grateful.

I would like to highlight the contribution of three people in particular:

- **Professor Engelbrecht** for his guidance as my study supervisor. It has been a privilege to have him as mentor.
- **Alice Meyer** for reading through draft chapters. With her help, numerous grammar and spelling bloopers were eliminated from the start.
- **Ferdie van Deventer** for lending me a high-end, multi-processor computing machine. This machine was used for most of the experiments described in this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

One of the earliest game learning research endeavours is the work started by Arthur Samuel in 1953. His first paper [53], published in 1959, introduces a program that learns the weights of a linear evaluation function. Samuel's research objective was to demonstrate to his contemporaries that it is possible to write a program that has the ability to learn. Minsky's paper of 1963 [43] shows that Samuel did achieve his objective. In the decades that followed, this work of Samuel formed the foundation of many machine learning efforts. This thesis is no exception.

The key influence of Samuel's paper on the current work is a challenge it submits. Samuel insisted that more research effort should be expended on the problem of constructing the terms of the evaluation function. The function terms used by Samuel's learner were based on Checkers literature and hand-coded using the method of trial-and-error.

The automatic construction of the terms of a function like Samuel's, has not been an objective shared by many researchers in the game learning arena. Here, the focus turned towards the construction of a world-class player, and search techniques proved to be an effective attack on that problem. Neural networks has also become a popular structure for evaluation functions, eliminating the need for linear terms. An exception is the work conducted by Michael Buro [10] that examines the construction of complex terms from simpler ones.

This thesis describes the research conducted that considers Samuel's challenge as a research objective. The fundamental idea was to identify and work with elements that are more elementary than those developed by Buro. This approach necessitates the definition of a representation that facilitates the construction of complex terms. This representation is the backbone on which

a learning framework is developed. This framework embodies an approach that discovers the terms of the evaluation function using as input a program that implements little more than the rules of the game. In addition, this framework also takes on the task of optimising the weights associated to the newly discovered terms. This task is not new, but the current research explores the use of Particle Swarm Optimisation and tournaments as an alternative approach to find optimal values for those weights.

1.2 Objectives

The research approach followed to address the chosen problem is one that ensures an adequate review of the game playing and game learning concepts, as well as the specialised learning techniques chosen for the current study. The problem was divided into smaller objectives, and each objective became a task in the research plan. These objectives are:

1. To provide an overview of game concepts and identify a particular class of games for which the game framework needs to be developed. This class must be general enough to ensure that the framework is widely applicable and complex enough to demonstrate the effectiveness of the learning framework.

2. To provide an overview of the techniques employed by automated game players as well as the information these players use. These insights describe the substance that must be learned as well as the environment in which the framework would operate.

3. To introduce a knowledge representation mechanism in which game knowledge can be captured. Clearly, this method must be able to capture the mentioned substance and be conducive to automated learning.

4. To review the methods employed previously to discover knowledge and optimise weights for game playing agents. The scope of this review should generally be restricted to methods related to the identified game class.

5. To define a method that induces the terms of an evaluation function. This method should produce a structure that corresponds with the previously defined representation.

6. To apply PSO to the weight optimisation problem, giving specific consideration to the use of tournament methods for this purpose.

7. To define a learning framework that combines the developed methods into a coherent learning process. This framework must be tested on game types that are included in the class previously identified.

1.3 Contributions

In addition to describing a learning framework as it set out to do, this research has contributed to each of the subjects it encountered.

The key contributions to the subject area of perfect information game playing agents include:

- The definition of a knowledge representation language that combines symbolic and numeric elements. This language can be used to describe the knowledge of a large subset of perfect information games.
- A new game tree search algorithm, called best-first shallow search that can search game trees selectively. It produces better results than its forerunners, especially when the available knowledge is weak.

Contributions to machine learning:

- An set of algorithms that use ID3 to induce game knowledge from a large set of example positions.
- Two post-pruning techniques that simplify the decision trees obtained from ID3. Experiments are used to compare alternative combinations of these two techniques.

Contributions to computational intelligence:

- A new PSO algorithm, called tournament PSO, which uses tournaments to determine the best particle in the neighbourhood and in the swarm.
- A review of the most popular tournament methods, and an experimental comparison of the performance of these methods.

At the time of this writing, the contribution mentioned first in this list is the only contribution published as a peer reviewed paper in an accredited journal. See [16] for the reference details on this article. It describes the knowledge representation language.

1.4 Thesis outline

The thesis consists of three distinct parts. The first part consists of chapters 2, 3 and 4 which predominantly focus on the automation of a game-player. In these chapters the problem of game playing is described and techniques that affect the playing of games are considered. In the second part, chapter 5, 6 and 7 describe the learning problem and introduce techniques that pertain to the learning process. In the final section, chapter 8 and 9, some experiments are conducted with the learning framework as a whole and the conclusions derived from this study are summarised. The subsequent paragraphs provide a brief summary the rest of the chapters.

Chapter 2 reviews fundamental two-player definitions and concepts used in chapters that follow. Furthermore, the reasons for using games in research and the properties that are typically used to classify two-player games are explored.

Chapter 3 provides a review of the knowledge types and the knowledge representation schemes used by successful learning- and playing agents. It also proposes a new representation language for game knowledge that includes symbols and numbers. A precise definition of this language is provided.

Chapter 4 considers the manner in which the agent applies knowledge during the search of the game tree. A new search method called the best-first shallow search is presented and evaluated.

Chapter 5 reviews machine learning methods applicable to the game learning domain. It also introduces the learning framework. This framework has a cycle (called the macro learning cycle) that consists of four stages.

Chapter 6 covers the discovery stage of the macro learning cycle. It provides an overview ID3 and describes how this data-mining method is employed to discover game knowledge. Alternative pruning methods are introduced and evaluated to identify the discovery strategy that is most suitable for the learning framework.

Chapter 7 considers the optimising stage of the macro learning cycle. The particle swarm optimisation (PSO) algorithm is described. A new form of PSO that uses tournaments is introduced. Experiments in this chapter select the most viable option amongst alternative configurations of the new PSO.

Chapter 8 is an empirical analysis of the performance of the learning framework. The game of C is used as the subject for this investigation. The effect of some framework parameters on the learning performance is measured.

Chapter 9 summarises the most important conclusions and contemplates opportunities for research that would extend the work originated from this study.

Chapter 2

Two-player games

This chapter reviews fundamental two-player definitions and concepts used in chapters that follow. Furthermore, the reasons for using games in research and the properties that are typically used to classify two-player games are explored. Game convergence is such a property, and a new, more general definition of this property extends work done previously in this regard. Throughout the chapter, the perspective remains on the influence the presented material might have on the definition of a learning framework.

2.1 Introduction

For centuries, two-player games continued to be a source of amusement to the human race. This lasting interest can surely by attributed, at least in part, to the variety of games available. From this variety many diverse strategies arise, such that a strategy that often wins any particular game is seldom transferable to another game. Because every game is a different kind of puzzle, many researchers found it best to choose a specific game as subject. Others, like David Fogel [21, 22, 23] started with simpler games and turned their attention to more complex games as the research progressed.

An automated learning process that can be applied to all two-player games has not yet been discovered, and it is likely to remain unattainable for some time. This general problem can be divided into smaller steps by considering layers of knowledge. Each layer builds on the knowledge obtained by the previous layer. At the bottom layer, there is the *playing agent* that has the knowledge of how to play the game. The next layer is the *learning agent*; it needs to know how to play, but also has knowledge on how to learn to play better. The third layer is the *learning framework*; it extends the knowledge of how to learn a particular game to define

a process by which other games of the same class can be learned. Finally, at the top of the stack, the *two-player game agent* automates the improvement of learning frameworks and the development of new learning frameworks for other classes of two-player games. The current work contributes to the research conducted at the second and third layers.

At the start of this chapter, Section 2.2 provides background on the terminology and the concepts pertaining to automated game-playing. Then, Section 2.3 reviews the most important drivers that motivate researchers to spend their time on the subject of games. Game playing cannot be automated without search, and Section 2.4 highlights this important facet and considers the influence of search on the learning framework. Section 2.5 addresses the classification problem by considering a few critical properties of two-player games. The final section of this chapter provides a summary of the notable conclusions.

2.2 Game playing terminology and concepts

Fundamentally, a game is a set of rules. In a *two-player game*, two participants compete in a contest determined by the game rules. These rules define how the game starts, what the legal moves are, when the game ends and who the winner is. In an abstract sense, game rules describe allowable changes to the *game state*. For board games, the game state is usually referred to as a *position*. A *legal position* is one that can be constructed by playing the game according to the rules. The *state-space* is the set of all legal positions.

Three mutually exclusive sets cover the set of legal positions: the set of *initial positions*; the set of *middle positions* and the set of *end positions*. A game starts with an initial position and it ends with an end position. Any legal position not in the initial- or end position set belongs to the set of middle positions. Many popular games have only one initial position, but there are exceptions. For instance, in T C the initial position is determined from a balloted sequence of three moves [57] – and the set of initial positions has 144 elements. A starts with each player in turn arranging 16 game pieces on the nearest two ranks on a C board. In this game there are thousands of initial positions*.

The outcome of a match is determined from the end position. A game in which the outcome is either a win, a lose or a draw for each player is said to have a *restricted outcome* [14]. The player's goal is to make moves that force the outcome towards an end-game position in which he wins. With respect to a player, the symbol $\mathcal{L}$ signifies the lose outcome, $\mathcal{D}$ signifies the draw,

*See http://www.arimaa.com for the A game rules

and $\mathcal{W}$ signifies the win of a match.

Using a position centric view, the rules of the game can be regarded as the input to a production system that starts from an initial position and produces all the legal positions that follow. Repeating the production, level by level, results in a tree structure with the initial position as the root and end positions as leaf nodes. This tree structure is called a *game tree* and each level in the tree is called a *ply* [53]. The *branching factor* is a measure of the average number of child nodes. The set of all game trees with an initial position as root is called the *game forest.*

During the game, the nodes of the game tree is traversed and a path is constructed. This path is called a *play-line* and it starts at the root of a game tree and extends towards the leaves of the tree as the game progresses. The *active game tree* is the tree in which the play-line is found. The *active position* is the last node in the play line. The *past positions* are the nodes that precede the active position, and the *future lines* are all the paths in the game tree with the active position as root.

For any given position, the *active player* is the player that must choose the next node in the play line. The other player is called the *passive player.* The player that is active at the first node in the play line is called the *first player* and the passive player for that node is referred to as the *second player.*

2.3 Motivations for using games in research

The domain of two-player games presents a scientist in the field of artificial intelligence (A.I.) with an artificial problem that has well defined rules and a definite goal. Even though the goal and the rules are elementary, devising a strategy to win a non-trivial game is a complex task that demands the use of intelligence. These qualities of game domains has been recognised as beneficial to A.I. research since the late 1950's [53].

The primary motivations for using games as a research tool support three different research objectives. One objective aims to win tournaments and focus on creating the best player for a particular game. Another objective aims to solve the game by producing an infallible strategy. The third objective regards games as ideal learning domains, and aims to investigate machine learning methods.

When compared to research projects conducted to create winning computer players, projects that apply games to the problem of investigating learning methods are less prevalent. There is reason to expect that a solution to the latter coincides with progress made to the former; and

notable contributions have been made in this regard. However, it is also possible for competition matches, especially public exhibitions, to stifle research progress. In Section 2.3.1 the contributive as well as the prohibitive aspects of these matches are explored. The objective to solve a game is discussed in Section 2.3.2. Section 2.3.3 considers the factors that make game domains ideal for automated learning.

2.3.1 Winning public tournaments

Artificial intelligence is convincingly demonstrated when the behaviour of the artificial agent is understood by the audience. Therefore a game playing program that beats a human opponent is ideal for such displays. Annual tournament matches against C playing agents have been held since 1970 [15]. In 1992, the American Checkers Foundation and the English Draughts Association even established a Man-Machine World Championship title for C [57]. The computer C match of 1997 that played off Deep Blue against Gary Kasparov (the world C champion since 1985) is arguably the most celebrated achievement in A.I. history.

The desire to perform well in the spotlight is a strong motivation: potential returns include monetary gains, publicity and prestige. However, the famous Deep Blue vs. Kasparov match and other like-minded efforts had a less than expected impact on the science of A.I. This is partly because exhibition programs implement techniques that many A.I. researchers regard as uninteresting. Public exhibitions encourage small, short-term projects that focus on the player's performance, while A.I. research attempts to find solutions to problems that require a longer term commitment. Some specific issues have been highlighted with regard to C programming endeavours [15]:

* While they were chasing deadlines, the neglect of C programmers to sell their ideas contributed to the diminution of the esteem once enjoyed by the game of C in A.I. research.
* Experiments conducted solely to test the strength of the player lead to a reliance on results of which the underlying theory was not always understood. Even though a lot of research was conducted, there is little theory of C knowledge and its interactions with search.
* High pay-off to the winner eradicates incentives for the dissemination of research results. The tendency is to hold back on ideas to maintain a competitive edge, leaving each team to discover principles that are already known to other teams.

Nonetheless, the exploration and definition of methods to create champion game software has become an established field with many research challenges. Even in C , human master players currently beat the best C programs more often than not [62]; and the next frontier is the game G . Research endeavours typically focus on high performance implementations that take advantage of specific hardware architectures and multiprocessor platforms. This commendable approach is likely to reassert itself as the most important factor in the next generation of champion game playing agents.

2.3.2 Solving the game

A game is solved if a playing agent is available that is guaranteed to win from a position where a win can be forced. In other words, the puzzle of winning the game against any opponent is solved. After a game is solved, it is possible to determine whether the game is fair. For example, T - - is trivial to solve, and it is not fair. Except for the uninitiated, everybody knows the strategy that makes it impossible for the second player to win a game of T - - .

A precise definition of what it means to solve a game is based on an irrefutable property of a game position, called the *game-theoretic value*. This value is computed in a retrograde search that starts with the end positions in the game tree. A precise definition of the game-theoretic value follows.

In restricted outcome games, every end position determines the outcome of the game. For end positions this outcome is the game theoretic value. The value is an element from the *outcome set* $\mathbb{O} \equiv \{\mathcal{W}, \mathcal{L}, \mathcal{D}\}$ that designates a win ($\mathcal{W}$), lose ($\mathcal{L}$) or a draw ($\mathcal{D}$) for the active player. For an initial and middle position the game theoretic value is $\mathcal{W}$ if it is in a play line in which the active player can force a win. Conversely, if the passive player can force a win from a position, the game-theoretic value of that position is $\mathcal{L}$. If neither player is able to secure a winning outcome, the position leads to a draw, and has the game-theoretic value of $\mathcal{D}$.

Formally, let the domain $\mathbb{S}_g$ be the state-space of game g and $\mathbf{C}_g : \mathbb{S}_g \rightarrow \{e \mid e \in \mathbb{S}_g\}$ be a set function that determines the positions that may legally follow from another. Then, the *game-theoretic value* for the active player can be defined as the recursive function $\theta_g : \mathbb{S}_g \rightarrow \mathbb{O}$:

$$\theta_g(s) = \begin{cases} \text{For the base step } (s \text{ is an end position}): \\ \quad \begin{cases} \mathcal{L} & \text{if the active player of } s \text{ loses} \\ \mathcal{W} & \text{if the active player of } s \text{ wins} \\ \mathcal{D} & \text{if } s \text{ is a draw position} \end{cases} \\ \\ \text{For the recursive step } (s \text{ is not an end position}): \\ \quad \begin{cases} \mathcal{L} & \text{if} \quad \exists e \in \mathbf{C}_g(s) \mid \theta_g(e) = \mathcal{W} \\ \mathcal{W} & \text{if} \quad \forall e \in \mathbf{C}_g(s)\ ,\ \theta_g(e) = \mathcal{L} \\ \mathcal{D} & \text{if} \quad (\nexists e \in \mathbf{C}_g(s) \mid \theta_g(e) = \mathcal{W}) \ \wedge \\ & \qquad (\exists e \in \mathbf{C}_g(s) \mid \theta_g(e) = \mathcal{D}) \end{cases} \end{cases} \tag{2.1}$$

Although the computation is simple, the size of the state space for non-trivial games makes it impossible to compute the game-theoretic value for all positions. However, as the availability of faster hardware and larger memory sizes increases it becomes slightly easier to compute these values.

From the game-theoretic value follows a definition in Allis [1]: A game is *strongly solved* when the game-theoretic value for both players can be determined for any legal position using reasonable computing resources. Without the reference to "reasonable resources", any two-player game is strongly solved, because the game-theoretic value can simply be determined using Equation 2.1. A number of games have been strongly solved: these include C -, Q , N M ' M , G -M and A [72]. Because of the end-game database, Chinook strongly solved the end-game of C [55].

The term *perfect play* is used to describe a move decision that ensures an outcome that is no less than the game theoretic value. The weakness of perfect play is that once it has been determined that a position has been reached with an outcome of a draw, there is no reason to continue playing. But, humans are not perfect players, and under draw conditions, the playing agent would do well to select a move that maximises the likelihood that the opponent makes an error. Schaeffer [57] highlights this situation by introducing a stronger definition: A game is *ultra strongly solved* when it is strongly solved and a strategy is known that improves the chance of achieving more than the game-theoretic value against a fallible opponent.

This aim to solve a game ultra strongly is interesting, but in practice the playing agent needs

a fallible opponent that plays very close to perfection before this ability can be tested. Even amongst master players, these individuals are extremely rare.

2.3.3 Investigating methods in machine learning

Arthur Samuel was the first to use game playing to investigate machine learning methods. He felt compelled to win over his contemporaries to the notion that a computer program can learn [53]. With this end in mind, he set out to develop a C program that uses previous experience to make a move decision. As attested by the praise of another prominent A.I. researcher of the time, Marvin Minsky, Samuel's program proved to be a convincing demonstration of machine learning [43].

There are four aspects of game playing that make them suitable for game learning: 1) game playing is non-deterministic in the sense that a practical algorithm that always wins is not known; 2) there is a definite goal - that is to win the game, and 3) the learning process is confined to a countable set of known rules; 4) it is easy to measure the performance of the learner.

Many real-world activities can be related to game playing; such as making business decisions in a competitive environment, deciding which shares to buy at the stock exchange, and so on. It is then reasonable to suggest that learning methods obtained from game based research could be applied to many real-world problems. However, there is another perspective: automating the learning process in the simplified game-world is a prerequisite to the automation of learning processes to real-world problems.

Even the strongest playing agents can benefit from, and contribute to machine learning research. In lieu of the fact that game knowledge is difficult to obtain, Jonathan Schaeffer [55] focussed on the development of a large C position database. This database stored an extensive set of end-game positions with their evaluation values. These values were obtained through a retrograde search that starts from the end positions. This decision proved to be well founded – his program, called Chinook, became a world-champion player. Essentially, this database strongly solves the final end-game of C . The next frontier for Chinook is to solve that part of the end-game ultra strongly; that is to maximise the chance of winning against a fallible opponent from a losing or draw position [57]. A solution to this problem is likely to require some form of machine learning because the playing agent has to obtain and employ knowledge regarding the current opponent's strengths and weaknesses during play.

2.4 Knowledge and search

Given a set of positions that follows from the active position, the playing agent uses game knowledge and look-ahead to identify the most promising move. Look- ahead involves a *game tree search* to find a position that is likely to lead to a win. In order to play a game within reasonable time constraints, it is impractical to use search exclusively. But, because of the longer term tactics required to win, knowledge exploitation on its own is also an insufficient winning strategy. For these reasons, modern playing agents apply both knowledge and search.

The general relationship between knowledge level and search effort is illustrated in Figure 2.1. The vertical axis shows an increase in knowledge (K), and the horisontal an increase in search (S). Three performance levels are in order of increasing performance: p_1, p_2 and p_3. A playing agent at level p_1 can improve its playing performance to level p_2 by increasing its knowledge or by spending more time on search. Better knowledge requires less searching, and more search makes up for weak knowledge.

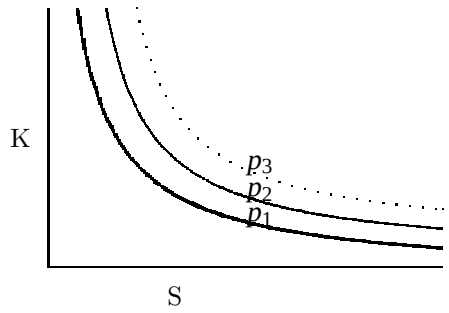


Figure 2.1: Knowledge and Search

The curve drawn in Figure 2.1 is a generalisation that suggests the power of knowledge is roughly equal to the power of search. However, this suggestion is imprecise. Deciding on the correct balance between the time spent to exploit knowledge and the time spent on searching is arguably the most critical decision when implementing a game playing agent.

Deeper search is achieved by optimisation and improving the search algorithm. Optimal search performance is achievable by architecture dependent implementations of algorithms that take advantage of specific multi-processor systems. However, the amount of search conducted during game play remains a function of time. This function is unfortunately not linear because the number of nodes grow exponentially as the search depth increases. In addition, experimental results [55] suggest that the incremental improvement brought about by searching one more ply

declines as the search goes deeper. When the point of diminishing return is reached, or the exponential growth becomes too great, the only way to gain playing performance is to improve the knowledge level of the agent.

The level of knowledge is improved by increasing the volume of the knowledge or by introducing more complex knowledge structures. Large volumes and complicated structures slow down the knowledge driven evaluation process. Another pragmatic hindrance to the use of knowledge is the difficulty of acquiring the knowledge that the agent should use.

Clearly, a learning framework should improve the accuracy as well as quantity of the knowledge used by the playing agent. Such a framework should be able to collect and represent complex knowledge. Ideally, the framework should build knowledge that remain useful when deeper searches are conducted by the playing agent.

2.5 Classification properties of games

Two-player games are available in many forms. There are two reasons to consider these forms as part of the current research. The first is to help with the selection of a subject domain; and the other is to assess the learning framework using games that are not too similar to the subject domain. This section introduces three properties that can be used to classify two-player games. The focus of the first property is the information available to the active player. The second property is called convergence, and it considers the changes to the game tree structure as the game progresses. The third property is the playing complexity of the game.

2.5.1 Information

In a game of pure chance, the active player has no control over the outcome of the game, and no skill is required to win[†]. As the level of control available to the active player increases, the ability of the player has more influence on the game outcome. There are two factors that determine the level of control: the freedom of choice, and state visibility. From these two factors, Schaeffer [56] distinguishes three classes of games: those with elements of chance, those with hidden information, and those without chance or hidden elements.

Games with an element of chance are called *stochastic games.* Chance influences the freedom of choice of the active player in two ways. In the first case, chance places constraints on the states that may follow the active state. The active player does not have at his disposal the

[†]An example of a board game of chance is the game snakes-and-ladders

information that describe the future options available to him or to his opponent. Examples of these games include M , L and B . The second way in which chance influences the freedom of choice is when it determines the final state of the move after the player made a decision. In this case, the active player lacks the information regarding the result of his move decision. A good example of this type of chance, is found in the game *Risk*. Regardless of the way in which chance introduces uncertainty into the game; it always places a constraint on the information available to the active player.

In *hidden information games*, the information is deliberately withheld by keeping some part of the active game state from the active player. In this case the active player is unable to determine the complete game state that follows the current state; rather, he decides how the visible part of the game state changes.

Many games have hidden as well as stochastic elements, and one can add another classification called *hidden-stochastic games* if the need arises. In the literature, games with hidden information, stochastic elements, or both, are called *imperfect-information games*. Most card games fall in this category. For example, in R , the order of the cards in the closed pile provides the element of chance, and the cards held by his opponent is hidden from the active player. However, the active player can decide which card to dispose from his hand, and in so doing, he decides how the part of the game state that is known to him will change.

In *perfect-information games*, the active player has access to the full game state and there is no element of chance to consider. Examples of perfect-information games include C , C and O .

Strategies that win perfect-information games are unlikely to be transferable to imperfect-information games, and *vice versa*. To achieve optimal play in a perfect information game, a *pure strategy* must be followed. In such a strategy, the aim is to select a move that is guarenteed to lead to a win. For optimal play in an imperfect information game it is best to follow a *mixed strategy*. In a mixed strategy the aim is to choose a move from a set of moves, where every element in the set has a non-zero probability of leading to a win [2].

Two reasons support the assertion that pure strategies should be preferred when defining a knowledge based learning framework. The first reason is that a pure strategy demonstrates skill with more conviction than a mixed strategy. It is less likely that the winner in a perfect information game is considered lucky; therefore a pure strategy learner shows improvement in skill levels with greater accuracy. The second reason is that it is easier to acquire knowledge

when a pure strategy is sought. In a stochastic game, a good move followed by bad luck, could be classified by the learner as a bad move. In hidden information games, the move decision is based on an assessment of what the hidden elements contain. A good move may be overlooked because of an incorrect appraisal of the hidden state information. In perfect information games the consequence of a good move (or a bad move) is more predictable.

2.5.2 Convergence

Another property that can be used to classify games is that of convergence. This property, introduced by Allis [2] identifies three classes: converging games, diverging games and unchangeable games. The three classes depend on the frequency at which a particular type of move, called a converging move, occurs; and how that frequency changes as the game progresses.

A *conversion* is a move from position p to position q such that no position in the play-line leading to p can occur in any of the future lines of q. In other words, a conversion changes the game state such that the active state cannot be obtained later in the game. For most games, the main conversions involve adding or removing pieces from play, and Allis [2] provides a definition that takes these moves into account. However, many games have conversions that do not alter the number of pieces in play; such as moving a pawn in C and a checker in C . Therefore, a new definition of a converging game is introduced. Because this definition includes all conversions, it can be applied to more games and as such, it is more general than the definition supplied by Allis.

Definition 2.1: *Conversion space.* The *conversion-space* of position p is the set of all conversions available in the future lines of p. The *conversion potential* of p is the cardinality of its conversion space. Note that every conversion decreases the conversion potential of the game. That means, every game with conversions has a gradual decrease in convergence potential as the game progresses.

Definition 2.2: *Converging game.* A *converging game* is one in which the average branching factor of the positions with the same conversion potential decreases as the conversion potential decreases. If the average branching factor for a game increases as the potential decreases, it is a *diverging game.* A game that does not diverge or converge is an *unchanging game.* In C all moves are conversions, except for the moves of kings at the end-game. Generally, the branching factor decreases, and *Checkers* is a converging game. O and G are diverging because every piece placement is a conversion that increases the future options available to

the players.

For automated learning, there is no general reason to prefer either diverging or converging games. However, it is more practical to implement an end-game database for converging games. Thus, if a learner is expected to learn open or middle game strategies, it would be more reasonable to choose a converging game as the subject domain. End-game databases are available for C , whereas the number of end positions in G makes it close to impossible to create an end-game database of any practical use.

2.5.3 Complexity

Perfect-information games are similar in many respects, but these games cover a wide range of complexity. One of the simplest games to master is T - - , and one of the most challenging games is the game of G . On the complexity scale, somewhere between these two extremes lie the popular games of O , C and C . The advent of world class playing agents brought about innovative new games that are purposefully developed to be difficult for present-day approaches to automate game playing. Examples of these game are O and A .

Complexity, especially the complexity of a game, can be a subjective measurement. Many may think of C as a simple game, but to play this game at grandmaster level requires remarkable skill and talent. One of the best players of our time, Dr. Marion Tinsley, was able to play at a level of 99.6% perfection [55]. This means he would make a less than perfect move only four times out of a thousand.

An objective measurement of complexity seems to require an assessment from the perspective of a grandmaster: that is to consider the complexity of achieving perfect play. Given two games, *A* and *B*, one can consider game *A* to be less complex than game *B* if it takes less effort to learn how to play game *A* perfectly than it would to learn how to play game *B* perfectly. This does not remove the notion of subjectivity from the definition, but a clear line can be drawn between those games that are played perfectly by a computer and those that are not (see Section 2.3.2). Games that are played perfectly could be considered less complex than those that are not solved.

In an attempt to obtain a more precise and more objective measurement, the assumption that complexity is simply a function of the number of positions in the state-space seems reasonable. This suggests that a larger state-space is evidence of a more complex game. Surprisingly, Allis *et. al.* [3] show that this assumption is incorrect using a trivial variant of G . This variant is

played on a 19×19 board, each player moves by placing a stone on an empty square, and it assigns the first person to fill 181 squares as the winner. In this game there are 10^{170} positions in the state-space, but the game is trivial to solve.

Later, Allis [2] identifies two dimensions that contribute to a complete definition of complexity: space complexity and decision complexity:

* **Space complexity**: This is the total number of legal game positions reachable from any initial position of the game (i.e. the cardinality of the state-space). The space complexity of a game can be approximated by obtaining an upper bound and then sharpening that upper bound through the elimination of illegal positions.

* **Decision complexity**: This is the number of nodes in the game forest. The decision complexity is typically much larger than the space complexity because of the duplicate positions in the game tree. The decision complexity can be approximated using the average branching factor and the average game length. This branching factor can either be constant or a function of the depth of the game tree. Allis suggests using tournament games from which the averages can be observed. This approach provides a method in which the decision complexity of games that have tournaments can be compared.

Measuring the complexity of the game to which an automated learner is subjected is an important factor when the learning ability of dissimilar learning agents are compared. As a perfect-information game, the game of C is a moderate choice. It has not been solved, and therefore it cannot be considered simple. On the other hand, a world-class computer player has been developed, indicating that it is one of the less complicated games from the unsolved set.

2.6 Conclusion

The three drivers for using games in research, *i.e.* the creation of a winning agent, the solving of a game and the investigation of machine learning methods, provide a platform for cooperation amongst researchers. However, endeavours that compete to create a winning playing agent can lead to the opposite. The driver of the current research falls in the third category: it investigates the plausibility of establishing a learning framework for two-player games.

A learning framework that aims to obtain the knowledge required to improve the playing performance of an automated agent cannot discard the importance of game tree search during

play. Search serves not only to amend shortcomings in knowledge, but it is also needed to develop (and to counter) longer term tactics during play.

Perfect-information games are more predictable than imperfect information games; and therefore are more suitable for a knowledge based learning agent. Games that converge has fewer moves in the end-game and are therefore more prone to have end-game databases. This leaves the possibility of incorporating such a database to develop opening and middle game knowledge.

The complexity of a game is another important consideration when choosing a subject domain. The game of C is neither too simple, nor is it an example of the most complex games available; and it is a converging game. Consequently, it has been chosen as the initial domain for which the learning framework will be developed.

In the next chapter, the problem of representing the knowledge for two-player board games is explored in detail, and a new representation language is developed. C is used to demonstrate how the new language can be used to express game knowledge.

Chapter 3

Representing game knowledge

In the previous chapter, an introduction to the broader concepts of two-player games were provided. This chapter builds on those concepts by providing a review of the knowledge types and the knowledge representation schemes used by successful learning- and playing agents. It also proposes a new representation language for game knowledge that includes symbols and numbers. A precise definition of this language is provided.

3.1 Introduction

Arthur Samuel's C player [53] uses hand-crafted heuristics to decide on the next move, but he considered it a reasonable research objective to construct a program that could attain useful knowledge automatically. Since then, it became clear that a world-class playing agent could be created by improving the search techniques, and the quest for knowledge become a secondary concern in the field (see Section 2.3.1). However, a close interaction between knowledge exploitation and tree search is required during game play (Section 2.4). Before the knowledge can be exploited it must be acquired and presented to the playing agent in a usable format. If this format, or representation scheme used by the playing agent is also employed by the learning agent, a seamless transfer of knowledge from learner to player is guaranteed. This chapter explores the aspects of representing game knowledge in this way, and defines a new representation scheme that combines symbolic components with numeric elements.

Section 3.2 starts this chapter by considering the aspects that influence the choice of knowledge representation. In order to improve its ability, a playing agent could employ different types of knowledge. These types are described in Section 3.3; and one of these types are selected

for the game learning framework. A question closely related to representation considers the foundation from which the learning agent should gain its knowledge. Section 3.4 identifies the smallest components of representable knowledge. Using these components, this section also introduces the zero-knowledge concept.

Section 3.5 defines an evaluation function and describes precisely how this function will be used by the playing agent. A term in this function encapsulates a knowledge component, and Section 3.6 describes a novel feature language used to express these terms. Finally, Section 3.7 provides a discussion on the new language, and reviews the most important findings of this chapter.

3.2 Representation considerations

In order to choose a representation language for learning, three important aspects must be carefully considered: the definition of the learning problem, the expected quality of the solution and the learning method. In the subsequent paragraphs, these aspects are addressed.

The learning problem

The task of a learning agent (defined in Section 2.1) is to gain the knowledge for the playing agent. In order to gain this knowledge and to make it usable to the playing agent, a clear description is required of the composition of this knowledge. The usefulness of this composition is determined by the number of games to which it can be applied. As a starting point, perfect information games represent the superset of games to which the knowledge must be applied (see Section 2.6). This superset is refined by placing two more constraints on the games. The first is that the game world consists of a countable number of game pieces, and the second constraint is that each piece in play could find itself in no more than one of a countable number of squares*. The subset of perfect information games that satisfies these constraints is referred to as *two-player board games.* This subset forms the subject domain on which the learning framework is founded; and for which the representation scheme is required.

The problem of learning is to improve the skill by which a playing agent makes move decisions through the acquisition of knowledge. Established methods to automate move decisions presuppose some function that obtains a numeric evaluation of a game position. Using this func-

*Actually, the word *places* would be more precise. But most known games have squares; and this association to squares simplifies the discussion a little.

tion, the problem of the playing agent is conceptually reduced to choosing the game position with the highest evaluation; and the problem of the learning agent is to obtain such an evaluation function. The conclusion is that the representation method should incorporate the notion of a function that facilitates the numeric comparison of game positions.

The quality of the solution

The quality of the solution is a measure against an expected outcome. The expected outcome of the envisaged learning process is a player that has shown an increase in winning performance. In addition, there is also an expectation that the knowledge should be comprehensible. The comprehensibility of knowledge relates directly to the chosen representation method. Alternatives fall into two broad categories: black box methods and knowledge oriented methods [37]. The *black box* methods are difficult to interpret and are less comprehensible. The use of neural networks falls in this category. The *knowledge oriented* methods are easier to comprehend, and symbolic representations such as PROLOG clauses fall within this category. Some researchers from both of these categories aim to improve the comprehensibility of learned knowledge. In 1988, Michie [42] identified three broad criteria that can be used to classify the comprehensibility of the representation method used by a system:

* *Weak criterion.* The system uses sample data for improved performance of subsequent data. This criterion is met by all approaches to machine learning, including the black box approaches.
* *Strong criterion.* The weak criterion is satisfied and the system can communicate its updates in explicit symbolic form. In this case, the machine is able to present new information in a form that makes use of symbols.
* *Ultra-strong criterion.* The strong criterion is satisfied and the system can communicate its updates in an operationally effective form. The information presented by the machine must be usable by a person without the aid of a computer.

Note that Michie's definition of the strong criterion does not exclude the use of numbers, but it excludes a representation that has no symbols. If the ultra-strong criterion is met for game knowledge, humans will be able to learn how to improve their play by inspecting the knowledge produced by the program.

The current work aims to produce a learning solution that satisfies the strong criterion. Consequently, the learning agent should use a representation that includes symbols; and it should be able to communicate updates to its knowledge explicitly.

Learning method

The third consideration when choosing a representation is the learning method. For the current study, this learning method is a learning frame that is introduced in chapters that follow this one. However, it would not be practical to define a representation without (at least) a conceptual view on the application of this framework. Conceptually, the learning framework consists of two learning steps. The first step discovers the symbolic knowledge from which a function is composed. The second step obtains optimal values for the numeric weights that is contained by this function. These two steps form part of the greater four stage iterative process of the framework (described in Chapter 5).

The framework places an important constraint on the representation: it specifies that the representation should contain symbolic as well as numeric elements. However, as the current chapter unfolds, it quickly becomes clear that such a representation is ideal for game knowledge. One could therefore conclude that the method is defined for the language, and that the language does not follow from the method. This assessment is not entirely accurate. It is entirely possible to define other representation schemes that employ both symbols and numerics but use the same conceptual framework for learning. Also, an alternative learning framework that uses the representation scheme provided in this chapter could be developed by another researcher in the field. But without the concept of method, the concept of language would have no application; and without the concept of language, the concept of method would have no purpose. In other words, the *concepts* are related; but the realisation of these concepts requires the language to be fully developed first, then the learning method can be detailed and implemented.

In this case, the conceptual framework requires the representation scheme to provide a separation between the symbolic and the numeric elements. This separation should allow symbolic elements to be obtained with little regard to the value of the numeric elements. Furthermore, the alteration of the values of the numeric elements should not change the semantics of the symbolic elements; allowing weight adjustments during optimisation with little regard to the meaning of the symbolic elements.

3.3 Types of game knowledge

The learning process can be regarded as a process of gaining knowledge. In this section a brief overview of the types of knowledge available to two-player game programmers is provided. It is necessary to select from the available types because each type has a different application and is likely to lead to a different representation. In this section, a commitment is made to the knowledge type that is considered to be more fundamental.

To demonstrate what is meant by *more fundamental*, consider the activity of learning how to multiply. If the learner knows how to add, multiplication is easier to grasp - this is because multiplication can be regarded as a sequence of additions. One can consider the knowledge of addition as a component of the knowledge of multiplication. In this sense, addition is more fundamental than multiplication.

The source of the knowledge used during the game play provides a primary classification. Knowledge that was available when the game started is called *static knowledge*, and knowledge gained during game play is *dynamic knowledge*. Dynamic knowledge cannot replace static knowledge. The role of dynamic knowledge is to augment static knowledge: it provides additional information that influence a largely static knowledge driven decision process. Of these two, dynamic is regarded as less fundamental, and is therefore not chosen.

An example of dynamic knowledge is a set of rules that is used to represent the computational model of the opponent. Each time the opponent makes a move these rules are updated. Schaeffer [55] suggested the use of such an opponent model to improve the outcome of a drawn or lost game against a fallible opponent. For instance, a game can be changed from a draw to a win, by employing a tactic that is likely to be more difficult for a particular opponent to counter.

Static knowledge is based on facts that do not change during the game. However, the exclusive use of this knowledge does not imply that the computer player will make the same move in the same situation because the selection algorithm could be non-deterministic. However, with static knowledge, the performance (i.e. the average number of wins) against the same opponent will remain constant.

It is notoriously difficult to gather static knowledge from human experts. A solution to this *knowledge acquisition bottleneck* is to provide a mechanism for the program to learn this knowledge for itself. In general, this goal is elusive, but there are notable successes in the gaming domain. For instance, Logistello, an O program that plays better than any human, used self-play to obtain values for parameters that choose which one of 11 play-patterns to apply.

The patterns themselves were supplied to the program. Another program, developed by Gerald Tesauro, called TD-Gammon, combined self-play and temporal difference learning to train a neural net. The result was a backgammon game comparable to human champions [67].

Static knowledge is further divisible into two subtypes: *database knowledge* and *heuristic knowledge*.

3.3.1 Database knowledge

An advantage the machine has over its human opponent is the ability to use vast amounts of information in a database. The size of the available memory has grown considerably in recent years, enabling game playing programmers to use larger databases. The reigning C champion, Chinook [57], is a playing agent that employs database knowledge extensively. It uses three different databases: an opening book, an antibook and an endgame database. Each one solves a different problem.

The *opening book* solves the problem of strategic planning. Strategic planning is a natural ability of human players, but it is not easy to automate. A weak opening is bound to end in a lose, especially when the program plays against a master player. The opening book is a library of known opening lines, and the planning problem is reduced to making the best choice from this library. Chinook has a large opening book that contains opening lines collected from the C literature. These opening lines have been subjected to automatic verification which resulted in hundreds of corrections and also in some refutations of major lines of play.

The *antibook* is needed when the opening book is automatically extended. Through self play, Chinook searches for new opening lines and it extends the opening book. It often finds interesting opening innovations, but from the C literature, some of these new opening lines are known by human players to be losing. The opening lines that should be avoided are part of the antibook. Using the antibook, Chinook can play its own openings while avoiding play-lines known to be losing.

The largest database Chinook uses is the *endgame database*. This database contains the *game-theoretic value* of every position with eight or fewer pieces on the board (about 4×10^{11} entries). The game-theoretic value is the result of an exhaustive search that marks a position as either won, lost or drawn. When perfect players compete, it is not possible to do better than the game-theoretic value in any future play-line that follows from the computed position. Using the endgame database, Chinook is able to announce the result of a game within 15 moves of the

start [54].

Database knowledge effectively shrinks the problem space. At the start of the game, the program selects only from positions in the stored opening lines, not from all possible moves. Although the problem of selecting the opening line is significant, it is arguably easier than implementing a program with the ability to do strategic planning. The endgame database prunes the game tree: the outcome of the game is determined when a position in the endgame database is reached, and not when an end position is reached. If the opponent is fallible, the program could attempt to improve the outcome from a lost or drawn play-line. But, when it finds a winning play-line, the opponent cannot change the outcome.

Extending the database, especially the opening book is a form of learning. This extension process requires external knowledge because it is impossible to verify the final outcome of an opening line. In order to decide which opening line is better, the last position in the candidate lines can be compared. This comparison requires an evaluation that employs knowledge that is more fundamental than database knowledge. This type of knowledge is discussed next.

3.3.2 Heuristic knowledge

The *heuristic knowledge* employed during game play is based on the premise that characteristic features of a position can be used to choose the position that is more likely to lead to a win than the other positions in a set. In the sections that follow features are explored in more depth, but for now it suffices to consider a feature as a function that divides the state-space into two parts: those positions on which the feature is present, and those that do not exhibit the feature. For two-player perfect information games, there are two dominant approaches to represent and use heuristic knowledge: artificial neural networks and static evaluation functions.

Artificial Neural Networks

The development of training procedures for multi-layer neural networks has been the most important step on the road leading to automatic feature discovery. In a network, the nodes in the hidden layers represent features. Typically, the training of hidden layers is completely unconstrained and it is unclear what concepts will evolve. Three primary design problems influence the knowledge representation scheme of the neural network: encoding the input, the network topology and the meaning of the output.

In game playing programs, the input is the position and the network is provided with an

encoding of the position to evaluate. In many cases, the position is translated directly using trivial encoding methods, such as a bit-string representation for each square. In the C domain, a program called Anaconda is a representative example [12, 11], and it does not use a trivial encoding. Anaconda uses a neural network with 5046 weights, and provides as input, the position presented in a variety of spatial forms [14]. This program achieved expert play levels, and it was trained from no game knowledge other than the game rules.

The topology is defined by the hidden layers and the connections in the network. The number of nodes and the number of layers has a direct influence on the number of inter-dependencies that can be represented by the network. However, the optimal configuration for a particular problem is not known, and usually a commitment to a topology is made after a number of experiments.

It is also possible to discover the configuration of the network automatically. Fogel [21] developed a learning agent that discovers the number of nodes in the hidden layer of a neural network used by a T - - player. In his program that learned O , Moriarty [45] evolved the entire network topology using a genetic algorithm. His learning agent played tournaments against a number of fixed playing agents during training and it discovered the concept of mobility[†]. After this discovery, the evolved player was able to win 70% of the games played against a 3-ply search program that used an evaluation function without mobility features.

The third design problem is a decision of how the output should be represented. A direct approach to use the output of the network is to output a single value that is compared with the output of another position to decide which position is more likely to lead to a win. This has been used by Franken and Engelbrecht in their C program [25]. Anaconda uses this output in a minimax search [14]. In an alternative approach, Moriarty's Othello program used a network that has 64 output units, one for each square on the board. The unit value indicates the strength of the suggestion to move into that square. The neural network of Fogel's T - - player selects the target square that is most likely to lead to a win.

The training of neural networks has been the focus of many machine learning research endeavours. A flagship contribution in this respect was temporal difference learning [67]. TD-Gammon combined self-play and temporal difference learning to train a neural network. The result was a B player that could challenge human champions [54, 67]. The application of evolutionary methods to train neural networks is also an active research area. For

[†]A tactic that aims to limit the opponent's choices

instance, Anaconda uses co-evolution as a learning mechanism.

The primary reason for not using neural networks in the current research has been explained in Section 3.2. However, for typical game playing programs, the comprehensibility of the knowledge is far less important than the ability to play well. For this reason, neural networks will continue to be an integral part of many future computer game players.

Static Evaluation Functions

A static evaluation function is used to compare positions. It has the set of legal positions as domain, and it calculates a floating-point number called the position's *score*. Usually, evaluation functions are linear and have the form $\sum w_i \times f_i$ where each f_i quantifies some feature of the position, and each w_i is the weight assigned to the feature. The value of w_i indicates the relative importance of the feature f_i. The feature with the highest weight is considered to be the most important. *Evaluation function tuning* is the process of determining weight values that produce an optimal function and it is the most extensively studied learning problem in game playing programs [27]. For evaluation functions, the knowledge representation scheme is affected by two design considerations: feature design and the evaluation model.

Samuel [53] explicitly coded his features and based them on a study of C literature. In addition to the literature, Chinook incorporates knowledge acquired from Martin Bryant, a C expert [55]. Existing source code is another source of heuristic knowledge. The hand-crafted knowledge for Fogel's C player was obtained from an inspection of open source C programs [23].

Instead of hand-coding all the details explicitly, Buro used first-order predicate calculus to express the features of the evaluation function he used in GLEM (generalised linear evaluation model) [10]. Compositional representations, such as Buro's are more suitable for knowledge discovery because new features can be constructed by abstraction, regression or specialisation of existing features [27].

Two perspectives on the mathematical model implemented by the evaluation function (i.e. the meaning of the score) lead to two different optimisation strategies: move adaptation and value adaptation [9]. The *move adaptation* strategy aims to mimic moves made by experts. In this case the actual score is not important, as long as a move that is more likely to be chosen by the expert has a better score than a move less likely to be chosen. The *value adaptation* strategy aims to accurately measure how well a position fits a specific model. For instance, in a model

where the evaluation function is defined to predict the outcome, a higher score could indicate that a position has a greater chance of winning.

As a game progresses from its opening and middle game to the endgame phase, the required set of features to make an optimal decision does not remain constant. Therefore, the ideal evaluation function cannot remain the same throughout the game's phases. One way to achieve this goal, is to create a different function for each game phase. This was done for C and for Othello [57, 9]. Chinook's evaluation function has 25 heuristic features and four phases. Each phase has its own set of weights. For O , the disc count[‡] was found to be an adequate measure for the game phase, so the number of phases is the same as the number of moves in an Othello game.

Phased functions introduce a problem of their own. During search, the scores from different evaluation functions are compared to find the optimal move. For the comparison to work, a normalisation method is needed to ensure that a score from one function has precisely the same winning chance as the same score from another function. For example, in C , a small advantage in the opening should outweigh larger advantages in the endgame. This normalisation problem arises only when the search is deep enough to transcend phase boundaries. In Chess programs, the search depths are such that the search does not easily cross phase boundaries, and the issue of game phases and normalisation is less apparent [55].

A function optimised for move adaptation is less suited for phase dependent evaluation than one based on value adaptation. This is because functions optimised by move adaptation have no global interpretation. The scores of value adapted functions can be used as long as the model is chosen such that the position score has a phase independent meaning [9].

3.4 Feature granularity

Michael Buro presented an approach to automate the construction of the evaluation function for Othello [10]. According to Buro, an evaluation function is built in two steps: first the features are selected and then they are combined to form complex expressions. The process of combination includes the merging of selected features and the assignment of floating-point weights that occur in the resulting expression. The first step is difficult for the machine, whereas the processes of combining features and fitting weights are computationally feasible.

When selecting features by hand, the compositional nature of features is soon realised. A

[‡]The total number of pieces on the board

feature consists of smaller, probably less significant ones. Buro exploits this notion and considers *atomic features* as those features that cannot be expressed in terms of other features. Instead of selecting a single complicated feature, a small number of atomic features are selected. The selection of features to be used as building blocks is simpler than selecting the complicated terms of the evaluation function. Buro stresses that ideal building blocks require domain knowledge and although atomic, these features may be complex. In Buro's approach, the process to select features is not automated. Buro's Generalised Linear Evaluation Model (GLEM) employs a representation that is suitable for learning. A brief overview of the evaluation function used in GLEM follows.

In GLEM, the evaluation function e for the position p has the following form,

$$e(p) = g(\sum_{i=1}^{n} w_i \times f_i(p))$$

As a statistical approach, the value of e in GLEM estimates the probability that a position leads to a win, and as such this value must be between zero and one. Because w_i is in $\mathbb{R}$, e cannot be guaranteed to be in $[0, 1]$. The purpose of the link functions, g is to ensure that the value e falls within the required range[§]. Least-squares optimisation on the error of applying e to a set of labelled training positions is used to find values for the weights, w_i for $(1 \leq i \leq n)$. Each f_i is a combination of atomic features, called a *configuration.* A configuration is a binary function that has a value of either 0 or 1. Each weight, w_i indicates the relative significance of the configuration f_i.

A configuration is composed from atomic features that range over $\mathbb{Z}$. For each atomic feature a_i and position p, there is one and only one integer k_i such that $a_i(p) = k_i$. Let A be the set of atomic features, then define relation R_A as $R_A \equiv \{a_i(p) = k_i \mid a_i \in A, k_i \in \mathbb{Z}\}$. In other words, R_A contains the feature-to-value mapping for a position, and $R_A \subset A \times \mathbb{Z}$. In the context of position p, the triple (p, a_i, k), denoted as $r_i(p)$, takes the value *true* if $r_i(p) \in R_A$, otherwise it is *false*. A configuration is a conjunction of these elements. In particular, a configuration c covering n atomic features is denoted as $c = r_1 \wedge r_2 \wedge \ldots \wedge r_n$. The function associated with this feature, f_c takes a value 1 for a position p if and only if $\forall_{i \leq n} r_i(p) = true$. It takes value 0 otherwise.

In order to define a suitable learning framework, a clear commitment needs to be made regarding the level of granularity of the acquired knowledge. Buro left the choice of granularity to the programmer. Before the learning process starts, the program is provided with established

[§]An example of a suitable function for g is $\frac{1}{1+\exp^{-x}}$

features, already known to be influential in the outcome of the game. From the perspective that these features are further decomposable, it can be reasoned that Buro's features are not atomic.

Conceptually, the idea is to start the learner from zero knowledge. However, the term zero knowledge is not absolute. For instance, if a program has learned the rules of a game, it had less knowledge to start with than a program that employs a neural network trained through self play. The latter is currently regarded as a program that learnt from zero knowledge. Maybe in the future, the program that uses zero knowledge is the one that first identifies a particular activity as a game, then learns the rules, and then devises some way to improve its own ability to play.

A sensible choice for a granularity level is to consider the knowledge that can be obtained from visual inspection as atomic. Consider a program that possesses the sense of sight and contains an implementation of the game rules. Using the game rules, visual observations can be transformed into facts that have some relevance to the game. For example, compare a program that sees a red puck on a chequered board to another program that looks at the same thing, but sees the opponent's man on the seventh rank. The first program does not interpret the meaning of what is seen. The second program demonstrates a basic understanding of the rules of C
and it is a plausible subject for machine leaning research. Using this idea, a definition of a zero knowledge agent is formulated:

Definition 3.1: *Zero Knowledge agent.*
A *zero knowledge agent* is a game playing program that is provided with game rules and the ability to describe a game state in terms of those rules. This description does not go beyond that which is observable by the human sense of sight. In particular, zero knowledge does not contain any deductions, estimations or conclusions.

Angeline and Pollack [4] trains a T - - player without giving it the rules for T - - . If an invalid move is made, the player's move is forfeited – giving significant advantage to its opponent. However, this approach is not an example of zero knowledge learning. The agent effectively learns an alternative version of T - -T .

Clearly, an implementation of the GLEM features would not result in a zero knowledge agent, the basic building blocks must be observable. Having smaller building blocks is a natural progression towards the goal of automatic evaluation function construction. It is in fact far simpler to define observable features than GLEM features, but the construction task left to the machine is much more complicated.

3.5 The evaluation function

This section introduces the evaluation function F as the function used by a game playing agent that employs knowledge expressed in $\mathcal{F}$ to make move decisions. F is a static evaluation function (see Section 3.3.2), and as such it compares game positions by mapping these positions onto floating-point numbers. The function is based on the GLEM function (see Section 3.4), and is similar to that function in form. However, unlike GLEM, the value of F does not estimate the probability of winning.

The evaluation function F ranges from 0 to 1, such that 0 is the value of a losing position, 1 the value of a winning position, and for draw positions, F is 0.5. For positions that are not end positions, the range of F is further constrained using a small positive constant, $\delta \in (0, 0.05)$. For non end positions, the value of F is in $[\delta, 1 - \delta]$. Generally, a higher value for F indicates a better position.

The δ constant is introduced to differentiate between an immediate win (or lose) and a high likelihood to lead to a win (or lose). Consider a choice between the n positions, $\{p_1, p_2, \ldots, p_n\}$, and two elements from this set, p_w and p_h. Let p_w be a winning position; thus $F(p_w) = 1$. Let p_h be the best non end position possible, implying $F(p_h) = 1 - \delta$. With a value of 0 for δ (or omitting δ from the definition of F), the agent is unable to distinguish between the p_w and p_h. With δ set to some small positive constant the desired effect is achieved: p_w is regarded as a better position than p_h.

In addition to some arithmetic involving δ, the function F is defined using another function h. The function h is composed from features, f_i, and weights, w_i. Although features are complex structures (fully defined in Section 3.6), it is sufficient for the purpose of this discussion to regard each feature, f_i, as a function that takes a legal position as argument. The range of a feature is $[0, 1]$. Roughly, a value of 1 indicates that the feature is present on the position, and 0 means that the feature is absent from the position. A higher value signifies a better fit of the position to the feature. The weight w_i is a positive floating-point value associated with a feature f_i that indicates the relative importance of f_i. Because the weights have a relative meaning, the weights can be normalised to have a sum of 1 without information loss. Thus, the following equation always holds for the function h with k features:

$$\sum_{i=1}^{k} w_i = 1 \tag{3.1}$$

Let $\mathbb{P}_m$ denote the set of all legal positions that are not end positions. Consider the function $h : \mathbb{P}_m \rightarrow [0, 1]$ that uses k features, such that each feature is denoted by f_i and associated with a weight, w_i ($0 \leq i \leq k$). Then, h is defined for position p as follows:

$$h(p) \equiv \sum_{i=1}^{k} w_i \times f_i(p) \tag{3.2}$$

Note that equation 3.1 restricts the range of h to $[0, 1]$. One way to read the value of h is to regard it as an indication of how well the position fits into the mould defined by the features: a higher value would be a closer fit to the weighted features than a lower value.

The function F provides an abstraction of a position that is used to compare the position to other positions. This abstraction is called the model of F. For example, the function in GLEM models the probability that a position will lead to a win. Like GLEM's function, the model of F must be able to compare any two positions from any of the future play-lines of the current position. As explained in section 3.3.2, the dissimilarity between these positions encourages the use of a value adaptation strategy. Also, the need to have separate functions for different game phases implies that the chosen model has to be phase independent.

The function h manifests a model that measures "feature fitness", where a better fit to the most important features, produces a higher evaluation. For example, if the most important feature is to have many game pieces, the feature counting the number of pieces will have a relatively large weight. Thus, a position with more pieces will be valued higher than a position with fewer pieces.

If the positions from the same ply in the game tree are compared, h would be a suitable evaluation function [27]. However, even with the normalisation constraint of equation 3.1, the model of h is inadequate when sufficiently dissimilar positions are compared. For example, in C the opening positions have no kings, while closing positions typically have only kings. However, it is always better to have kings than to be without kings. The problem is to define a model for the function that allows a meaningful comparison between the opening position on one play-line and a closing game position on a different play-line, even when both these positions have the same number of kings.

The solution to this problem lies in the observation that position strength is a measurement relative to the opposing player. Function h can be evaluated from each player's perspective, and the player with the higher value has the upper hand. If position a in the opening sequence fits

the features from the agent's perspective better, and position b is a better fit from the opponent's perspective, a belongs to a play-line that is better than b's play-line[¶].

This idea can be incorporated into the evaluation function by introducing a new concept: a *flip feature.* The flip feature of the feature f_i, denoted as $\overline{f}_i$, is f_i expressed from the perspective of the opponent. For example, let f_1 indicate "the opponent has kings". If the active player is red, this feature translates to "there are white kings", and the corresponding flip feature, $\overline{f_1}$ is "there are red kings". Flip features are used to compose the *flip function* $\overline{h}$:

$$\overline{h}(p) \equiv \sum_{i=1}^{k} w_i \times \overline{f}_i(p) \tag{3.3}$$

The expression to determine values that can be used to compare two dissimilar positions, combines h and $\overline{h}$: for a position p, the value is $(h(p)/(h(p) + \overline{h}(p))$. If viewed from the active perspective, a value higher than 0.5 indicates the active player has the upper hand, and a value of lower than 0.5 indicates the passive player has the upper hand.

Finally, these concepts are combined to form the evaluation function F. The maximum value of the expression, $(h(p)/(h(p) + \overline{h}(p))$ is 1, and it occurs when the flip function evaluates to 0. In order to keep F within the bounds, $[\delta, 1 - \delta]$, the value of this term is offset by δ after deflating it with $(1 - 2\delta)$. Let $\mathbb{P}$ denote the set of all legal positions, then $F : \mathbb{P} \rightarrow [0, 1]$ for position p is defined as follows:

$$F(p) \equiv \begin{cases} 0 \text{ if } p \text{ is a losing position} \\ 0.5 \text{ if } p \text{ is a draw position} \\ 1 \text{ if } p \text{ is a winning position} \\ \delta + \left(\frac{h(p)}{h(p)+\overline{h}(p)}\right) \times (1 - 2\delta) \text{ otherwise} \end{cases} \tag{3.4}$$

3.6 The feature language

In the section above, the function F has been introduced as a method to compare board positions in order to select the next move in a play-line. In this section the building blocks of F are described as expressions in a formal language, denoted by $\mathcal{F}$. All features are expressed and evaluated according to the rules of $\mathcal{F}$. The language can be applied to most perfect information, two-player board games. The game C is used to provide a concrete example of such an application.

[¶]assuming h is accurate

As a formal language, $\mathcal{F}$ consists of symbols and operators. These symbols and operators are combined to form *well-formed formulas*. These formulas, or *wff*s conform to strict syntax rules and have precise semantics. The semantics define the meaning of the *wff*. A feature is in itself a *wff*, and like all other *wff*s, the meaning of a feature is the value the *wff* takes in the context of a given position. This value is called the *valuation* of the *wff* and is defined by the function $V : \mathbb{P} \times wff \rightarrow [0, 1]$. For position $p \in \mathbb{P}$ and a *wff* $\mathcal{A}$, the valuation is given by $V(p, \mathcal{A})$.

In this section the syntax and the valuation for all the formulas in $\mathcal{F}$ is provided. Section 3.6.1 defines the symbols that are used to refer to squares, pieces and the relationship between the two. The two ground features of $\mathcal{F}$ are defined in section 3.6.2, namely the atomic occupation feature and the fuzzy occupation feature. These ground features are combined to construct more complex features using the operators defined in section 3.6.3.

3.6.1 Symbols

The *occupation state* of a square is the most specific observation on the board. The symbols in $\mathcal{F}$ identify and describe these states. The set of occupation states is determined from the game pieces and the game rules, and it is therefore different for every board game. For the language $\mathcal{F}$ the set of occupation states is denoted as $\mathbb{O}$.

The set $\mathbb{O}$ has a finite number of elements. $\mathcal{F}$ places two constraints on the elements in $\mathbb{O}$: 1) Every playable square (i.e. squares that can be occupied during the game) on a position must be occupied with one and only one state. 2) No state is impossible – that is; for every state $o \in \mathbb{O}$, a legal position exists that contains one or more squares occupied with o. These constraints ensure that a functional mapping is possible from every square in any position to an element in the occupation set.

The method used to determine the occupation states for a particular game starts with the composition of a list of the simplest Boolean properties that describe a square on a position. When this list is ordered, any square in a position can be described by a binary string of values, where each character corresponds to the value of this square for the Boolean property. This binary string is an occupation state. However, the Boolean properties are not disjoint, and consequently the number of occupation states for n Boolean properties is typically much smaller than 2^n. The final set is obtained by eliminating combinations that are impossible.

For the game of C , 10 Boolean properties are identified and listed in table 3.1. From the rules of C , the dependencies amongst these properties are quite clear. For

Table 3.1: C properties a for square *s*

b_1	*s* is occupied
b_2	An active piece can jump to *s*
b_3	An active piece can move to *s*
b_4	A passive piece can jump to *s*
b_5	A passive piece can move to *s*
b_6	The piece on *s* is active
b_7	The piece on *s* is a king
b_8	The piece on *s* can be jumped
b_9	The piece on *s* can move
b_{10}	The piece on *s* can jump

example, if b_7 is *true*, b_1 must be *true*. Also, b_2, b_3, b_4 and b_5 must be false. After eliminating all impossible combinations, only 44 occupation states are found in the set $\mathbb{O}$ for C . Instead of using clumsy binary strings as symbols for these occupation states, a short mnemonic string is employed. Table 3.2 shows the 44 combinations. The mnemonic identifier is shown in the last column. In other columns 'T' indicates the property is *true* and 'F' indicates *false*. The '-' indicates the value of the property is not relevant in this combination. In table 3.3 the meaning of the mnemonics is supplied. As a shorthand notation, the suffix '*' is used to refer to a mnemonic group of occupation states.

The 44 symbols are the finest (i.e. least granular) selection of occupation states that can be derived from the given Boolean properties. Another valid, but more granular choice is to use the 7 mnemonic prefixes as occupation types. An inspection of the hand-coded features of Samuel's C program [53] indicates that the mnemonic table may be adequate. However, Samuel's features are defined from expert knowledge, while the 44 occupation states are derived directly from the game rules. Thus, from the definition of a zero knowledge agent (Section 3.4) it follows that the set of 44 elements in $\mathbb{O}$ is correct for C .

The agent uses states to refine its "perception" of a game position. Board diagram 6.1 (page 39) shows how the occupation states are used to view one of the moves available from the initial position. As indicated in the diagram, a position is always viewed from the passive player's perspective. This is because the evaluation of the position occurs after the move under evaluation is applied. For example, if the agent plays red, the red pieces will be perceived as the passive pieces on the position provided as argument to the evaluation function. The diagram shows that the occupation states facilitate observations that provide rich and useful information.

Table 3.2: C occupation states

b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	Id
F	F	F	F	F	-	-	-	-	-	n
F	F	T	F	F	-	-	-	-	-	a1
F	T	F	F	F	-	-	-	-	-	a2
F	T	T	F	F	-	-	-	-	-	a3
F	F	F	F	T	-	-	-	-	-	p1
F	F	F	T	F	-	-	-	-	-	p2
F	F	F	T	T	-	-	-	-	-	p3
F	F	T	F	T	-	-	-	-	-	b1
F	F	T	T	F	-	-	-	-	-	b2
F	F	T	T	T	-	-	-	-	-	b3
F	T	F	F	T	-	-	-	-	-	b4
F	T	F	T	F	-	-	-	-	-	b5
F	T	F	T	T	-	-	-	-	-	b6
F	T	T	F	T	-	-	-	-	-	b7
F	T	T	T	F	-	-	-	-	-	b8
F	T	T	T	T	-	-	-	-	-	b9
T	-	-	-	-	T	F	F	F	F	x0
T	-	-	-	-	T	F	F	F	T	x1
T	-	-	-	-	T	F	F	T	F	x2
T	-	-	-	-	T	F	F	T	T	x3
T	-	-	-	-	T	F	T	F	F	x4
T	-	-	-	-	T	F	T	F	T	x5
T	-	-	-	-	T	F	T	T	F	x6
T	-	-	-	-	T	F	T	T	T	x7
T	-	-	-	-	T	T	F	F	F	X0
T	-	-	-	-	T	T	F	F	T	X1
T	-	-	-	-	T	T	F	T	F	X2
T	-	-	-	-	T	T	F	T	T	X3
T	-	-	-	-	T	T	T	F	F	X4
T	-	-	-	-	T	T	T	F	T	X5
T	-	-	-	-	T	T	T	T	F	X6
T	-	-	-	-	T	T	T	T	T	X7
T	-	-	-	-	F	F	F	F	F	o0
T	-	-	-	-	F	F	F	F	T	o1
T	-	-	-	-	F	F	F	T	F	o2
T	-	-	-	-	F	F	F	T	T	o3
T	-	-	-	-	F	F	T	F	F	o4
T	-	-	-	-	F	F	T	F	T	o5
T	-	-	-	-	F	F	T	T	F	o6
T	-	-	-	-	F	F	T	T	T	o7
T	-	-	-	-	F	T	F	F	F	O0
T	-	-	-	-	F	T	F	F	T	O1
T	-	-	-	-	F	T	F	T	F	O2
T	-	-	-	-	F	T	F	T	T	O3
T	-	-	-	-	F	T	T	F	F	O4
T	-	-	-	-	F	T	T	F	T	O5
T	-	-	-	-	F	T	T	T	F	O6
T	-	-	-	-	F	T	T	T	T	O7

Table 3.3: Mnemonic meanings for C symbols

Symbol(s)	Meaning
n	Noman's land
a*	Active field
p*	Passive field
x*	Active man
X*	Active king
O*	Opponent's man
o*	Opponent's king

Active (o)

8		o0		o0		o0		o0
7	o0		o0		o0		o0	
6		o2		o2		o2		o2
5	b5		b5		a2		a2	
4		x2		p2		p2		p2
3	x2		p2		x2		x2	
2		x2		x2		x0		x0
1	x0		x0		x0		x0	
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Passive (x)

Board diagram 6.1

Clearly, there is a need to address a square and therefore a notation to represent squares is required. In $\mathcal{F}$ a square is represented by an ordered pair (f, r) where f denotes the *file* and r the *rank* of the square. At the bottom-left corner of the board the square $(1, 1)$ is found, the file increases to the right and rank increases upwards. In many games some squares, called *non-playable* squares can never be occupied (such as C and Arimaa). The symbol $\mathbb{S}$ is used to denote the set of *playable* squares.

In $\mathcal{F}$, the symbol function $S : \mathbb{P} \times \mathbb{S} \rightarrow \mathbb{O}$ maps the square in a position to an occupation state. For any position $p \in \mathbb{P}$ and square $s \in \mathbb{S}$, and for $o \in \mathbb{O}$, where $O(p, s) = o$, there is no element in $\{\mathbb{O} - o\}$ that also describes the state of s in p.

3.6.2 Features

A feature is a floating-point valued function with the set of legal positions, $\mathbb{P}$, as domain and a range of $[0, 1]$. A value of 1 indicates that the feature is present on the position, and a value of 0 indicates an absence of the feature. For two features $\mathcal{A}$ and $\mathcal{B}$, the valuation with respect to position p is represented as $V(p, \mathcal{A})$ and $V(p, \mathcal{B})$ respectively. If $V(p, \mathcal{A}) > V(p, \mathcal{B})$ then p is a *closer fit* to $\mathcal{A}$ than it is to $\mathcal{B}$. In this section the two ground features, namely the atomic occupation feature and the fuzzy occupation feature are introduced.

Definition 3.2: *Atomic Occupation Feature.* The *atomic occupation feature* epitomises a square in terms of its occupation state. An atomic occupation feature is a *wff* with the following form,

$$(o \oplus s)$$

where $o \in \mathbb{O}$ and $s \in \mathbb{S}$.

For a position, p, $V(p, (o \oplus s))$ is 1 if $O(p, s) = t$. If $O(p, s) \neq t$, $V(p, (o \oplus s))$ is 0.

As an example, consider the atomic feature $(x \oplus (1, 1))$. On Board diagram 6.1, this feature evaluates to 1.

The atomic occupation feature is binary - either the square is occupied as specified or not. However, some knowledge requires fuzzy definitions. For example, consider the representation for this knowledge: *the first rank is occupied by active pieces.* The value of this feature ranges from 0 to 1, adding 0.25 for each square in the first rank occupied with an x or an X.

From the example, one may conclude that the value of this feature must be computed as $\left(\frac{\text{count of matching squares}}{\text{count of squares}}\right)$, however, this is not accurate. When the number of squares exceeds the maximum possible matches, the value may never be able to get close to 1. For example, there can be at most 12 red pieces on a C position, and if all squares are included, the highest value of the feature is $\frac{12}{32} \approx 0.38$. According to this value, the feature is not present on the position. The fuzzy occupation feature considers the fact that there are at most 12 red pieces, and concludes that the value for this position is $\frac{12}{12} = 1$.

Before defining the fuzzy feature, a few new concepts must be introduced. These are cardinality, size dependence, cardinality sets and maximum cardinality. These concepts relate to the occupation states and are derived directly from the game rules.

The *cardinality* of an occupation state is the number of squares on the board that is occupied

with that state. For position p, the cardinality of $o \in \mathbb{O}$, denoted as $|o|$ is defined as follows:

$$|o| \equiv \sum_{s \in \mathbb{S}} V(p, o \oplus s)$$

The *maximum cardinality* of an occupation state is an upper bound on the cardinality of the occupation state. It is the maximum number of squares that can be occupied with that state on a non endgame position. For $o \in \mathbb{O}$ with maximum cardinality, denoted as $|o|_m$, the following holds:

$$\forall_{p \in \mathbb{P}}(|o| \leq |o|_m)$$

In table 3.4, the maximum cardinality for each C occupation state is shown. Motivation for the first entry in the left column is provided. The others have a similar derivation method.

Two occupation states, o_1 and o_2 are *size dependent* when there is some integer k such that $0 < k < |o_1|_m + |o_2|_m$ and $|o_1| + |o_2| \leq k$ always hold. In general, a set of occupation states, $\mathbf{O}$, is size dependent if the following holds:

$$\exists_{k \in \mathbb{Z}}(0 < k < \sum_{o \in \mathbf{O}} |o|_m), (\sum_{o \in \mathbf{O}} |o|) \leq k$$

In the expression above, k is an upper bound on the maximum cardinality of the set. Ideally, the smallest possible value for k must be chosen as the maximum cardinality value. Using a value closer to the smallest upper bound results in a more accurate evaluation of the fuzzy feature.

A *cardinality set* is a set of size dependent occupation states. The collection of cardinality sets, denoted as $\mathbb{C}$ must be disjoint. To keep to the zero knowledge principle, these sets must follow directly from the game rules. Furthermore, the union of these sets must cover the set of occupation states (i.e. $\cup_{\mathbf{C} \in \mathbb{C}} \mathbf{C} = \mathbb{O}$). The maximum cardinality of a cardinality set is an upper bound on the number of squares that can be occupied by any occupation state in the set on a non end-game position. For $\mathbf{C} \in \mathbb{C}$, the maximum cardinality is denoted as $|\mathbf{C}|_m$. Note that the maximum cardinality of a set is less than the sum of the maximum cardinalities of the elements in the set:

$$\forall_{\mathbf{C} \in \mathbb{C}}(|\mathbf{C}|_m < \sum_{o \in \mathbf{c}} |o|_m)$$

For example, consider two simple occupation states of a C position: o_k indicates *the*

Table 3.4: Maximum cardinality of C occupation states

Occupation states	Maximum Cardinality
n	28. There are 32 squares. At least 2 pieces on non end positions, 30. These two take up at 1 square of field, leaving a maximum of 28.
a1, p1	26. Consider a configuration with a king in the middle and 4 open squares around it. This configuration takes up 5 squares and has 4 a1's. There are 32 squares, so no more than 6 such configurations are possible giving 24 a1's. Assuming that the 2 remaining squares end up as a1, the value of 26 is obtained.
a2, p2	17. Using the same idea as above, but this time the size of the configuration that has 4 a2's is 9. Three of these fits in 32 squares with 5 spares.
a3, p3	14. Expanding on the configuration above, adding 4 squares makes 13 for 4 configurations. Two of these make 26, leaving 6 squares.
b1	12.Requires a configuration of 3 squares for one b1, using 30 for 10 b1s and a remainder of 2
b2, b4,x3,x5,X3,X5, o3,o5,O3,O5	8. Requires 4 squares in a configuration that yields 1
b3, b8,x7,X7,o7,O7	7. Requires 6 squares in a configuration that yields 1 b3
b6, b5 ,b7	7. Requires 5 squares in a configuration that yields 1 b6
b9	8. Requires 7 squares in a configuration that provides 1 b9.
x0,x1,x2,x4,x6, X0,X1,X2,X4,X6, o0,o1,o2,o4,o6, O0,O1,O2,X4,O6	12. The maximum number of pieces on a side is 12.

square is occupied by an active king and o_m that *the square is occupied by an active man.* The maximum cardinality of these occupation states is 12. However, no position can contain more than 12 active pieces, so $|o_m| + |o_k| \leq 12$ holds for any C position. This means the two occupation states are size dependent. The set $\{o_m, o_k\}$ is a cardinality set of C and it has a maximum cardinality of 12.

The cardinality sets for C , along with the maximum cardinality for each set are shown in table 3.5. The maximum cardinality values of the cardinality sets are used to determine

Table 3.5: C cardinality sets, together with their respective maximum cardinalities

Set	Elements	Maximum Cardinality
$\mathbf{C_1}$	n,a*,p*,b*	30. At least two squares must be occupied
$\mathbf{C_2}$	x*,X*	12. There are at most 12 active pieces
$\mathbf{C_3}$	o*,O*	12. There are at most 12 passive pieces

the maximum number of squares that can be occupied by any element from an arbitrary set, $\mathbf{O} \mid \mathbf{O} \subseteq \mathbb{O}$. This maximum number is referred to as the *upper limit* of the set, and it is denoted as $|\mathbf{O}|_u$. A recursive function is used to determine this upper limit, for a set $\mathbf{O}$:

$$|\mathbf{O}|_u \equiv \begin{cases} 0 & \text{if } \mathbf{O} = \emptyset \\ |\mathbf{O}/\mathbf{C}|_u + & \text{where} \\ \min\{|\mathbf{C}|_m, \sum_{o \in \mathbf{T}} |o|_m\} & \mathbf{C} \in \mathbb{C}, \mathbf{T} = \mathbf{O} \cap \mathbf{C}, \mathbf{T} \neq \emptyset \end{cases}$$

As an example consider the set of occupation states $\mathbf{O} = \{a3, b1, x3, X7\}$. In order to determine $|\mathbf{O}|_u$, a cardinality set from table 3.5 must be chosen such that the elements in $\mathbf{O}$ are also in the cardinality sets. There are two valid selections: $\mathbf{C_1}$ and $\mathbf{C_2}$. The order is not important, and

below, $\mathbf{C_1}$ is chosen first:

$$\begin{aligned}
|\mathbf{O}|_u &= |\mathbf{O}/\mathbf{C_1}|_u + \min\{|\mathbf{C_1}|_m, \textstyle\sum_{o\in\mathbf{O}\cap\mathbf{C_1}} |o|_m\} \\
&= |\{x3, X7\}|_u + \{30, 14 + 12\} \\
&= |\{x3, X7\}|_u + 26 \\
&= |\{x3, X7\}/\mathbf{C_2}|_u + \\
&\quad \min\{|\mathbf{C_2}|_m, \textstyle\sum_{o\in\{x3,X7\}\cap\mathbf{C_2}} |o|_m\} + 26 \\
&= |\emptyset|_u + \min\{12, 8 + 7\} + 26 \\
&= 0 + 12 + 26 \\
&= 38
\end{aligned}$$

As this example illustrates, the upper limit can be larger than the cardinality of the set of playable squares, $|\mathbb{S}|$. The definition of the fuzzy occupation feature given below takes this into account:

Definition 3.3: *Fuzzy Occupation Feature.* Consider a set of squares, **S** ($\mathbf{S} \subseteq \mathbb{S}$) and a set of occupation states, **O** ($\mathbf{O} \subseteq \mathbb{O}$). A *fuzzy occupation feature* provides a measure of the fraction of squares in **S**, that is occupied with an occupation state found in **O**. A fuzzy occupation feature is a *wff* with the form,

$$(\{t_1, t_1, \ldots, t_k\} \otimes \{s_1, s_2, \ldots, s_n\})$$

where $\forall_{1\leq i\leq k} t_i \in \mathbb{O}$ and $\forall_{1\leq i\leq n} s_i \in \mathbb{S}$.

Let $\mathbf{O} = \{t_1, t_1, \ldots, t_k\}$ and $\mathbf{S} = \{s_1, s_2, \ldots, s_n\}$. Then for position p the value for $V(p, (\mathbf{O}\otimes\mathbf{S}))$ is calculated as follows:

$$\frac{\sum_{i=1}^{k}\sum_{j=1}^{n}(o_i \oplus s_j)}{\min\{|\mathbf{S}|, |\mathbf{O}|_u\}}$$

Consider the feature, *the first rank is occupied by active pieces.* This feature is expressed in $\mathcal{F}$ as $(\{o*, O*\} \otimes \{(1,1),(3,1),(5,1),(7,1)\})$ and it has value of 1 with respect to the board in Board diagram 6.1.

3.6.3 Fuzzy Operators

A feature is seldom considered in isolation. The utility of one observation often depends on the availability of another feature. The following strategy demonstrates this idea: *if the opponent has no kings, keep as many men in the first rank as you can.* This strategy cannot be expressed

if the feature language is unable to indicate an association between the occupation feature in the first rank and the occupation feature regarding the opponent's kings. To solve this problem, disjunction and conjunction operators are introduced. The negation operator facilitates the use of negative features. This operator is used in expressions that indicate the absence of some aspect of a position.

Definition 3.4: *Disjunction.* A *disjunction*, denoted by $\vee$ combines two *wff*s to form another *wff*, and it has the following form,

$$(\mathcal{A} \vee \mathcal{B})$$

where $\mathcal{A}$ and $\mathcal{B}$ are *wff*s.

If $V(p, \mathcal{A}) = a$ and $V(p, \mathcal{B}) = b$ for a position p, then $V(p, (\mathcal{A} \vee \mathcal{B}))$ evaluates to $\max\{a, b\}$. The disjunction operator, $\vee$ combines two *wff*s, $\mathcal{A}$ and $\mathcal{B}$, such that the value of $(\mathcal{A} \vee \mathcal{B})$ is 1 if either $\mathcal{A}$ or $\mathcal{B}$ has a value of 1.

Definition 3.5: *Conjunction.* A *conjunction*, denoted by $\wedge$ combines two *wff*s to form another *wff*, and it has the following form,

$$(\mathcal{A} \wedge \mathcal{B})$$

where $\mathcal{A}$ and $\mathcal{B}$ are *wff*s.

If $V(p, \mathcal{A}) = a$ and $V(p, \mathcal{B}) = b$ for a position p, then $V(p, (\mathcal{A} \wedge \mathcal{B}))$ evaluates to $\min\{a, b\}$. The conjunction operator, $\wedge$ combines *wff*s, $\mathcal{A}$ and $\mathcal{B}$, such that the value of $(\mathcal{A} \wedge \mathcal{B})$ is 0 if either $\mathcal{A}$ or $\mathcal{B}$ has a value of 0. It evaluates to 1 if and only if $\mathcal{A}$ and $\mathcal{B}$ has a value of 1.

Definition 3.6: *Negation.* A *negation*, denoted by $\neg$ inverts the value of a *wff* to form another *wff*, and it is used in following form,

$$(\neg\mathcal{A})$$

where $\mathcal{A}$ is a *wff*. For a position p, $V(p, (\neg\mathcal{A}))$ evaluates to $1 - V(p, \mathcal{A})$ The negation operator, $\neg$ is a unary operator, and the value of $\neg\mathcal{A}$ is 1 if $\mathcal{A}$ has a value 0 and it has the value of 0 if $\mathcal{A}$ is 1.

Instead of operator precedence rules, $\mathcal{F}$ employs parentheses to eliminate ambiguity. Here follows the syntax of all the operators in $\mathcal{F}$ written in Backus-Naur form,

```
<formula> → ( ¬ <formula> )
          |  ( <formula> ∨ <formula> )
          |  ( <formula>  ∧  <formula> )
          |  ( <ground> )

<ground>  → <square> ⊕ <state>
          |  <square set> ⊗ <state set>
```

3.7 Conclusion

F is an evaluation function that provides an effective and a simple method to capture heuristic knowledge in terms of significant features. Each feature is associated with a weight that indicates its relative importance. F is used to map game positions to floating-point numbers, so that these numbers can be used to identify the best amongst various alternative moves. An useful characteristic of the function is that it can be used to compare positions that are very different.

Central to the definition of the evaluation function is the language $\mathcal{F}$ devised to be used in a learning framework. This language is able to represent knowledge to a game playing agent, such that the knowledge can be used in the evaluation function, F. The flexibility of the language $\mathcal{F}$ stems from the level of granularity chosen for the basic building blocks of the expressions, and also from the notion that the operators of the language allow for arbitrarily complex expressions. The classification of feature forms that cannot be expressed in $\mathcal{F}$ could lead to a more complete language definition. However, a more intricate language could complicate the knowledge discovery process.

The use of Boolean operators ensures that known techniques can be used to construct, manipulate and evaluate expressions. For instance, it is possible to translate $\mathcal{F}$ expressions to a disjunctive normal form, and to use and-or tree structures to store these expressions. In addition, the weights in the evaluation function F are easy to separate from the function expression. This allows the use of numerical optimisation methods like Particle Swarm Optimisation (PSO) [33] (the PSO method is detailed in Chapter 7).

An important notion is the fit-to-features model used by F. This model provides an opportunity to exploit and to further develop selective game tree search algorithms that do not assume the compared positions are on the same ply. In the next chapter, the search activity is explored

in more detail; and these concepts and the benefit of this model becomes clearer.

Chapter 4

Exploring the game tree

The previous chapter focussed on how knowledge can be presented and used by the playing agent to make a move decision. In this chapter, the manner in which the agent applies that knowledge to search the game tree is discussed. A new search method called the best-first shallow search is presented and evaluated.

4.1 Introduction

Chapter 2 described the game tree structure and discussed the interaction between the knowledge exploitation process and the search process when a playing agent makes a move decision. In the previous chapter, different types of knowledge were introduced and a knowledge representation language was developed. In this chapter the search process is defined in detail, and the possibility of using knowledge to guide the search path is explored.

Search algorithms can be classified into *fixed-depth search* and *selective search* algorithms. Fixed-depth algorithms explore the game tree to a specified depth. When the depth increases, the time to make a move decision increases dramatically, especially for games with high branching factors. Selective search algorithms avoid this problem by searching some branches in the tree deeper than others. The difficulty faced by these algorithms lies in the decision to abort a particular branch and selecting another branch in the tree to explore further.

Many variations of fixed-depth and selective algorithms exist. In practise the two classes of search algorithms are often mixed; for instance switching over to a selective search when a node on the last ply of a fixed-depth search meets certain conditions. This chapter discusses two

fundamental methods from which many of the other approaches were developed*. The reason for not exploring the available variety of search techniques is that the focus of this work is on knowledge discovery, and the application of a complicated search method could dissipate the conclusions of this study.

Section 4.2 introduces the minimax algorithm and its companion, alpha-beta pruning. This popular algorithm searches the tree to a fixed depth and it does not employ knowledge to decide how the search through the tree must be conducted. A search method that does employ knowledge to direct the search, the best-first minimax algorithm is described in Section 4.3, together with the outline of a recursive algorithm that implements this approach. Shortcomings of best-first minimax is discussed in Section 4.4, and an improvement developed by other researchers is described. Section 4.5 introduces a new search algorithm called best-first shallow search.

The first experiment of this chapter is described in Section 4.6. This experiment establishes baseline values that describe the behaviour of alpha-beta at various search depths. These baselines are used in Section 4.7 to interpret the outcome of an experiment conducted to compare alpha-beta with best-first minimax and best-first shallow search. Section 4.8 describes the last experiment in this chapter that compares best-first shallow search with best-first minimax. Finally, Section 4.9 highlights the important remarks and conclusions of this chapter.

4.2 The minimax approach to tree searching

Searching is the key to many artificial intelligence problems. The typical approach involves representing the solution domain as a (possibly infinite) set of vertices and a set of edges that connects these vertices to each other [49, 46]. These edges and vertices form a *graph*, and the search process involves walking through this graph until a satisfying solution is found.

Even though the game tree is also a graph, general graph search methods do not apply to the game tree. The game tree is constructed ply by ply, where each alternating ply represents the moves available to a different player. A general search method might find a 'winning' position for the active player in this graph; and the agent could choose to move to the first position in the play-line that leads to the chosen vertex. However, that play-line is likely to present move opportunities to the opponent such that the 'winning' position is never reached.

The *minimax* algorithm† takes advantage of the alternating layers in the game tree to define

*Refer to [27] for an overview of significant innovations in game tree search algorithms

†Minimax is a well known algorithm, but it is difficult to trace its origin. It is described in many A.I. text books. See [52] for an example.

a simple, but effective, playing strategy. This strategy assumes that the positions in the game tree can be evaluated. The value of a position is called the *static value*, and it is a floating-point value such that a position with a greater advtantage to the active player has a greater value. The evaluation function described in Section 3.5 provides some insight into how such a value can be obtained.

The minimax algorithm searches the game tree to a fixed depth. At a depth of one ply, only the first position of all future play-lines are considered. At this ply, the active player chooses the position with the maximum score. When the search depth is two, consideration is given to the next ply, and it might become apparent that a counter move is available to the opponent that would eliminate the advantage implied by the score of the first move. The opponent's best move is to choose the future play-line that starts with the lowest scoring position, *i.e.* the position that has the lowest advantage to the active player. Generally, the passive player always minimises the score and the active player maximises the score.

The terminal nodes of the minimax search tree are the positions in the last ply and the end positions that were encountered during the search. A node in the minimax search tree is a *minimising node* if it is on a position from which the opponent must make a move. A *maximising node* is a node on a position from which the active player must make a move. Therefore, all the nodes in an even ply are maximising nodes, and those in an odd ply are minimising nodes.

The key to the minimax algorithm is the manner in which the score is propagated upward from the last ply to the first position in the future play-line. Every node in the minimax search tree has a *minimax value*. The minimax value of a terminal node is equal to the score of the associated position. The minimax value of a minimising node is the smallest minimax value found in its child nodes. Likewise, the minimax value of a maximising node is the greatest value found in its child nodes. The active player chooses the node with the highest minimax value in the first ply as the next move.

Interestingly, there is no need to visit every node in the minimax tree to determine the correct minimax value of every node in the first ply. Hart and Edwards [29] developed the *alpha-beta* heuristic that prunes the branches from the minimax search tree that will not effect the outcome of the move decision. The reasoning for alpha-beta is as follows: if a child of a maximising node has a minimax value less than the value of the child of a minimising node with a smaller ply, the maximising node must not be explored further. There is no need to continue

the exploration because the minimising player would have taken the move at the higher level. The same reasoning is also extended to the minimising nodes - if a child of a minimising node has a minimax value that is greater than the value of the child of a maximising node higher up in the tree, the exploration of the minimising node is discontinued.

Alpha-beta obtains the same results that a full minimax tree traversal would obtain, but it requires much less effort. Therefore, applications for minimax and research activities that involves minimax tree searching typically use an implementation that employs alpha-beta pruning as the reference minimax algorithm.

4.3 Best-first minimax search

The *best-first minimax* algorithm formulated by Korf and Chickering [35] is a selective algorithm that selects the best play-line discovered so far to explore further. Best-first minimax has been shown to outperform alpha-beta on the game of O . Korf and Chickering concludes that best-first minimax performs best when only relatively shallow searches are feasible, and in games with accurate evaluation functions. It is likely to be most valuable in games with large branching factors and/or expensive evaluations. These games (such as G) are those in which computers have been least successful against humans.

Best-first minimax has much in common with minimax. It starts with the current position as root node, and expands the tree as the search continues. With every expansion, the static value of a new node is determined by passing the position it represents as argument to the evaluation function. This value is then propagated towards the root node using the minimax method. If the parent is in a maximising ply, its value is set to the greatest value found in its children. If it is in a minimising ply, the value is set to the smallest found in the child nodes. The root node is in a maximising ply, and plies beneath it alternate between minimising and maximising. At the end of the search, a child of the root node with the highest value is selected as the next position in the play-line.

The difference between minimax and best-first minimax is in the way the next node is selected to expand. Minimax does a breadth-first scan through the game tree. In contrast, best-first minimax applies the knowledge encapsulated in the evaluation function to decide which node in the search tree must be expanded next. The strategy is a simple one: always explore the most promising path in the search tree, further.

The *principal variation* is the path from the root node to a leaf where every node in the path

has the same value. The *principal leaf* is the leaf node of a principal variation and its static value is propagated to the root node. Best-first minimax always selects the current principal leaf node for expansion. Using this approach, the search can stop at any time, and the best move found thus far can be chosen by the playing agent.

Because the principal leaf is always expanded, it is possible that best-first minimax could explore a single path to the exclusion of all others. Korf and Chickering note that this does not occur in practise because of the *tempo effect.* The tempo effect comes from the notion that the static value of a position tends to be an overestimate. Each move strengthens the position of the player moving, and when a child node expands, it becomes weaker from the parent's perspective. Thus, if the principal leaf is expanded, its value tends to decrease, and it is likely that another play-line would become the principal variation.

The simplest implementation of best-first minimax maintains the search tree in memory. When a node is expanded, its children are evaluated and its value is updated. This value is then propagated up the tree until it reaches the root or a node that does not need a value update. The algorithm moves down the tree until it reaches a new principal leaf. This implementation requires exponential memory.

The recursive best-first minimax search algorithm (RBFMS) is an implementation of best-first minimax that does not require exponential memory. RBFMS (also developed by Korf and Chickering [35]) associates with each node on the principal variation a lower bound α and an upper bound β, similar to the branches of alpha-beta pruning. A node remains on the principal variation as long as its backed-up value stays within these bounds. The root is bounded by $-\infty$ and ∞. RBFMS has two symmetric functions: one for MAX and the other for MIN. Pseudo code listings for these functions are available in Figure 4.1. Each takes a node and the two bounds as arguments. These functions perform a best-first minimax search on the sub-tree of a node as long as the backed-up minimax value remains within the α and β bounds. A value that falls outside these bounds is returned as the new value of the node. On a MAX node, the lower bound is the same as that of the parent. The upper bound is the least of the parent's upper bound and the smallest value among its siblings. On a MIN value the upper bound is from the parent and the lower bound is the maximum of the parent's lower bound and the largest value amongst its siblings. If the value of a MAX node's child exceeds its upper bound or the value of a MIN node's child is less than its lower bound, the child value is immediately returned without evaluating the other children.

Algorithm Recursive best-first minimax
Input A game position p from which the search must be conducted, and an evaluation function F
Output A child of p that must be selected as the next move

```
RBFMS(p)
    MAX(p,−∞,∞)
    return a child of p in the principal variation

MAX(n, α, β)
    for each c in child nodes of n
        c.value = F(c)
        if (c.value > β)
            return c.value;
    do
        best = max valued node in siblings of n;
        if n has more than 1 sibling
            near-val = max valued node in (siblings without best)
        else
            near-val = −∞
        best.value = MIN(best,maximum(α,near-val), β);
    while (α <= best.value <= β)
    return best.value;

MIN (n, α, β)
    for each c in child nodes of n
        c.value = F(c)
        if (c.value < α)
            return c.value;
    do
        best = min valued node in siblings of n;
        if (n has more than 1 siblings)
            near-val = min valued node in (siblings without best)
        else
            near-val = ∞
        best.value = MAX(best, α, minimum(β,near-val));
    while (α <= best.value <= β)
    return best.value;
```

Figure 4.1: An outline of the RBFMS algorithm

Syntactically, RBFMS appears similar to alpha-beta, but it behaves differently. The primary difference is that alpha-beta makes decisions based on nodes on the same ply, while RBFMS relies on node values on different levels. Also, alpha-beta does not use the evaluation function to decide which nodes to expand, whereas RBFMS does.

4.4 Shortcomings of best-first minimax

Korf and Chickering explored the performance of RBFMS when it is applied to O . Certain peculiarities of O makes this game more ideal for RBFMS than it might be for other games. In particular, a move in O alters the state of many squares on the board. This influences the game tree in two significant ways: a) The tempo effect is more pronounced, and b) the children of the current position are likely to have significant variations in static values. Also, all the moves in O are conversions - this means any given position cannot re-occur in any of its future play-lines. This property of O makes it less likely that nodes on the tree will have the same static value. The third peculiarity is that the number of moves in an O game is limited to the number of squares on the board and every decision will lead to an end position. At the endgame phase, all future play-lines will reach an end position at approximately the same depth. The final aspect of importance is that draws in O are rare.

These properties can be contrasted with other games such as C and C . (In the discussion that follows, reference is made to C , but the argument also applies to C .) Most moves in C make small alterations to the game state, and the static value of a position tends to change gradually as the game progresses. Although the tempo effect exists, it is far less intrusive on the search of a C game tree. The children of a node in the C tree have more or less the same static value, and if there are variations, groups of moves can be identified, not individual moves. Conversions in C are found in the opening game, but become rarer as the number of kings increases. This means, at some depth in the tree, there are many children that have the same backed-up value because they share a descendant. In C , if no limit is placed on the number of moves, a game can potentially have an infinite number of moves. Searching such a play-line is fruitless. In practice, the number of moves are limited and consequently, many C games end in a draw.

RBFMS does not state what happens when an end position is reached. Korf and Chickering decided to terminate the search when a winning position is found in the O game tree. The rarity of equal values in O implies that the expanded tree will mostly contain a single principal variation, and the variation leading to a winning position is likely to be a good decision. However, this strategy is not effective in C . In the partially expanded C game tree, there is likely to be more than one principal variation. If one leads to a win, search must continue, because another principal variation may also lead to a win, but in less moves. In general (for O and C), it is better to aim for the shorter win, because the decision

is imperfect, and a longer strategy is less likely to materialise. Also, in C , many endgame positions are draws - termination of search when a draw is reached on a principal variation will exclude the possibility that another principal variation may lead to a win.

Shoham and Toledo [61] postulate that the defect with best-first minimax is that it never attempts to raise the scores of the non-best children of a node. Their solution to this problem is to choose amongst child nodes where the probability of choosing is higher for greater valued children than it is for smaller valued children. This approach mitigates the problem of having more than one principal variation, but it weakens the best-first tactic of the algorithm.

The next problem of RBFMS is one of efficiency. RBFMS preserves memory at the cost of losing the value of previous calculations and re-expanding nodes that were explored before. The call stack of RBFMS represents the current principal variation, and when the procedure backtracks, nodes and calculations are lost. This problem becomes more prevalent when there are two (or more) branches with the same values. In this case, one branch can be aborted for another, only to be re-explored when the other branch has reached a weaker position.

It is reasonable to expect that an algorithm that works better for C will also work better for O . A solution to any one of the problems discussed above would not be damaging to the O game strategy. Shoham and Toledo have shown that their algorithm, called the *Parallel Randomized Best-first Minimax Search*, performs better than RBFMS for the O - game. The reason why the Shoham and Toledo algorithm is not chosen for the current research is because it chooses the next move stochastically. This non-deterministic selection is not ideal when the focus is on finding and refining knowledge. When the choice is stochastic, weaker knowledge may win against stronger knowledge, complicating the knowledge evaluation procedure.

4.5 Best-first shallow search

The shortcomings described in the previous section lead to the development of a novel game tree search algorithm called *best-first shallow search* (BFSS). Essentially, BFSS is a specialisation of best-first minimax that allows a knowledge based search on a game tree in which multiple principal variations occur frequently during the search. In addition, the BFSS implementation is a non-recursive procedure that eliminates the need to recalculate node values.

Best-first shallow search selects the next node to expand from a set of nodes called the search frontier. This frontier is the unexplored ‘edge’ of the search tree. The search starts with

the active position, referred to as the *search root*, as the only node in the frontier. The search procedure selects a node in the search frontier to expand, and removes it from the frontier. When the node expands, its children are added to the frontier. If the node is an end position, it will not be replaced, and the frontier will shrink. In general, non terminal nodes are encountered and the frontier grows as the search continues.

Deciding which node in the frontier to expand is a key aspect of BFSS. The idea to select a node on the principal variation is kept; and if only one variation is available the choice will be no different from best-first minimax. If more than one principal variation is available, the node with the smallest ply is selected. The move decision is based on the value of the positions in the first ply. A deeper search tree below one of these positions improves the accuracy of its value assessment. By expanding the principal variation with the smallest ply first, the tree below the least accurate first-ply position is expanded, and its value becomes more accurate. Also, if the evaluation function is unable to distinguish between the nodes close to the root of the search tree, this smallest ply first strategy ensures that BFSS does a breadth-first walk through the game tree until a promising node is identified by the function.

BFSS uses a constant called the *search span* as a termination criterion. This approach was also used by Shoham and Toledo [61]. Except for the search root, the expansion of all nodes in the frontier count towards the termination condition. When the number of expansions reaches the search span, BFSS terminates. It is possible (especially during the end-game) that the game tree is completely traversed before the search span is reached. In this case the search frontier would be empty and BFSS would terminate. The third condition for termination is when a principal variation that leads to a win for the active player is found. Because the smallest ply first strategy is applied when more than one principal variation is available, it is likely that the first winning variation encountered will also be the one with the fewest moves.

Like best-first minimax, BFSS also makes use of the minimax value of a node. However, instead of alternating the minimum and maximum operation, the minimax value of the minimising nodes are negative. These values, called *negamax values* [27], are calculated the same way for all nodes. The negamax value of a node is the negation of the minimum value of the node's children. The value of a terminal node in the search tree is calculated from the evaluation function. The evaluation function values are negated for nodes in a ply from which the passive player makes a move. An outline of the BFSS algorithm is shown in Figure 4.2.

The algorithm consists of two functions, SELECT and BEST, and one procedure EVALU-

Algorithm Best-first shallow search
Input A search root r from which the search must be conducted, an evaluation function F, and a search span s_{max}
Output A child of p that must be selected as the next move

```
SELECT(r, s_max)
    frontier = r
    nodesExpanded = 0
    while (frontier has nodes and nodesExpanded <= s_max and principal is not an end-game)
        bestNode = BEST(frontier, r)
        remove bestNode from frontier
        if (bestNode is not an endgame node)
          append children of bestNode to frontier
          for (each node n in the children of bestNode)
              EVALUATE(n,r)
          bestNode.value = – (max value from bestNode's children)
          parent = best
        do
            parent = parent.parent
            parent.value = – (max value from best.childs)
        while (parent is not r)
        nodeExpanded = nodesExpanded + 1
    return the first node in the current principal variation

EVALUATE(n, r)
    if (node is a negative node with respect to r)
        n.value = - F(n)
    else
        n.value = F(n)

BEST(f, r)
1:   Sort nodes in f in ascending ply depth order
    selected = first node in f
    while (selected is not a principal leaf of r)
        selected = next node in f
    return selected
```

Figure 4.2: Outline of Best-first shallow search

ATE. SELECT returns the node chosen as the next move of the playing agent. BEST returns the next node to be searched in the frontier and EVALUATE calculates and stores the negamax value of a node.

EVALUATE takes two parameters, the node to evaluate, n, and the search root, r. The search root is required to determine whether a node is in a negative ply or not. If the result of subtracting the ply of n from the ply of r is even, the node is in a positive ply. If not, the node is in a negative ply. The evaluation function value is negated for nodes in a negative ply.

BEST also has two parameters, the frontier f and the search root r. In this function, the

search root is used to determine whether a node in f is a principal leaf. A node p is a principal leaf if its absolute value is equal to the absolute value of r, and a path exists from p to r such that the absolute value of all the negamax values in that path are equal. Because the frontier is sorted in descending ply depth, the principal leaf closest to the root will be found first and returned.

SELECT takes as argument the search root r and the search span s_{max}. After initialising the frontier to contain only r, the nodes in the frontier are expanded until the required number of expand operations are reached or one of the other termination criteria is met.

If line 1 in the outline of BFSS is changed to sort the frontier in ascending order, the algorithm would behave exactly like best-first minimax described by Korf and Chickering. This frontier based implementation of best-first minimax will be referred to as best-first deep search, or BFDS.

4.6 Experiment: The search span of alpha-beta

Objective

The aim of this experiment is to determine the search span of alpha-beta. Alpha-beta is a fixed depth search algorithm, and as such the search span of this algorithm cannot be finely tuned. In order to establish a common ground for comparison with selective algorithms, the number of node expansions done by alpha-beta at various ply depths are determined. Only ply depths of 1, 2 and 3 are considered. Deeper ply depths are too expensive to use in learning experiments.

Method

In this experiment C playing agents use a simple but effective evaluation function called F_a. The function is hand-crafted, and is defined as a feature language expression in Equation

4.1.

$$F_a \equiv \left(\begin{array}{l} 30 \times \{O*\}@\{(*,*)\} \\ +\,20 \times \{o*\}@\{(*,*)\} \\ +\,20 \times \neg\{O4, O6\}@\{(*,*)\} \\ +\,10 \times \neg\{o4, o6\}@\{(*,*)\} \\ +\,7 \times \{o*\}@\{(*,7)\} \\ +\,6 \times \{o*\}@\{(*,6)\} \\ +\,5 \times \{o*\}@\{(*,5)\} \\ +\,4 \times \{o*\}@\{(*,4)\} \\ +\,3 \times \{o*\}@\{(*,3)\} \\ +\,2 \times \{o*\}@\{(*,2)\} \\ +\,8 \times \{o*\}@\{(*,1)\} \\ +\,1 \times \{X4, X6\}@\{(*,*)\} \\ +\,1 \times \{x4, x6\}@\{(*,*)\} \end{array} \right) \tag{4.1}$$

The precise meaning of the symbols used in this expression is described in Section 3.6 (a table is available on page 38). The intention of the formula F_a can be described by interpreting each term. The first term asserts that it is good to own kings anywhere on the board. The second term indicates that it is also good to own checkers anywhere on the board. The third and fourth term says that owning a king or a checker that can be jumped is not so good. The next six terms bring across the idea that a checker gains value when it moves towards the 7^{th} rank. The term with a weight of 8 indicates that it is valuable to keep your checkers in the 1^{st} rank. The last two terms imply that it is advantageous to place the opponent's kings and the opponent's checkers under threat.

Using F_a as the evaluation function, a playing agent using alpha-beta at ply 1, ply 2 and ply 3 has been set to play against a playing agent that selects moves at random. If the next move is a win, the playing agent chooses it. If not, the move decision is made according to the agent's selection procedure. At each level, 1000 games were played, each agent taking alternating turns to be the first player. In total, 3000 games were played.

For each game, the number of expand operations requested by alpha-beta was counted. These measurements were aggregated to obtain a sample mean and a sample variance for each ply. Using the aggregates, the population mean can be estimated with a confidence of 95% to fall within the following interval [71]:

$$(\overline{X} - 1.96 \times \frac{\sigma}{\sqrt{1000}}, \overline{X} + 1.96 \times \frac{\sigma}{\sqrt{1000}}) \tag{4.2}$$

where $\overline{X}$ is the observed sample mean, σ is the standard deviation, and n is the number of measurements in the sample.

Results

Table 4.1 shows the results of the measurements. As expected, the number of expansion operations conducted by alpha-beta increases dramatically as the ply depth increases.

Table 4.1: The search span of Alpha-beta at different ply depths

Depth	Mean span (95% conf.)	Variance	Minimum	Maximum
1 ply	7.14355 ± 0.0237925	0.147356	5.9243	8.1421
2 ply	22.32709 ± 0.0501393	0.654403	20.0000	25.0625
3 ply	91.29213 ± 0.3507121	32.01764	75.4468	111.544

4.7 Experiment: Best-first *vs.* Alpha-beta

Objective

The aim of this experiment is to measure how well the best-first search algorithms (BFSS and BFDS) perform when these are set against minimax with alpha-beta pruning.

Method

All the playing agents used in this experiment employ the formula F_a (Equation 4.1). The alpha-beta agent always searches to the third ply. In the previous experiment, the search span of alpha-beta at ply 3 was estimated to be approximately 91, and the minimum span encountered was 75. In this experiment alpha-beta is set against best-first where the search span of best-first increases. The search span ranges from 2 to 70. First BFSS is matched with alpha-beta at each search span interval, then the sequence is repeated for BFDS. In total, 138 matches (or 276 games) where played.

The alpha-beta and best-first playing agents are deterministic; there are no stochastic elements that influence the move decision. The strength of moves selected by a deterministic agent could depend on playing order – some strategies might work better when used by the first player. For this reason, two games are required to determine whether one deterministic agent is better than another. Each agent gets a turn to be the first player in a C game. At the end of each game, match points are awarded: the winner gains 2 match points and 1 match point is allocated

to each player when there is a draw. In total, there are four match points awarded during the match.

The score of an agent is that total of the match points it collected. Only the scores for the best-first agent are recorded. These scores provide some insight into how large the BFSS search span should be to beat alpha-beta with the same knowledge. In addition, it is possible to observe for each search span, whether BFSS or BFDS fares better against alpha-beta.

Results

The recorded scores are shown in Table 4.2. The maximum score an agent can achieve against alpha-beta is 4 (2 points per game). The span associated to each result is the number of nodes the best-first was allowed to expand. For alpha-beta the search span was set to 3 ply. The totals at the end of this table is the sum of all match points, out of a possible score of 276.

Table 4.2: Best-first against alpha-beta at various search spans

Span	BFSS	BFDS	Span	BFSS	BFDS	Span	BFSS	BFDS
2	2	2	**25**	3	3	**48**	3	2
3	1	1	**26**	2	2	**49**	3	2
4	3	2	**27**	3	2	**50**	3	2
5	2	1	**28**	2	2	**51**	3	3
6	1	0	**29**	2	3	**52**	3	4
7	1	1	**30**	2	3	**53**	2	3
8	2	2	**31**	2	2	**54**	3	3
9	2	2	**32**	3	4	**55**	1	2
10	1	3	**33**	3	3	**56**	2	2
11	3	2	**34**	3	3	**57**	3	2
12	3	3	**35**	3	3	**58**	2	3
13	3	3	**36**	3	2	**59**	3	3
14	3	3	**37**	2	2	**60**	4	3
15	2	3	**38**	3	2	**61**	4	4
16	2	2	**39**	2	2	**62**	3	3
17	2	2	**40**	2	2	**63**	4	3
18	2	2	**41**	3	1	**64**	4	4
19	2	2	**42**	3	3	**65**	4	3
20	2	3	**43**	3	3	**66**	4	4
21	2	3	**44**	3	3	**67**	4	4
22	3	4	**45**	3	3	**68**	4	4
23	4	4	**46**	3	4	**69**	4	3
24	3	4	**47**	2	2	**70**	4	3
						Total	**185**	**182**

These results do not show a gradual, or even a consistent increase in performance as the search span increases. This is because the two strategies that are used in this experiment are very specific. For instance, searching BFSS to a span of 10 with the given evaluation function

provides the alpha-beta with an opportunity to win, while a search span of 9 does not. Albeit not consistent, the increase in performance is clear when observing that the frequency of 4-point scores increases as the span increases. The score of BFSS stabilises at 4 when the search span exceeds 62.

It is surprising that alpha-beta does not consistently beat best-first for the spans less than 10, and that it rarely beats (only exception at BFDS, span 41) best-first for spans greater than 10.

The totals for BFSS and BFDS are very close (185 and 183), and as such these totals do not clearly indicate a superior strategy. Table 4.3 shows the average performance of these two strategies in intervals of ten. From these values, it is clear that BFSS performed better in the last interval (61-70).

Table 4.3: Best-first against alpha-beta at search span intervals

Span	BFSS	BFDS
2 to 10	1.6667	1.5556
11 to 20	2.4000	2.5000
21 to 30	2.6000	3.0000
31 to 40	2.6000	2.5000
41 to 50	2.9000	2.5000
51 to 60	2.6000	2.8000
61 to 70	3.9000	3.5000

4.8 Experiment: Deep first *vs.* shallow first

Objective

The aim of this experiment is to determine which method is best to select the next node to expand during best-first search. The two alternatives are: select the shallowest principal variation (BFSS) or select the deepest principal variation (BFDS). Essentially, best-first shallow search is compared with best-first minimax search.

Method

For this experiment the best-first agents play C against a random moving agent. As before the random agent selects the next move at random when no immediate winning move is available. For this experiment a match consists of 10 games. Allocating 2 points for a win and 1 point for a draw leads to a maximum score of 20 points per match.

The primary difference between BFSS and BFDS is that BFSS searches the lowest ply first when more than one principal variation is available. Multiple variations occur when an evaluation function is unable to distinguish clearly between different positions. Thus, when comparing the two best-first searches, the evaluation function is likely to influence the outcome of the comparison. For this reason, two evaluation functions are used: the first is the function F_a from page 59 and the second function, F_b, is adapted from F_a:

$$F_b \equiv \left(\begin{array}{l} 1 \times \{O*\}@\{(*,*)\} \\ +1 \times \{o*\}@\{(*,*)\} \\ +1 \times \neg\{O4, O6\}@\{(*,*)\} \\ +1 \times \neg\{o4, o6\}@\{(*,*)\} \\ +1 \times \{o*\}@\{(*,7)\} \\ +1 \times \{o*\}@\{(*,6)\} \\ +1 \times \{o*\}@\{(*,5)\} \\ +1 \times \{o*\}@\{(*,4)\} \\ +1 \times \{o*\}@\{(*,3)\} \\ +1 \times \{o*\}@\{(*,2)\} \\ +1 \times \{o*\}@\{(*,1)\} \\ +1 \times \{X4, X6\}@\{(*,*)\} \\ +1 \times \{x4, x6\}@\{(*,*)\} \end{array} \right) \tag{4.3}$$

F_b is F_a with all the feature weights set to 1. Making all the weights equal produces a function that is more likely to find multiple principal variations during search, because this function would assign the same value to many positions. Also, because the weights of F_a was tuned (by hand), setting all terms to a weight of 1 will produce a weaker evaluation function.

This experiment was conducted for search spans ranging from 2 to 25. For each search span, 100 matches (1000 games) were played where the best-first agent competes with the random agent. The mean match points for 100 matches can be determined at a 95% confidence to be:

$$(\overline{X} - 1.96 \times \frac{\sigma}{\sqrt{100}}, \overline{X} + 1.96 \times \frac{\sigma}{\sqrt{100}}) \tag{4.4}$$

where $\overline{X}$ is the mean of the observed match points, σ is the standard deviation, and n is the number of measurements in the sample.

Measurements were taken for BFSS and BFDS for the different search spans. One set of measurements was collected for F_a and another for F_b.

Results

The match points with 95% confidence intervals achieved by the different configurations are listed in Table 4.4. As expected the F_a evaluation function clearly outperforms F_b. An increase in search span coincides with a decrease in variance, showing that the outcome becomes more predicable as the best-first agent searches further.

Table 4.4: The performance of BFSS and BFDS against random

Span	BFSS F_a	BFDS F_a	BFSS F_b	BFDS F_b
2	18.5700 ± 0.2617	18.0714 ± 0.4312	17.4800 ± 0.3337	17.4300 ± 0.3516
3	18.4500 ± 0.2544	18.4400 ± 0.2450	17.0200 ± 0.3225	16.5800 ± 0.3981
4	18.7200 ± 0.2282	18.4900 ± 0.2322	16.9400 ± 0.3294	16.7000 ± 0.3616
5	18.2900 ± 0.2199	18.5600 ± 0.2497	17.3400 ± 0.3802	17.1300 ± 0.3397
6	18.9400 ± 0.2005	18.9700 ± 0.1815	17.2600 ± 0.3047	17.2400 ± 0.3015
7	19.0800 ± 0.1777	19.2600 ± 0.1819	17.5900 ± 0.3344	17.3800 ± 0.2945
8	19.5000 ± 0.1461	19.2500 ± 0.1535	17.9800 ± 0.2868	17.1500 ± 0.2979
9	19.4400 ± 0.1256	19.3600 ± 0.1613	18.0200 ± 0.2974	17.3700 ± 0.2739
10	19.7200 ± 0.1045	19.6600 ± 0.1221	18.2100 ± 0.2558	17.6000 ± 0.3139
11	19.5900 ± 0.1185	19.5600 ± 0.1345	18.0700 ± 0.2449	17.5000 ± 0.2765
12	19.5300 ± 0.1406	19.5500 ± 0.1347	17.9500 ± 0.2777	17.5700 ± 0.2761
13	19.6100 ± 0.1273	19.4300 ± 0.1607	18.0500 ± 0.2678	18.0400 ± 0.2743
14	19.6400 ± 0.1097	19.6700 ± 0.1216	18.2700 ± 0.2328	17.6800 ± 0.2867
15	19.6700 ± 0.1184	19.7000 ± 0.1165	18.5000 ± 0.2272	18.0700 ± 0.2647
16	19.6400 ± 0.1166	19.7600 ± 0.0929	18.3300 ± 0.2461	17.7600 ± 0.2976
17	19.5700 ± 0.1284	19.6100 ± 0.1178	18.2800 ± 0.2446	17.9200 ± 0.2609
18	19.7100 ± 0.1090	19.6700 ± 0.1007	18.7300 ± 0.2044	18.2300 ± 0.2489
19	19.7600 ± 0.1047	19.6300 ± 0.1296	18.7600 ± 0.2178	18.3200 ± 0.2261
20	19.7000 ± 0.1097	19.6900 ± 0.1068	18.7300 ± 0.2226	18.2700 ± 0.2442
21	19.7700 ± 0.09592	19.7700 ± 0.09178	18.9900 ± 0.1816	18.1200 ± 0.2858
22	19.7300 ± 0.1074	19.7500 ± 0.1019	19.0200 ± 0.2047	18.3700 ± 0.2358
23	19.7700 ± 0.09178	19.6900 ± 0.1205	18.7800 ± 0.1901	18.2778 ± 0.2837
24	19.8000 ± 0.0924	19.5600 ± 0.1429	18.8300 ± 0.2029	18.1300 ± 0.2407
25	19.8300 ± 0.07907	19.7100 ± 0.1016	18.9700 ± 0.1899	17.9900 ± 0.2591

Although BFSS performs better for both functions, the results for F_b is more conclusive. As expected, the weaker function performed better with a greater margin than the stronger function. The line diagram in Figure 4.3 highlights this observation.

4.9 Conclusion

Minimax search with alpha-beta pruning is a popular method used by game playing agents to search through the game tree. This is a fixed-depth method, and as such it has two problems. The first is that it is not possible to fine tune the amount of time used for the search, and the second is that it does not use knowledge during the search process.

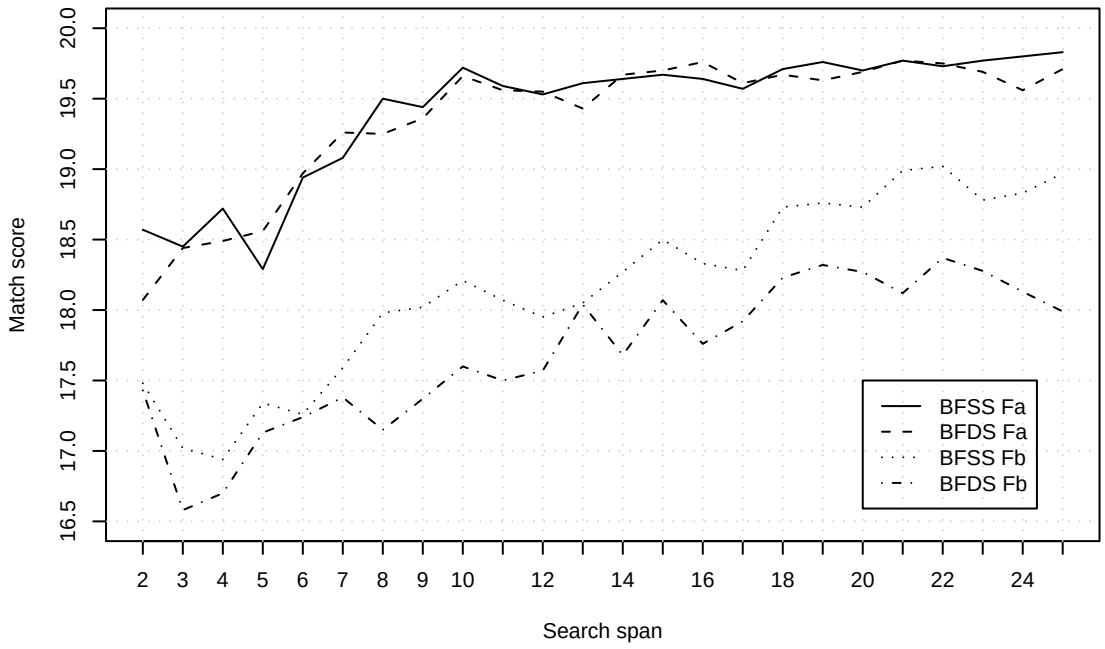


Figure 4.3: The performance of BFSS and BFDS against random

Best-first minimax is a selective search technique developed by Korf and Chickering that uses the knowledge encapsulated in the evaluation function to decide which branch in the game tree to explore further. As a selective algorithm, the time it uses can be limited easily, and when the search is terminated the most promising move found thus far can be selected by the playing agent. A problem with best-first minimax is that it is more suited for games where the tempo effect is more pronounced, and the evaluation function is able to separate good positions from bad positions very accurately.

Best-first shallow search is a new game tree search algorithm proposed in this chapter that is able to work with weaker knowledge and possibly with more game types. This new algorithm expands the principal variation with a leaf that is closest to the root of search tree first. In this way it handles game trees with duplicate positions better and it is more likely to find the shortest winning play-line during the end-game phase.

In the second experiment alpha-beta competed against the selective algorithms. The results show that it is better to employ knowledge during search. The last experiment compares best-first minimax with best-first shallow search. For weaker knowledge it is clear that best-first shallow search is better, but with stronger knowledge neither algorithm dominates.

In the next chapter, the spotlight moves away from the playing agent and it provides a review of the machine learning techniques used to optimise the weights of the evaluation function.

Chapter 5

Learning the evaluation function

The previous two chapters described the facets of the playing agent. In this chapter, the focus returns to the learning agent. A review of the predominant learning methods is provided, and the learning framework is introduced. This framework represents a key contribution of the current work, and the details of the framework is presented in the two chapters that follow this one.

5.1 Introduction

The playing performance of an agent is improved by increasing the search capabilities, or by strengthening the knowledge of the agent (see Section 2.4). This chapter provides a review of the methods used to improve the game knowledge of the playing agent. This learning task is the responsibility of the learning agent.

As a starting point for learning, the zero-knowledge agent defined on page 32 prescribes that learner should not be equipped with any game knowledge other than the knowledge contained in the game rules. In order to argue that learning has occurred, Minsky [43] suggests as the minimum criterion that such an agent learns to play a better-than-chance game.

The definition of the zero-knowledge agent does not place any constraints on the learning environment. Clearly, the environment in which the agent learns would have a marked influence on its learning performance. In this environment, the agent could be provided with examples obtained from related literature, it could be guided by the interactive feedback of an expert, or it could learn on an on-line game server. The latter training environment was used to train Baxter's C program, Knightcap [5] and in Fogel's Blondie [22]. It learned by playing on an Internet chess server against human players. On this server, human players tend to compete

against opponents of their own strength. Consequently, as Knightcap's strength increased, so did the strength of its opponents.

The variety of environments and different objectives of the researchers (as highlighted in Section 2.3) lead to a large number of approaches to game learning. The collection of techniques described in this chapter is not the result of a complete literature survey. Methods considered to be generally important and those that are directly related to the techniques developed for the current study are described.

The knowledge representation method described in Chapter 3 encapsulates knowledge in a linear evaluation function. Section 5.2 discusses the problems introduced by this linearity and the use of game phases as a remedy to these problems. In Section 5.3, the challenge of feature discovery is discussed and the approach used by GLEM to identify features is reviewed. This is followed by Section 5.4 that describes the application of machine learning techniques to the training of the evaluation function's weights.

An overview of the learning framework is provided by Section 5.5. Specific attention is given to the macro-cycle of the framework. Section 5.6 describes a method that is used in subsequent chapters to compare the performance of playing agents. Section 5.7 concludes with a summary of this chapter.

5.2 The phased evaluation function

A reason for choosing non-linear representations such as neural networks for game learning is that these techniques are able to approximate a much larger class of evaluation functions than linear representations. Berliner [6] describes a problem with linear functions, called *suicide construction.* Consider a simple term in a linear function, $S = I \times D$, where S denotes suffering, I pain intensity and D denotes pain duration. If the aim is to minimise suffering, this term seems like reasonable advice: reduce I and D. If the program is able to manipulate the value of D, and an excruciating pain cannot be removed, it may well recommend suicide by driving D to 0. This advice would usually not be a valid solution. However, when this term appears amongst others in a linear evaluation function in which other terms place a high value on staying alive, the value of $D = 0$ might be taken. In other words, suicide construction occurs when the potential exists for one of the weights to be adjusted in an undesirable direction.

Given a linear representation for the evaluation function, non-linearity can be introduced by separating the game into phases, and assigning a different evaluation function to each phase.

The result is a composite function called a *phased evaluation function.* Clearly, the possibility remains to use neural networks instead of linear functions as the parts in this composition. Boyan [7] explored this possibility using B as domain. His results indicate that using a different network for each game phase is an improvement over learning the weights of a single neural network for the entire game.

There are many ways to define the phases for a given game. Boyan identified 12 game phases for B , and trained 12 networks. He determined the game phase from the pip count* for each player. The player's pip count for contact positions† is classified as small, average or large. Pairing the pip count classification of the players gives rise to 9 game phases: {(small,small),(small,average), ..., (large,large)}. The remaining 3 phases categorises the non-contact positions. A non contact position can be in one of 3 states: (a) the first player leads, (b) the second player leads, or (c) the race is relatively close.

The phases used by Lee and Mahajan [38] for O is much simpler. They use the total number of discs on the board to determine the game phase. Their learner focussed on the middle game and 26 (out of 64) phases were used during training: from the 24^{th} phase to the 49^{th} phase. The training problem was to find the weights for four hand-coded features. The results obtained by Lee and Mahajan show an improvement in the performance of an O program that already played at world championship level.

5.3 Discovering features

Nearly five decades before this writing, the challenge to automatically discover game features was set by Arthur Samuel [53]. Samuel's challenge implies the discovery of features that can be used in place of the hand-crafted features used by his linear evaluation function. However, the non-linear, neural network approach proved to be an effective knowledge representation technique that is ideally suited for traning.

As an evaluation function, a neural network encapsulates knowledge in three ways: the network topology, the weights and the encoding. Encoding refers to the process that translates the game position into the floating-point values that are assigned to the input nodes of the network. The encoding is usually a fixed procedure, and conceptually node weights correspond with the

*The Pip Count is the total number of points that a player must move his pieces to bring them home and bear them off. At the start of a game each player has a pip count of 167.

†A contact position is one in which one or more pieces of the players are (or can potentially be) on the same point position.

weights in a linear function. It is possible to 'discover' an optimal network topology. Consequently, one may expect the problem of identifying the topology of a neural network and the problem of finding features for a linear evaluation function to be related. However, the two problems are very distinct, and little synergy can found in the methods used. The main difference is that linear feature discovery expects to find highly structured information while an optimal neural network architecture represents an optimal structure for the non-linear function.

The problem of finding a neural network architecture revolves around the structure of the hidden layers in the network. One approach is to have a single hidden layer and to determine the number of nodes in this layer. This approach was used by Fogel for an agent that learns how to play T - - [21]. Moriarty and Miikulainen [45] takes on the more complicated problem of also determining the number of hidden layers in the architecture. Their program uses O as learning domain.

The discovery of features for a linear evaluation function is closer to the aim of the current work. In the subsection that follows, an approach developed by Michael Buro to learn such features is described in detail. He also used O for his learning experiments.

5.3.1 GLEM configuration discovery

Micheal Buro introduced the Generalised Linear Evaluation Model (GLEM) to discover the terms of a linear evaluation function [10]. The GLEM representation scheme and GLEM configurations have already been described in Section 3.4. A GLEM configuration is a combination of GLEM atomic features, and can (in a more general sense) be regarded as a feature in a linear evaluation function. Consequently, the approach used by GLEM to identify a set of configurations is essentially an approach to discover evaluation function features.

The number of possible configurations for a set of GLEM atomic features $\{f_1, f_2, \ldots, f_k\}$ grows exponentially with the size of the set. If all the atomic features are binary, there are 2^k possible configurations. However, the possibilities are typically much more than 2^k because a GLEM atomic feature is likely to have more than two possible values, and a valid configuration does not need to contain all the atomic features. Hence, it is impractical to enumerate all the possible configurations in an evaluation function and then to find optimal weights for them. Also, most of these combinations that will not be useful in an evaluation function. There are certain combinations describe positions that are impossible to attain: for example, having all 8 your pawns in the 7^{th} rank during a C game.

Buro's deduction process [10] decides on a set of *active configurations* based on a large set of example positions, E, and a cut-off value n. The algorithm includes a configuration in the active set if and only if there are n or more instances in E that match the configuration. A higher value of n reduces the risk of overfitting. Figure 5.1 outlines the most basic implementation of this algorithm.

Algorithm GLEM Discovery
Input A set of examples E, a set of features F and a cut-off value n.
Output A set of configurations C

```
    GlemDiscover(E,F,n)
1:      Set R to the most general values in F
        C = R
        N = R
        while (|N| > 0)
            M = {}
            for each c_j in N
                for each r_k in R
                    e = c_j ∪ r_k
                    if (e is found no less than n times in E)
                        M = M ∪ {e}
            N = M
            C = C ∪ N
2:      Remove all general configurations from C
        return C
```

Figure 5.1: An outline of the GLEM discovery algorithm

In line 1 of the outline, the set C is initialised to the set of most general elements, R. An element in this set is a feature combined with an element from the feature's domain. For example, an atomic feature called *color* with domain {*red,green,blue*} would contribute 3 most general elements to the set. R is also used in the inner loop of the algorithm. Here, every configuration in N (initially set to R) is combined with an element in R and if the new combination is supported by more than n examples, it eventually becomes an element in C. This process is repeated with all the new elements identified in the `while` loop until no new configurations are found.

A combination a is more specific than a combination b if an only if a contains all the elements in b and a has more elements than b. For every combination in C, created during the collection cycle that has a length greater than one, the same combination with the last atomic feature removed is also in C. Thus, most of the combinations collected are generalisations of other combinations that are also collected. The final step of the procedure (line 2) returns C after all the general configurations are removed from this set.

Buro's contribution [10] includes novel techniques to improve the computational efficiency of the algorithm itself and also the efficiency of the access to, and the storage of the resulting data structure.

Essentially GLEM expressions are two levels deep - the first level contains the configuration, and the second contains the atomic features. The feature language, $\mathcal{F}$, introduced in Section 3.6 is a much richer representation method, that allows complex expressions of arbitrary length. In addition, the atomic features used in $\mathcal{F}$ are less coarse than GLEM's atomic features. These two differences make the number of possible expressions in $\mathcal{F}$ much more that the possibilities in GLEM. It is therefore not practical to use GLEM to find $\mathcal{F}$ expressions from a set of example positions.

5.4 Learning weights

The automatic optimisation of the weights of an evaluation function is the most extensively studied learning problem in game playing [27]. Like the feature discovery problem, the problem of finding optimal weights for game playing agents has also been more visible in the neural network arena. In contrast to the feature discovery problem, the problem of finding optimal weights for a linear function and for a neural network that plays a game are very alike. In both problems, the aim is to find a solution in a multi-dimensional space of floating-point values. In addition, the problems share the same input (albeit encoded differently), and learns the same activity. A variety of different approaches to solve this problem has been proposed.

Samuel [53] provides his C learning agent with a linear evaluation function composed from hand-crafted terms based on C literature. The sign of the weights was also part of the input. Samuel remarks that the program could already play a 'fairly interesting game', even before learning started.

Pollack *et. al.* [48] trains a fixed neural network to play B . The network's input nodes encode only visible features and a flag that indicates whether the game is in the endgame phase.

Kotnik and Kalita [36] trains the weights of a fixed neural network to play G R . In this case, the network does not encode visible features of the game state. The input nodes accept an encoding of the accessibility of the playing cards. More accessible cards have a higher value, *i.e.* a card that can potentially be accessed by the player has more value than a card known to be in an opponent's hand.

Franken and Engelbrecht [25] developed a C learning agent that finds optimal weights for a neural network. As input, this network takes a value for each square, such that a higher value indicates the contribution the square makes to the strength of the active player. For instance, a square occupied by an opponent king has less value than an empty square or one that is occupied by the active player's king.

Another recent example is the neural networks trained by Fogel *et. al.* [23] for a C player. In addition to the trained neural networks, the player was also equipped with an advanced tree search that extends when a position at the edge of the search meets certain criteria, as well as opening- and closing books. The research problem addressed here is to improve the skill of a very strong player. In this case the input nodes accepted the material value of the piece on the square of the C board.

Although the specific learning problems and the representations are diverse, a relatively small number of different learning methods are employed by most of the researchers. These methods are general approaches to the class of learning problems associated with weight training for games. In the subsections that follow, supervised- and reinforcement learning are described as fundamental learning methods. Temporal difference learning is a method that has been devised specifically for game learning. In the last subsection, strategies to use coevolution as a game learning method are explored.

5.4.1 Supervised learning

The *supervised learning* process typically uses a large collection of rated example game positions as input. The rating of an example is an expert's estimate of the value the evaluation function should assign to the position [27]. Successful training produces weights for the evaluation function such that the function correctly predicts the value of new positions encountered during play. Essentially, the evaluation function generalises the specific information supplied in the form of rated examples.

The challenge of supervised learning is to acquire accurate ratings for examples. The value assigned by an expert to a particular example is a subjective value; and there is no absolute standard to measure the correctness of his assessment. Incorrectly rated examples complicate the generalisation process and lead to ineffective training. Consistency is more important than absolute accuracy. As long as the ratings provide the learner with enough information to decide the direction in which to adjust the weights, the examples are adequate.

The success of the achievements of Tesauro has to be partly due to the knowledge he has of B . As an expert player, Tesauro himself provided the ratings for the initial work he conducted on this game [27, 69]. However, for the typical research endeavour, experts are not readily available and the rating process is a time-intensive and costly ordeal [27].

A common method used for supervised training adjusts the weights such that the mean squared error is minimised. This error measure is calculated as follows for a set of weights, $\vec{x}$:

$$E(\vec{x}) = \frac{\sum_{i=1}^{N}(F(\vec{x}, e_i) - R(e_i))^2}{N}$$

where $F(\vec{x}, e_i)$ is the evaluation function value for example e_i, and $R(e_i)$ is the rating for example e_i. This equation was used by Buro [9, 10] and Mitchell [44] to train O players.

Comparison training is an approach introduced by Tesauro [68] that does not use quantitative ratings. Instead of rating each individual example separately, the expert is given a set of examples, and he decides which of the examples in the set is the best, and which example is the worst. The training objective for comparison training is to maximise the number of correct choices made by the evaluation function when it is presented with the same example sets.

If the research aim is to train a playing agent that is able to beat top human players, supervised learning is not a practical option. Even if enough correctly rated (or compared) examples are available, the learner still generalises over the examples. As a generalisation, the learner may achieve very high levels of playing performance, but it will not discover a novel strategy that will surprise an expert player. In effect, such a learner will be unable to beat the expert(s) that provided the ratings for the examples from which it obtained the generalisation.

5.4.2 Reinforcement learning

Reinforcement learning [65] is a learning process whereby an agent learns which action to take from feedback information it receives from the environment. This information, called *reward*, is received after the learner takes one or more actions. If the reward for an action is positive, the action is reinforced, and the tendency of the learner to take that action again in the future is increased. A negative reward decreases the tendency to repeat the action that was taken.

The *learning rate*, θ, is a positive floating-point parameter to the reinforcement learning process ($\theta < 1$). This parameter controls the rate at which the agent adjusts its behaviour based on each reward. A high value for θ increases the learning rate and the agent adapts quicker when a reward is received. A low value slows down the learning progress, but it is more reliable in

the uncertainty caused by the noise that is typically present in the feedback process.

As an example, consider an agent that must choose between two actions: turn left or turn right. The learning exercise is to teach the agent that it must always turn right. Internally, the agent keeps a probability value by which it chooses to turn right. The initial value for this probability is 0.5. Let the sequence $p_1, p_2, p_3, \ldots$ represent this internal probability value as it changes over time (p_t is the probability before the t^{th} action is taken). At time i, the agent turns right and it receives a positive reward. In this case p_i is adjusted as follows:

$$p_i = p_{i-1} \times (1 - \theta) + \theta$$

Conversely, if the agent turns left it receives a negative reward and p_i is adjusted as follows:

$$p_i = p_{i-1} \times (1 - \theta) - \theta$$

This example clearly demonstrates the role of the learning rate. A low rate elevates the importance of the experience gained thus far, whereas a high value for θ considers the last feedback as more important. After a number of attempts, the probability by which the agent chooses to turn right will increase and eventually the agent will always turn right.

The term *delayed reinforcement* is used to describe a reinforcement learning process where more than one action is taken before a reward is received. The delayed feedback is typical of the games of interest to the current study; only after an entire sequence of moves are made, the outcome of the game becomes available. The actions available to the game learning agent are presented as the list of available moves at each decision point in the game. Typically, the actions are reinforced or penalised by increasing or decreasing a score associated to the position on the play-line. The score adjustment for a draw is somewhere between the positive adjustment for a winning position and the negative adjustment for a losing position.

It is possible that a lost game is caused by one bad move, and the other moves in that game are good moves. In such a case, only the culprit move should receive a negative reward. Unfortunately, the learning agent is usually unable to distinguish the bad moves from the good moves in the play-line. This leads to the *credit assignment problem* [43]. This problem concerns the method by which the reward is distributed to the individual actions responsible for it. Two simple approaches are practical [41]: the first is to distribute the reward equally amongst all the moves that contributed, and the second option is to increase the proportional assignment

for positions that occur later in the game. This proportional assignment is based on the premise that initial moves have a lesser impact on the outcome of the game than moves in the endgame. Clearly, these approaches do not solve the assignment problem for a specific game line. However, after applying a consistent approach over a multitude of training games, the score of a position becomes more accurate.

During the initial training of reinforcement learning it is possible that some properties are mistakenly chosen to be important. The problem of "unlearning" this knowledge can hinder learning progress at a later stage [43]. This problem can be mitigated by choosing the training sequences well.

The basic process of reinforcement learning is problematic for non-trivial games. In these games, the likelihood of coming across the same middle game position is very low. During play many positions are encountered that have not yet been scored. For these games some kind of generalisation over positions is required. One method is to assign the reward to a class of positions, and not to an instance. A related problem is that training on the outcome of games is a slow process because many games are needed before the scores converge to accurate values [9].

5.4.3 Temporal Difference learning

Temporal difference learning (TD) is a reinforcement learning process that mitigates the problems associated with delayed reinforcement. When the reward is immediate, the behaviour of TD is no different than the reinforcement learning process described in the previous section. However, when a *multi-step activity* is learnt, TD converges faster and produces better predictions than the basic reinforcement approach [64].

The key to understanding TD is to consider the sequence of observations, $\vec{x}_1, \vec{x}_2, \vec{x}_3, \ldots, \vec{x}_m$ that leads to the outcome z. Each vector $\vec{x}_t$ describes the state of the environment at time t; and z is the outcome at time m. The aim of the TD learner is to produce a corresponding sequence of predictions of the outcome (z), say $P_1, P_2, P_3, \ldots, P_m$. At time t, the prediction for z is P_t. If these predictions are accurate, the player is able to determine at time t which move would lead to a winning outcome because the prediction, P_{t+1} for the resulting position, $\vec{x}_{t+1}$ will be a win.

Each prediction is based on a vector of modifiable weights, $\vec{w}$. Thus, P_t can be denoted as $P(\vec{x}_t, \vec{w})$. During learning, the values of $\vec{w}$ is updated.

Consider an approach to determine the value of $\vec{w}$ after the sequence (of m steps) is complete

and the outcome is known to be z. The supervised learning process would use the position-outcome pairs $(\vec{x}_1, z), (\vec{x}_2, z), (\vec{x}_3, z), \ldots, (\vec{x}_m, z)$ as rated examples. An assignment can be made to $\vec{w}$ that satisfies the equation $P(\vec{x}_m, \vec{w}) = z$. Now, for each observation leading to $\vec{x}_m$ an increment as $\Delta\vec{w}$, is determined and $\vec{w}$ is the sum of these increments:

$$\vec{w} = \sum_{t=1}^{m} \Delta\vec{w}_t \tag{5.1}$$

The increment, $\Delta\vec{w}_t$, depends on the difference between P_t and z, and how a change in $\vec{w}$ will affect the value of P_t. For supervised learning a learning rate, θ and a gradient, $\nabla_{\vec{w}} P_t$ can be used to determine the increment:

$$\Delta\vec{w}_t = \theta(z - P_t)\nabla_{\vec{w}} P_t \tag{5.2}$$

The gradient, $\nabla_{\vec{w}} P_t$ is the vector of partial derivatives of P_t with respect to each component of $\vec{w}$. For the special case where a linear function, P_t is learnt:

$$P_t = \vec{w}^T \vec{x}_t = \sum_i \vec{w}(i)\vec{x}_t(i) \tag{5.3}$$

where $\vec{w}(i)$ and $\vec{x}_t(i)$ are the i-th components of $\vec{w}$ and $\vec{x}_t$ respectively. In this case,$\nabla_{\vec{w}} P_t = \vec{x}_t$ (it is the derivative of Equation 5.3 with respect to $\vec{w}$), and Equation 5.2 is reduced to the Widrow-Hoff rule [75]:

$$\Delta\vec{w}_t = \theta(z - \vec{w}^T \vec{x}_t)\vec{x}_t \tag{5.4}$$

The difference $z - \vec{w}^T \vec{x}_t$ represents the scalar error between the prediction and what it should have been. This error is multiplied by the vector $\vec{x}_t$ because $\vec{x}_t$ indicates how each weight must be adjusted to reduce the error. For example, if the error is positive, $\vec{x}_t[i]$[‡] is positive and $\vec{w}[i]$ will be increased - resulting in an increase of $\vec{w}^T \vec{x}_t$ and reducing the error.

The equation to determine P_t is dependant on z, and P_t cannot be computed before z is known. In other words, Equation 5.4 cannot be calculated incrementally. There is a temporal difference learning procedure, called TD(1) that produces the same result as Equation 5.4 such that the value can be computed incrementally. Let $P_{m+1} \equiv z$, then the error at P_t is the sum of

[‡] $\vec{d}[b]$ refers to the b^{th} element of $\vec{d}$

the changes in the predictions after P_t:

$$z - P_t = \sum_{k=t}^{m} (P_{k+1} - P_k)$$

From Equation 5.1, the linear weights are determined as follows:

$$\begin{aligned} \vec{w} &= \vec{w} + \sum_{t=1}^{m} \theta(z - P_t)\vec{x}_t \\ &= \vec{w} + \theta \sum_{t=1}^{m} \sum_{k=t}^{m} (P_{k+1} - P_k)\vec{x}_t \end{aligned}$$

Because $\sum_{i=1}^{n} \sum_{j=i}^{n} a_i b_j = \sum_{j=1}^{n} \sum_{i=1}^{j} a_i b_j$:

$$\begin{aligned} \vec{w} &= \vec{w} + \theta \sum_{k=1}^{m} \sum_{t=1}^{k} (P_{k+1} - P_k)\vec{x}_t \\ &= \vec{w} + \sum_{t=1}^{m} \theta(P_{t+1} - P_t) \sum_{k=1}^{t} \vec{x}_k \end{aligned}$$

The increment can now be expressed as:

$$\Delta\vec{w}_t = \theta(P_{t+1} - P_t) \sum_{k=1}^{t} \vec{x}_k \tag{5.5}$$

The value of $\Delta\vec{w}_t$ can be computed incrementally because it depends on the value of the two successive predictions and the sum of all preceding increments. In response to a new prediction, P_{t+1}, $\Delta\vec{w}_t$ can be determined and used to update the predictions, $P_1, P_2, \ldots, P_t$. The TD(1) procedure alters all these to an equal extent. TD(λ) is a class of procedures that make greater alterations to more recent predictions: the prediction for an observation k steps in the past are weighted by λ^k where $\lambda \in [0, 1]$:

$$\Delta\vec{w}_t = \theta(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \vec{x}_k \tag{5.6}$$

A larger value for λ increases the number of preceding observations that has an influence on the value of $\Delta\vec{w}_t$. For TD(1), all the preceding observations are considered, and for TD(0) none of the preceding observations have an influence on the weight increment.

The advantage of using the exponential form to alter the weights is that it can be computed

incrementally: that is, the value $\Delta\vec{w}_{t+1}$ can be computed using the values obtained from the computation of $\Delta\vec{w}_t$. The way this is done is shown in the following derivation:

$$\begin{aligned}\Delta\vec{w}_{t+1} &= \theta(P_{t+2} - P_{t-1}) \sum_{k=1}^{t+1} \lambda^{t+1-k} \vec{x}_k \\ &= \theta(P_{t+2} - P_{t-1})(\vec{x}_{t+1} + \underbrace{\sum_{k=1}^{t} \lambda^{t-k} \vec{x}_k}_{\text{from Equation 5.6}})\end{aligned}$$

In order to gain an intuitive understanding of how TD learning works, consider a particular game position q. From experience, the probability of winning from q is estimated at 0.10. In a particular game line that leads to a win, another position, p, precedes position q. What is the estimated probability of winning associated to p? Assuming p was encountered only once during training and it lead to a win, a supervised learning method would conclude that p is a good move. The TD method would use q's estimate to determine p's value because q precedes p. The low estimate of 0.10 for q would lead TD to the conclusion that p is more likely to lead to a loss. If q is correctly rated, the TD method is more correct. Given infinite experience, both methods would converge to the same value for p, but when the experience is limited, TD learns a better evaluation.

Samuel's [53] C learner also adjusts the weights in accordance with the immediate successors in a game line. The difference is that Samuel's program does not utilise an *a priori* rating of the positions, while TD requires that every training sequence ends with a definite, externally supplied outcome. Samuel's decision to fix the weight of the most prominent feature (piece advantage), prevents his training procedure from finding useless evaluation functions.

Tesauro [66, 67] presented the first results of temporal difference learning on the training of an evaluation function for B . The trained playing agent, named TD-Gammon, achieved expert level play. Other research attempts failed to achieve this level of performance, and TD's success has been attributed to the characteristics peculiar to B that makes it an ideal game for learning from self-play [27, 48, 36]. The primary characteristics cited is that the game does not require a large amount of search and that the dice-rolls guarantee the exploration of a sufficient variety of positions to identify all regions of the feature space.

5.4.4 Coevolution

In the context of game learning coevolution is a learning process that involves adaptation of the environment in which the learner finds itself. As the environment changes, the learner must adapt. Coevolution also requires that the changes in the environment are largely a consequence of the changes in the learning agents.

The simplest example of coevolution is two agents that learn by competing against each other. In this case, one agent is the learning environment of the other. After each competition, the evaluation function of at least one of the agents are adjusted. This approach has been used by Pollack *et. el.* [48] and Kotnik [36] to train B and G R players, respectively. They compared their results with the TD approach and concluded coevolution works better than TD.

This simple coevolution approach works well for B and G R because these games are stochastic. The stochastic element ensures that a variety of match conditions are tested, even if the strategies of the competitors remain unchanged. Thus, the accuracy of the assessment of stochastic competitors can be improved by increasing the number of matches played during a competition.

For perfect information games, this simple strategy would not be effective. In perfect information games, the outcome of the game is always the same when two players compete. A win against a single opponent does not imply the winning player's strategy is better in general. For this reason, coevolution learning for perfect information games typically involves a population of learning agents.

Two approaches toward the coevolution of a population are popular. The first is the use of a *genetic algorithm* (GA) in which the population evolves using natural selection and mutation. The second approach is called *particle swarm optimisation* (PSO), where the population is modelled as a swarm that finds solutions by exploring the multi-dimensional problem space. These approaches have a common challenge: the separation of the better individuals in the population from those individuals that do not perform well. The strategy to rate the individuals in the swarm is encapsulated in a function called the *fitness function.*

The use of a GA or a PSO does not always imply coevolution. Coevolution is not present when the environment used to rate individuals is entirely static. For instance, the fitness function of a GA used by Tunstall-Pedoe [70] uses a set of example moves. The fitness of an individual depends on the number of moves chosen correctly when presented with these examples. The

training environment consists of a static example set, and thus Tunstall-Pedoe does not employ coevolution. Another example is the O learning agent of Moriarty and Miikulainen [45]. It initially plays against random moving players, and later against an advanced alpha-beta agent to determine the fitness of an individual.

A simple mechanism to introduce coevolution into the fitness function is to measure the fitness of an individual using the other individuals in the population. In game learning this mechanism is readily available due to the competitive nature of the activity that is learnt. This approach has been used to train a T - - player by Fogel [21], Franken and Engelbrecht [26] and Messerschmidt and Engelbrecht [40]. The same approach was also applied to the more difficult games, C [25, 11, 22] and C [23]. In these endeavours, the training involved the finding of optimal weights for neural networks. Davis and Kendall [14] used coevolution to find weights for a hand-crafted linear evaluation function that is used to play a perfect-information game called A .

In the GA approach, the population evolves in cycles called generations. The fitness function determines the probability that an individual will become a parent of the new generation. A new generation of individuals are built from the previous generation using the information of their parents. A *cross-over* operator determines how the information from the parents are combined to form the individual for the new generation. At a given probability, called the *mutation rate*, a *mutation* operator is applied to the offspring. The purpose of this operator is to introduce new material into the population.

Fogel's T - - learner [21] avoids the cross-over operator by using a single parent. He uses a mutation operator that makes random adjustments to the weights and adjusts the number of nodes in the hidden layer of the neural network. The C learner of Fogel *et. al.* [23] also used a single parent and weight adjustment for mutation. Davis and Kendall [14] used the same approach for their GA.

The PSO algorithm uses the fitness function to identify individuals in the population to which other individuals are attracted. Using the same fitness function, PSO has been shown to be an improvement on GA [25, 26, 40]. For this reason, PSO is selected as the method to be used by the current research. The detail of the PSO algorithm and how it is used in the game learning domain is discussed in a later chapter.

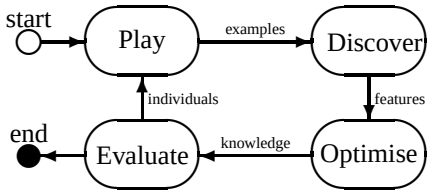


Figure 5.2: The learning framework process

5.5 The learning framework

The learning framework introduced in this section presents a general approach to automate the process of game learning. The framework is built from the premise that game knowledge consists of two distinct components. The first component is a set of heuristics, called *features*, that can be expressed in a symbolic form. The second component is a set of floating-point values, called *weights*. The features and weights are combined to form a mechanism by which game states can be evaluated, and consequently this mechanism can be used by game playing agents. The objective of the learning agent is to find the features, and to optimise the weights.

The framework consists of a *macro cycle* that has four general stages. The cycle is shown in Figure 5.2. During the play stage, example positions are produced and provided as input to the discover stage. The discover stage addresses the first knowledge component by deducing relevant features from the given examples. These features are used in the optimise stage where ideal values for the weights are sought. The evaluate stage considers the value of the new knowledge, and creates new individual agents. These new agents are input to the play stage where they are used to create new examples.

The macro cycle learning method employs coevolution. The manner in which the knowledge coevolve in the cycle is not same as the coevolution described in Section 5.4.4. In competitive coevolution, also called *predator-prey* coevolution, the interaction between the individuals is such that the dominance of one leads to the extinction of the other – if one wins the other loses [19]. The macro cycle is an example of another coevolution type, called *cooperative coevolution.* In this case, disparate 'species' cooperate to improve the ability of the learner. As the learning progresses, the individuals improve and better examples are produced. Better examples lead to better features that are optimised to produce better individuals.

In the subsections that follow, each stage of the macro cycle is described in more detail.

The play stage

The individuals that evolve in the play stage are the *stage players* that produce the examples. For the initial cycle, two random moving players are used to generate the first set of examples. In subsequent cycles, one or more new players are provided to the play stage. The new players are then used to generate the new examples.

The examples produced by this stage is simply a set of play-lines. Each play-line is the sequence of moves of a game between a stage player and a random moving player. The examples are written to a *play-file.*

The discovery stage

At the start of the *discovery stage*, the play-lines in the play file are rated. The rating procedure classifies a position as a win position, lose or draw position. Also, the game phase of the position is determined. These rated positions are written to a *position-file* that is read by the knowledge discovery algorithm. The output of this algorithm is a phased evaluation function represented in the feature language $\mathcal{F}$ (Section 3.6). The detail of the discovery algorithm is introduced in a later chapter.

The current study defines a total of 23 game phases for C , and consequently the phased function generated contains 23 evaluation functions. Every time a checker is captured the game phase increments. In other words, the phase of the game is a count of the number of pieces removed from the game.

The optimise stage

The new phased evaluation function is input to the *optimise stage.* During this stage optimal values for the weights associated with newly discovered features are sought. For this, the PSO algorithm is used. Like the previous stage, a subsequent chapter is dedicated to describe the detail of the techniques developed for the framework.

The evaluate stage

During the evaluate stage, the new knowledge (i.e. an optimised phased evaluation function) is incorporated into a *champion list.* The champion list is kept in order of descending strength, and a new champion competes with the functions in this list (starting from the top) to determine its placement in this list.

5.6 The performance of a function

In this section, a method to evaluate a player's performance is described. It is not possible to measure the progress of the learner if the method to evaluate the resulting playing agent is undecided. Let f be a phased evaluation function, and N be the number of games played in a competition between an agent that uses f and random moving player. If $L(f)$ denotes the number of games lost and $W(f)$ denotes the number of games won, then the performance measure of f is determined as follows:

$$F(N, p) = \frac{N + W(f) - L(f)}{2 \times N} \times 100 \tag{5.7}$$

During the measurement, the number of games played as first player and second player are equal (that is $N/2$). The measurement ranges from 0 to 100, and a greater value indicates a better performance.

This measurement is equivalent to the Franken and Engelbrecht [25] function called the *F-measure*. Like the F-measure, this function also accounts for the large number of draws that often arises in C matches. Clearly, a larger value for N would increase the accuracy of the measurement. Franken and Engelbrecht [25] used a value of 200 000 and Messerschmidt and Engelbrecht [40] used 20 000 for T - - as a value for N. This is a ten-fold increase in the computational resources required to measure the performance of an agent. The value chosen for N depends on the required accuracy, and also on the available computing resources. The need for computing resources is determined by the game chosen as domain - for example, a single T - - match requires much less computation than a single C match.

In order to determine a reasonable value for N for the C game, an illustrative experiment was conducted using five different phased evaluation functions {A,B,C,D,E}[§]. A total of 20 000 games were played using each of the functions. The value of Equation 5.7 was measured using an increasing value for N at intervals of 1000. Then, using the final measurement (that is $F(20000, p)$ as a yardstick, the 'error' of the preceding measurements was calculated as absolute value of the difference the measurement and the yardstick. Thus, a high error value at interval I indicates that $F(I, p)$ is an inaccurate prediction for $F(20000, p)$. The result of these calculations can be seen in Figure 5.3. At $N = 15000$, the error is well below 0.5 (the horisontal line). In effect, this is an error of 0.5%, and therefore a value of 15000 for N is considered adequate.

[§]The idea was to use a few arbitrary functions for this experiment. The five functions used here were taken from initial work conducted on the discovery process. The detail of this process is described in the next chapter.

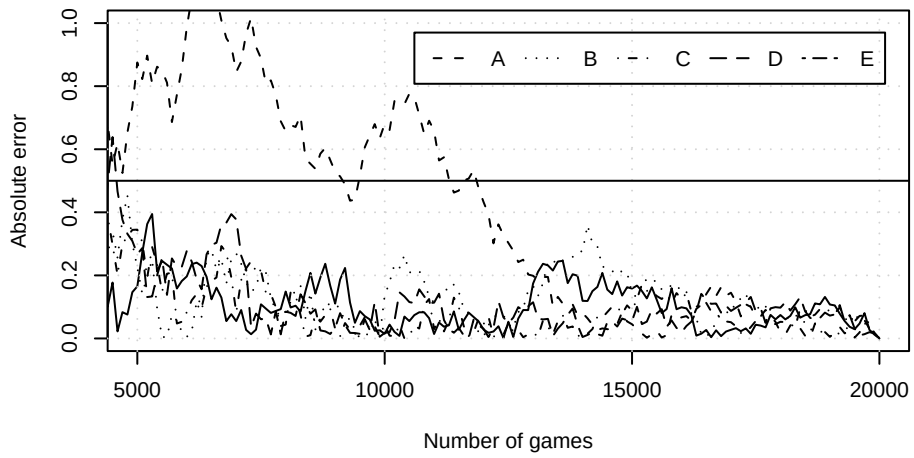


Figure 5.3: Error measurements of 5 functions

5.7 Conclusion

The use of a phased evaluation function instead of a single linear evaluation function introduces non-linearity into the evaluation process. Research endeavours using phased functions indicate that these function do indeed contribute to an improvement in the performance of learning agents. As a research problem, the discovery of features for linear functions is not as active as the problem of weight optimisation. The approach Buro used for GLEM finds the specific features that were found in a representative number of instances of a given example set.

Reinforcement is a general learning technique founded on the principles of supervised learning to find optimal values for weights. TD-learning is an important innovation on reinforcement learning specifically developed for multi-step tasks that impose a delayed reward. Although TD-learning has been shown to be an excellent learning method for B , other researchers failed to achieved comparable successes.

As a learning method, the macro-cycle of the learning framework uses coevolution. Examples are used to deduce features, the weights of the features are optimised to produce new players that in turn, creates new examples. In the next chapter, the discovery stage of the macro-cycle is introduced. The chapter after that describes the PSO algorithm in detail and introduces a new method to determine the fitness of individuals in the population.

Chapter 6

Knowledge discovery with ID3

The previous chapter introduced the macro cycle of the learning framework. This chapter covers the techniques used during the Discovery Stage of the macro learning cycle. The aim is to provide background knowledge of ID3 and describe how it is employed to discover game knowledge. Also, a novel method that utilises the ID3 decision tree induction algorithm to deduce an evaluation function, is introduced. In addition, an empirical analysis of variations in the deduction method is conducted to identify the strategy that is most suitable for the learning framework.

6.1 Introduction

Chapter 3, introduced the feature language, $\mathcal{F}$, as a set of symbols and operators that can be used to compose an evaluation function. The value of an expression in $\mathcal{F}$ can be determined from features that are directly observable on a game position. It has also been concluded that the evaluation of a game position is dependent on the game phase; mostly because the significance of a feature changes as the game progresses. The need for a function that adapts as the game goes through various stages lead to the definition of the phased evaluation function concept (see Section 5.2).

The focus of this chapter is on the first active step of the macro cycle described in Section 5.5: the Discovery Stage. It is during this stage, that the significant features of game positions are discovered and converted into phased evaluation functions that contain $\mathcal{F}$-expressions. The discovery process has as input the set of example positions produced by the Play Stage of the macro cycle. Using ID3, the significant features of these examples are identified, and the phased evaluation functions are composed from these features.

The process of discovery involves the induction of decision trees from a large set of example game positions. As background, a description of decision trees and their relation to and-or trees is provided in Section 6.2. The ID3 algorithm, described in Section 6.3, is used to induce the decision trees. This section also contains the details of entropy based attribute selection methods. Section 6.4 introduces a method by which the example game positions are encoded to become useful as input to ID3. New algorithms and strategies to deduce an evaluation function from a decision tree is introduced in Section 6.5. This section also contains specific details regarding the use of C4.5. Two experiments conclude this chapter: in Section 6.6 the performance of functions produced by the deduction strategies are compared, and the complexity of the functions produced by the stronger strategies are compared in Section 6.7.

6.2 Decision trees

Many artificial intelligence problems can be viewed as classification problems [73]. For example, the ideal C player is one that can faultlessly identify the subset of the available moves that leads to a win. The ability to classify can be described as the ability to predict a particular attribute value of an entity based on the values of the other attributes of that entity. The attribute to predict is called the *target attribute*, and its value is referred to as the *class* of the entity. A *decision tree* is a structure that contains rules that are used to predict the class of a given entity. A leaf node of a tree indicates the class, and a non-leaf node, called a *decision node*, specifies a condition to test. Each branch leading from a decision node signifies a possible outcome for the test. The classification process starts at the root of the tree, tests the value of the attribute specified in the condition and follows the applicable branch to the next node. This process continues until a leaf node that defines the class for the entity, is reached [31].

When compared to other classifiers such as neural networks and genetic algorithms, the distinguishing characteristic of decision trees is the *readability* of the knowledge that drives the classification process. The readability of the rules represented by a decision tree facilitates the discernment, analysis and evaluation of the classification process. Another advantage of decision trees is that the structure is easy to manipulate. In particular, the ability to transform decision trees into and-or trees, and logical expressions makes them very useful for the current research.

A disadvantage of decision trees is that the decision structures can become quite complex. Two primary motives drive the need to simplify decision trees. The first is to make it easier for

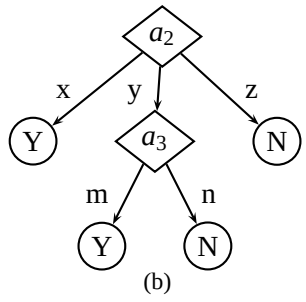
humans to interpret the tree, and the second is to rid the tree of inaccurate branches and to avoid overfitting [37]. Inaccuracies are introduced by noisy data and by induction with an inadequate example set. The typical approach, called *pruning*, involves the removal of the lower branches in the tree. When the pruning occurs during the induction process, it is called *on-line pruning*. In *post pruning*, the tree is simplified after the induction process. Post-pruning methods can be used to improve over-fitted trees. *Minimal object pruning* is an on-line pruning method that prevents the exploration of a node that has less than three branches that classify a specified minimum number of entities [51].

Arguably the most important quality criterion of a decision tree is the *classification accuracy* of the tree. A straight forward measurement of accuracy is simply the fraction of examples that are correctly classified. However, sometimes the identification of a specific class is more important than the identification of other classes. In such cases two types of errors can be identified: the *error of omission* considers the examples that should have been positively classified - but were not (i.e. false negatives); and the *error of commission* considers the examples that should not have been positively classified - but were (i.e. false positives). Omissions indicate over-specialisation, while over-generalisation is the cause of commissions. The aim is to develop concepts that are *consistent* and *complete*. A consistent concept does not produce an error of commission and a complete concept does not produce an error of omission [37].

An example serves best to demonstrate the utility of a decision tree. Consider a set of entities, $\{\vec{e}_1, \ldots, \vec{e}_8\}$ with attributes a_1, a_2 and a_3 and values as listed in Table 6.1(a). The decision tree shown in Figure 6.1(b) can be used to determine the class of entity $\vec{e}$. From the root node, the path $a_2 = z$ leads to a predicted classification of Y for $\vec{e}$.

Entity	a_1	a_2	a_3	Class
e_1	a	x	n	Y
e_2	b	x	n	Y
e_3	a	y	n	N
e_4	a	z	n	N
e_5	a	y	m	Y
e_6	b	y	n	N
e_7	b	y	m	Y
e_8	a	y	m	Y
e	b	z	m	?

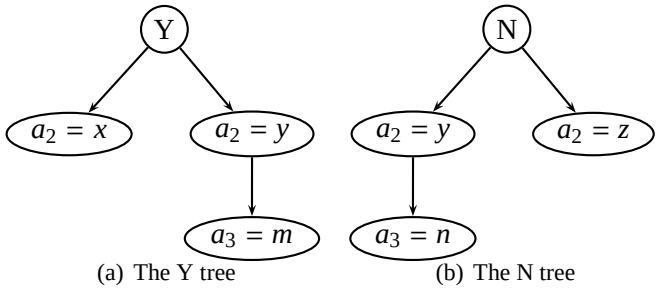
(a)



(b)

Figure 6.1: A set of entities and attribute values

To demonstrate the transformation of decision trees into and-or trees, consider the conver-



(a) The Y tree (b) The N tree

Figure 6.2: And-or trees derived from Figure 6.1(b)

sion of the decision tree in Figure 6.1(b) into the two *and-or trees* in Figure 6.2(a) and Figure 6.2(b). In an and-or tree, the branches leading from a node signify the ORs and the path from the root to a node signifies ANDs. The tree in Figure 6.2(a) can be read as *IF* $a_2 = x$ *OR (*$a_2 = y$ *AND* $a_3 = m$*) THEN the classification is Y*.

6.3 The ID3 algorithm

The purpose of the ID3 algorithm [50] is to induce a decision tree from a set of examples. It is also known as the Top Down Induction of Decision Trees algorithm, or TDIDT. The algorithm is based on the a divide and conquer principle: the initial example set is split into smaller subsets, which are then further divided in subsequent iterations of the macro cycle. It is also a greedy algorithm because a decision to split a set is final – there is no backtracking. An outline of the ID3 algorithm is given in Figure 6.3.

Note that the selected attribute is removed from the set before the recursive invocation of the algorithm. It is therefore conceivable that the algorithm is invoked with A as an empty set - however such an invocation is possible only if the example set contains entities with the same value for every attribute but with different classifications [8]. It is for this reason that the precondition of the input specifies that at least one attribute value must be different for every pair of examples that have different classes.

The criterion used to select the split attribute, a, on line 1 is the primary influence on the quality of the decision tree that is induced [37]. The *information gain* criterion aims to choose the attribute that maximises the amount of new information that become available after the split is made.

The classification problem will be solved when an attribute is chosen that divides E into homogeneous subsets; if it exists, the selection of such an attribute would have the maximum

Algorithm ID3
Input A set of attributes $A = \{a_1 \ldots a_n\}$ and a set of examples $E = \{\vec{e}_1 \ldots \vec{e}_m\}$. Each example contains an array of n values such that there is a value for every attribute in A (for some classification problems a predefined 'null' value can be used to specify missing values). Every example has a classification. As a constraint, all elements in the example set with the same values for the attributes must have the same classification.
Output A decision tree T

```
ID3(A,E,T)
1:    Given E, select the best attribute a;
      Create T as a tree with a as root;
      Use a to split the examples in E into a collection
          of sets S = {S_1, ..., S_k}, such that all examples
          in S_i ∈ S has the same value for the attribute a;
      for each S_i in S
          if all examples in S_i have the same classification
              Create a leaf node labelled with the
                  classification value and attach it to the root of T;
          else
              ID3({a}, S_i, T_i);
              Attach T_i to the root of T;
```

Figure 6.3: An outline of the ID3 algorithm

information gain as consequence. Conversely, an attribute that creates subsets that have an equal distribution between classifications does not aid in solving the classification problem. If selected, this attribute would contribute the minimum amount of new information. These two extremes demonstrate the ideas that lead to a concept called *entropy* [58], borrowed from information theory and used as the attribute selection criterion in ID3.

One way to understand the *entropy selection heuristic* is to consider the use of binary digits to distinguish elements. A single binary digit is able to distinguish between two elements; three binary digits can distinguish between 2^3 elements and in general, b binary digits can distinguish between 2^b different elements. By encoding every element into a string of b binary digits a sequence of k elements can be represented as a binary digit string of length $b \times k$. This encoded string is said to contain $b \times k$ *bits* of information.

This simple binary encoding may seem close to optimal, however, the probability distribution of the occurrence of the classes can be used to encode the same information in less bits. A shorter average bit count per element is ensured by using the shortest binary string to encode the element that occurs most frequently. For example, consider an experiment that observes passing cars with a colour from the set {white, red, blue, silver}. For these four elements, the simple

encoding would require $\log_2 4 = 2$ bits for each element in the observed sequence. However, previous observations determined that there is a 0.5 probability that the next car is white, that is Pr(white) = $\frac{1}{2}$, and for Pr(red) = $\frac{1}{4}$, Pr(blue) = $\frac{1}{8}$ and Pr(silver) = $\frac{1}{8}$. Using these probabilities, the following binary encodings would deliver an optimal binary string: white = 0, red = 10, blue = 110, silver = 111. For the cars experiment, the average bit count has been reduced to:

$$\Pr(\text{white}) \times 1 + \Pr(\text{red}) \times 2 + (\Pr(\text{blue}) + \Pr(\text{silver})) \times 3 = 1.75 \text{ bits}$$

Entropy is a measure of the information content of a distribution of discrete values. This measurement is based on the number of bits required to encode each value in the distribution: a higher bit count signifies a larger disarray of the information, and also a higher entropy. Using the method described in the previous paragraph, it follows that the number of bits required to encode an element, e, is $-\log_2 \Pr(e)$ (e.g. a white car needs $-\log_2 \frac{1}{8}$ bits). The entropy of a set of elements $\mathbf{S}$ in which every element is from a finite set of classes $\{c_1, \ldots, c_t\}$, is calculated as follows:

$$H(S) = -\sum_{i=1}^{t} \Pr(c_i) \times \log_2 \Pr(c_i) \tag{6.1}$$

Here $\Pr(c_i)$ is the probability of finding an instance of c_i in $\mathbf{S}$.

In ID3, the selected attribute is used to create a new configuration from the set. The new configuration contains a number of smaller subsets such that all elements in each subset has the same value for the selected attribute. The entropy of each subset can be determined using Equation 6.1. The entropy of the new configuration is determined by the weighted sum of the subset entropy values. For an attribute a that splits a set $\mathbf{S}$ into k subsets, $\mathbf{S}_i, \ldots \mathbf{S}_k$, the entropy of the new configuration is calculated as follows:

$$H(S, a) = \sum_{i=1}^{k} \Pr(\mathbf{S}_i) \times H(\mathbf{S}_i) \tag{6.2}$$

The weight used in the sum, $\Pr(\mathbf{S}_i)$, is the probability that an element in $\mathbf{S}$ is also in $\mathbf{S}_i$. This probability can be estimated from the size of the subset:

$$\Pr(\mathbf{S}_i) = \frac{|\mathbf{S}_i|}{|\mathbf{S}|} \tag{6.3}$$

In order to choose the attribute that maximises the amount of new information, the attribute that brings more order to the configuration must be chosen. That is, the attribute that brings

about the greatest decrease in entropy. For a set **S**, the entropy of the configuration before the split is $H(\mathbf{S})$, and the entropy after splitting into the subsets sired by attribute a is $H(\mathbf{S}, a)$. Thus, the information gain for attribute a, is calculated as follows:

$$I(\mathbf{S}, a) = H(\mathbf{S}) - H(\mathbf{S}, a) \tag{6.4}$$

The information gain heuristic selects the attribute with the highest information gain value.

When used in colligation with Equation 6.3, the information gain measure prefers attributes with many values [37, 31]. Such attributes would have a large number of subsets, each with a very small occurrence probability, resulting in a high information gain measure. As an example consider the *patient's full name* as the attribute a in database **S** that classifies illnesses of m patients. Assuming m is relatively large, the number of subsets for this attribute will be close to m - and the value if $P(\mathbf{S}_i)$ for each subset S_i will be close to zero. Consequently, $H(\mathbf{S}, a)$ will be a very small number and $I(\mathbf{S}, a)$ would be close to $H(\mathbf{S})$, making attribute a a likely candidate. However, this attribute has nothing to do with the illness classification, and selecting it as an important decision factor would be disastrous.

An alternative measure addresses this problem by penalising attributes with many values [73]. This measure, called the *gain ratio,* uses a term called the *split information* measure. For an attribute a that splits **S** into the subsets $\{\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_k\}$, the split information measure is calculated as follows:

$$S(\mathbf{S}, a) = -\sum_{i=1}^{k} \Pr(\mathbf{S}_i) \times \log_2 \Pr(\mathbf{S}_i) \tag{6.5}$$

Comparing this equation to Equation 6.1 reveals that the split information measures the entropy of **S** with respect to a, where $H(\mathbf{S})$ measures the entropy of **S** with respect to the target attribute. A higher split information measure indicates a more disorganised attribute – and, if possible, the selection of such attributes should be avoided. The gain ratio measure is an attribute selection heuristic that introduces this desired effect. It is calculated as follows:

$$R(\mathbf{S}, a) = \frac{I(\mathbf{S}, a)}{S(\mathbf{S}, a} \tag{6.6}$$

6.4 Encoding position examples for ID3

Armed with the background of decision trees and the ID3 algorithm described in the sections above, the focus in this section returns to the discovery of the knowledge used by game playing

agents. These agents use the knowledge to compare game positions. In order to uphold the zero-knowledge principle, only features that are directly observable from the game position can be used in this assessment. Of all the observable features, only a small subset have a significant influence on the game outcome. The task of learning agent is to identify the features that belong to this significant subset.

The identification of the significant features can be regarded as a classification problem. The observable features are classified into two groups, those that are significant and those that are not. Consequently, it seems reasonable to explore the possibility of using ID3 to induce significant features from example game positions. In the paragraphs that follow, an approach is proposed whereby a game position is encoded into a structure that can be used as input to the ID3 algorithm. This ID3 structure has two requirements: it needs an ID3 classification and it needs a set of ID3 attributes.

6.4.1 Position classification

Insofar the game player is concerned, the most interesting aspect of any position is whether or not this position leads to a win ($\mathcal{W}$), a lose ($\mathcal{L}$) or a draw ($\mathcal{D}$). It therefore makes sense to use the game outcome to classify the ID3 examples – i.e. a game position is classified as an element from $\{\mathcal{W}, \mathcal{L}, \mathcal{D}\}$.

The classification of a position is determined from the game lines generated during the preceding stage of the macro cycle (see Section 5.5). The positions in each game line are credited for their contribution (or their lack of contribution) to the outcome of the game. For example, if the outcome of the game is a win for the first player, positions chosen by the first player is ascribed with a higher likelihood to lead to a win than the positions chosen by the second player. Thus, a tally of the outcomes of the game-lines in which it appears determines the position's classification.

Essentially, the outcome with the greatest count dictates the classification of the position. But, special cases arise when more than one outcome has the same count. An equal count for $\mathcal{W}$ and $\mathcal{L}$ classifies the position as $\mathcal{D}$. In the case where the $\mathcal{D}$ count shares the maximum with another outcome, the latter is chosen as the classification of the position in question. For example, when the number of losses is 23, the number of wins is 24 and the number of draws is 24, the position is classified as $\mathcal{W}$. This allocation is made because the knowledge value of the "not lose" classification is higher than the knowledge value of a "draw" classification; and the

example position is more likely to lead to a $\mathcal{W}$ than a $\mathcal{L}$.

6.4.2 Position attributes

The feature language $\mathcal{F}$, described in Section 3.6, is adequate to *express* an attribute, but it does not *define* the set of attributes. An attribute used to encode a position consists of an $\mathcal{F}$-expression that is interpreted as a boolean expression. Because ID3 requires an enumerated attribute set, and because of the practical limitations on time and space, the feature language expressions must be chosen carefully. The constraint imposed by ID3 is that the attribute set must be able to distinguish between any two positions with dissimilar classifications. It is also possible to select attributes based on game knowledge, but such an approach would spurn the zero knowledge principle introduced in Section 3.4.

For the current research, the attributes are created from easily identifiable regions on the chequerboard. One region is the entire board, the other regions are the set of ranks (rows), the set of files (columns) and the set of diagonal lines. The set of concentric squares starting from the four squares in the centre of the board and ending with the squares on the border are also included as regions. Selecting regions in this way is arbitrary enough to be considered free of game knowledge, and organised enough to be quickly recognisable by observation. The list of regions are listed in Table 6.1 (see Section 3.6.1 for the notation used for squares).

These regions are used in Fuzzy Occupation Feature expressions to define attributes. Recall from Section 3.6.2 that the Fuzzy Occupation Feature takes the form $\mathbf{T} \otimes \mathbf{R}$, where $\mathbf{T}$ is a subset of the set of occupation states, $\mathbb{O}$. There is a very large number of expressions that is of this form; and it is from a subset called the Attribute Expression Set, $\mathbf{E}$, that the attributes are defined. One attribute is declared for every $\mathcal{F}$-expression in $\mathbf{E}$:

Definition 6.1: *Attribute Expression Set.* Consider a set of identified regions, $\mathbf{R}$ and the set of occupation states, $\mathbb{O}$. Then the set of attribute expressions, $\mathbf{E}$, are all the expressions of the form $\{t\} \otimes \mathbf{R}_i$ such that $t \in \mathbb{O}$ and $\mathbf{R}_i \in \mathbf{R}$.

For a position, the value of an Attribute Expression is determined by the number of squares in the region that are of the given occupation type. The value of an ID3 attribute is either *Y* for *yes*, or *N* for *no*. For position, p, the attribute defined by the expression $e = \{t\} \otimes \mathbf{R}_1$ takes a value of *Y* if and only if $V(p, e) > 0$, otherwise it takes the value *N* (V is defined on page 40). Thus, if the position contains any squares in the region of the specified occupation type, it has a value of *Y*.

Region	Description	Squares in region
all	All	$(*,*)$
r1	1^{st} Rank	$(*,1)$
r2	2^{nd} Rank	$(*,2)$
r3	3^{rd} Rank	$(*,3)$
r4	4^{th} Rank	$(*,4)$
r5	5^{th} Rank	$(*,5)$
r6	6^{th} Rank	$(*,6)$
r7	7^{th} Rank	$(*,7)$
r8	8^{th} Rank	$(*,8)$
f1	1^{st} File	$(1,*)$
f2	2^{nd} File	$(2,*)$
f3	3^{rd} File	$(3,*)$
f4	4^{th} File	$(4,*)$
f5	5^{th} File	$(5,*)$
f6	6^{th} File	$(6,*)$
f7	7^{th} File	$(7,*)$
f8	8^{th} File	$(8,*)$
bd1	1^{st} Back Diag	$(1,1)$
bd2	2^{nd} Back Diag	$(3,1),(2,2),(1,3)$
bd3	3^{rd} Back Diag	$(5,1),(4,2),(3,3),(2,4),(1,5)$
bd4	4^{th} Back Diag	$(7,1),(6,2),(5,3),(4,4),(3,5),(2,6),(1,7)$
bd5	5^{th} Back Diag	$(8,2),(7,3),(6,4),(5,5),(4,6),(3,7),(2,8)$
bd6	6^{th} Back Diag	$(8,4),(7,5),(6,6),(5,7),(4,8)$
bd7	7^{th} Back Diag	$(8,6),(7,7),(6,8)$
bd8	8^{th} Back Diag	$(8,8)$
fd1	1^{st} Forw. Diag	$(1,7),(2,8)$
fd2	2^{nd} Forw. Diag	$(1,5),(2,6),(3,7),(4,8)$
fd3	3^{rd} Forw. Diag	$(1,3),(2,4),(3,5),(4,6),(5,7),(6,8)$
fd4	4^{th} Forw. Diag	$(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8)$
fd5	5^{th} Forw. Diag	$(3,1),(4,2),(5,3),(6,4),(7,5),(8,6)$
fd6	6^{th} Forw. Diag	$(5,1),(6,2),(7,3),(8,4)$
fd7	7^{th} Forw. Diag	$(7,1),(8,2)$
cc1	1^{st} Square	$(4,4)(5,5)$
cc2	2^{nd} Square	$(3,5),(3,3),(5,3),(6,4),(6,6),(4,6)$
cc3	3^{rd} Square	$(2,6),(2,4),(2,2),(4,2),(6,2),(7,3),(7,5),(7,7),(5,7)$
cc4	4^{th} Square	$(1,7),(1,5),(1,3),(1,1),(3,1),(5,1),(7,1),(8,2),(8,4),$ $(8,6),(8,8),(6,8),(4,8),(2,8),$

Table 6.1: Regions used for attributes

6.5 The discovery algorithm

The discovery algorithm is introduced in this section. The purpose of this new algorithm is to build a phased evaluation function. As input the discovery algorithm is given a set of example positions. These positions are encoded and classified for ID3 using the method described in Section 6.4. Before ID3 is applied, the example positions are divided into *phased example sets*,

such that all positions in the set are from the same game phase. ID3 induces a decision tree for each game phase from the example set of that phase.

The decision rules in each tree are manipulated and transformed by the function deduction algorithm. This algorithm produces a function with terms weighted with 1.0. These functions are then combined to form the phased evaluation function used by the agent during game play. The outline of the algorithm is listed in Figure 6.4.

Algorithm Function discovery algorithm
Input A set of positions A encoded for ID3 usage and labelled with "phases".
Output A phased evaluation function F

Discover(A,F)
  For each phase
    Create a phased example set $A_i = \{e \mid e \in A \ \wedge \ e$ has phase $i\}$
  For each phased example set, A_i
    Induce a decision tree T_i
    Deduce an evaluation function F_i
  Combine all F_i to form F

Figure 6.4: An outline of the Function Discovery algorithm

For the induction of the decision trees, a well known program called C4.5 is used [51]. In section Section 6.5.1 the C4.5 parameter settings and the output of C4.5 is described. The function deduction algorithm is described in Section 6.5.2. This section concludes with a discussion on strategies to simplify the resulting evaluation function in Section 6.5.3.

6.5.1 Using C4.5

The C4.5 software [51], developed by Quinlan, implements the basic ID3 algorithm together with a variety of more advanced features that can be used to improve and streamline the decision tree. Quinlan has decided to discontinue the maintenance and support for C4.5, but improvements to the software (e.g. C5.0) can be purchased. Fortunately, at the time of this writing the source was still available from Quinlan's web site (http://www.rulequest.com/Personal)*.

For the deduction algorithm, the quality of the nodes in the decision tree is important. C4.5 measures the classification accuracy for each decision as the fraction of the examples from a test set that are correctly classified. For this purpose, $\frac{1}{3}$ of the examples are removed from the original phased example set and made available to C4.5 as 'unseen' examples. The other

*In order to compile the code using the latest version of GCC, it was necessary to make a few minor modifications.

Algorithm Function deduction
Input An ID3 tree, T
Output An evaluation function

```
    DEDUCE(T)
1:      Let T_W be the W and-or tree extracted from T
2:      Let T_L be the L and-or extracted from T
        Initialise empty function, F
        For each sub-tree, N of the root T_W
3:          Append the positive term from N to F
        For each sub-tree, N of the root T_L
4:          Append the negation of the term from N to F
        return F
```

Figure 6.5: The function deduction algorithm

66% of the examples are used to induce the decision tree. After the induction, the accuracy of the decision nodes are estimated by counting the number of the unseen examples classified correctly.

The information gain ratio is the default split attribute heuristic used by C4.5, and consequently it was chosen as the method to use. In order to avoid inaccurate decision rules, minimal object pruning was employed with a minimum value of 32. Although C4.5 implements post-pruning, the post-pruned tree was not used.

6.5.2 The function deduction algorithm

In the decision tree, the branches that lead to $\mathcal{W}$ describe attributes that were found on winning positions, and these are the attributes the player should strive to obtain. Conversely, the attributes described on the branches that lead to $\mathcal{L}$ should be avoided. From this premise, a new algorithm is introduced here, and it is called the function deduction algorithm.

A sub-tree of the decision tree can be obtained by eliminating all branches that do not lead to $\mathcal{W}$. Transforming this sub-tree into an and-or tree (as described in Section 6.2) produces the $\mathcal{W}$ and-or tree. Likewise, the $\mathcal{L}$ and-or tree can be constructed. For the evaluation function, feature expressions derived from the $\mathcal{W}$ and-or tree are combined with the negation of expressions derived from the $\mathcal{L}$ and-or tree. The outline of the procedure to derive functions terms from the decision tree is listed in Figure 6.5.

The 'append' operation used in line 3 and line 4 completes the term of F by adding a weight of 1.0 to the expression derived from the node N. This derived expression is simply the and-or

expression of N, where the *and*s are substituted with $\wedge$ and the *or*s are substituted with $\vee$. The rules of association (e.g. $A \wedge (B \vee C)$) are maintained in the resulting feature expression.

The details of the algorithm is best illustrated by an example. The decision tree depicted in Figure 6.6 is generated by C4.5. Instead of labelling the branches a simple convention is used: a branch that moves from the decision node towards the left signifies a *yes* decision and a branch to the right signifies a *no* decision. Each node in the C4.5 decision tree has three useful

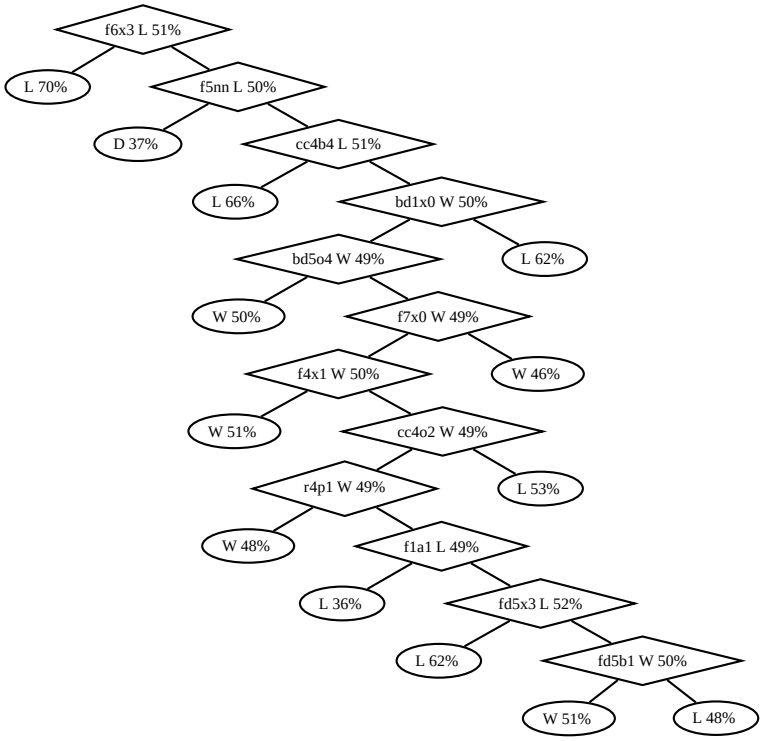


Figure 6.6: A decision tree generated by C4.5

properties: the *decision*, the *dominant class* and the *accuracy*. These properties are shown as the label of the nodes in the figure. The first part of the label represents the decision, the letter in the middle indicates the dominant class, and the accuracy is expressed as the percentage at the end of the label.

The *decision* of the root node is encoded as `f6x3`, and translates to the shorthand feature expression {x3} $\otimes$ {f6}. The meaning of `x3` in this expression is explained in Section 3.6.1; it refers to a square occupied by an active piece that is able to jump or to move away from its current position. The region key `f6` refers to the 6^{th} file of the chequerboard (see Table 6.1). Thus, the decision at the root node reads as the following question: *are there active pieces on file 6 that are able to move or able to jump?*. If, for a given position, the answer is *yes* the decision flows to the left, and the tree concludes that the position is likely to lead to a losing outcome.

The *dominant class* is the class most frequently encountered during testing. For the root node, the label specifies an `L` indicating that the example set used for testing has more $\mathcal{L}$ positions than any of the other classes.

The *accuracy* of the node is based on the number of classification errors accumulated for the node. For a leaf node, a classification error occurs when an example has an attribute set that corresponds with every decision that lead to the node, but it has a classification that disagrees with the node. The *certainty percentage* is a measure of the accuracy calculated as the ratio of the number of correct classifications over the total number of examples that agreed with the node's decision. For example, if 20 examples from the test set had attributes that coincided with the decisions that lead to a leaf node labelled with $\mathcal{W}$, and 14 of those turns out not to be $\mathcal{W}$ positions, the certainty of that node is 70%. For a decision node (i.e. non-leaf node), the certainty percentage is the sum of the correct classifications over the total example count, accumulated for all the descending leaf nodes. For the root node of the tree, the certainty percentage is therefore an indication of the accuracy by which the decision tree as a whole classifies the test examples.

The first steps of the function deduction algorithm (line 1 and line 2 in Figure 6.5) construct and-or trees from the decision tree. A $\mathcal{W}$ and-or tree constructed from the example decision tree is given in Figure 6.7. Notice that the 5 leaf nodes labelled with `W` in Figure 6.6 are transformed into the leaf nodes of the and-or tree. The label of all non-root nodes of the and-or tree contains a feature expression and an expected evaluation. For example, the node below the root is labelled as `f6x3 n` – a game position for which $\{x3\} \otimes \{f6\} > 0$ is not considered by this and-or tree to be a $\mathcal{W}$ position.

In line 3 of the function deduction algorithm, the terms of the evaluation function is constructed from the and-or tree. The and-or tree in Figure 6.7 has only one sub-tree, and only one

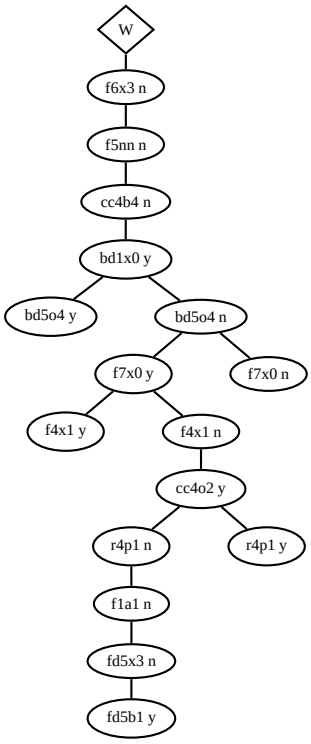


Figure 6.7: A $\mathcal{W}$ and-or tree derived from the decision tree in Figure 6.6

term would be added to the evaluation function. In shorthand form, this term would look like:

$$
\begin{aligned}
1.0\times \quad & (\neg\{f6\}\otimes\{x3\} \ \wedge \ \neg\{f5\}\otimes\{nn\} \ \wedge \ \neg\{cc4\}\otimes\{b4\} \ \wedge \ \{bd1\}\otimes\{x0\} \\
& \wedge (\{bd5\}\otimes\{o4\} \\
& \vee(\neg\{bd5\}\otimes\{o4\} \\
& \wedge ((\{f7\}\otimes\{x0\} \ \wedge \ (\{f4\}\otimes\{x1\}) \\
& \vee(\neg\{f4\}\otimes\{x1\} \\
& \wedge (\{cc4\}\otimes\{o2\} \ \wedge \ \neg\{r4\}\otimes\{p1\} \ \wedge \ \neg\{f1\}\otimes\{a1\} \wedge \\
& \neg\{fd5\}\otimes\{x3\} \ \wedge \ \{fd5\}\otimes\{b1\}) \\
& \vee\{r4\}\otimes\{p1\}))) \\
\vee\neg\{f7\}\otimes\{x0\})))
\end{aligned}
$$

From a decision tree, many $\mathcal{W}$ and-or trees can be derived. Although all the trees will be logically equivalent, they are different in structure. And-or trees represent logical expressions, and as such, these trees can be normalised. Every logical expression captured in an and-or tree can be converted to the conjunctive normal form (CNF) [28]. A *CNF-tree* is an and-or tree that is in the conjunctive normal form. A sub-tree is attached to the root of a CNF-tree for every leaf node in the decision tree that has the same class as the root of the and-or from which the CNF tree is constructed. As a result, the CNF tree contains more evaluation function terms than the and-or tree. More terms in the evaluation function provide finer control of the weights during

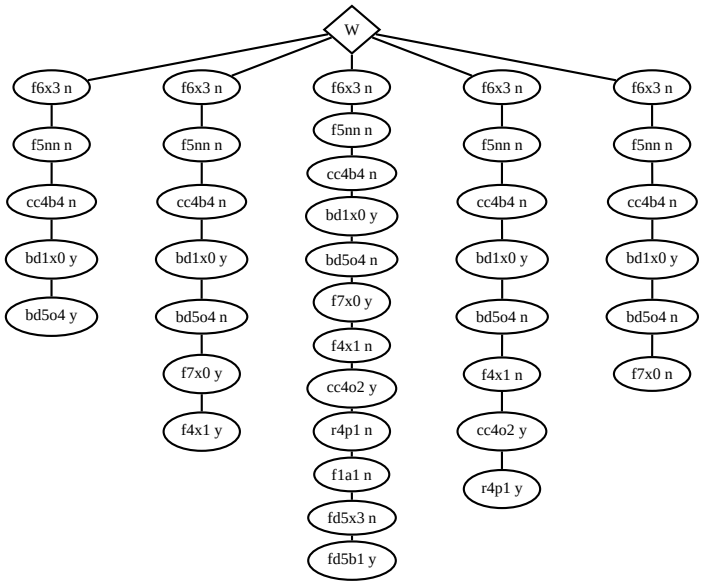


Figure 6.8: A 5 term $\mathcal{W}$ and-or tree derived from the decision tree in Figure 6.6

optimisation, but creates more complex functions. The CNF-tree of the tree in Figure 6.7 is given in Figure 6.8. This and-or tree has 5 sub-trees connected to the root and would produce 5 terms in the evaluation function.

The number of terms in an and-or tree should ideally be more than the minimal tree in Figure 6.7, and should be less than the number of terms in the CNF tree. It is possible to construct a tree closer to this ideal by using the dominant class of the nodes to create sub-trees called dominant clusters. A *dominant cluster* contains at least one leaf node; and it is a set of connected nodes that have the same dominant class. The and-or tree that contains one term for each dominant cluster creates a more balanced number of terms. One method to construct such an and-or tree is to construct the CNF-tree first, and then branches that have the same dominant classifications at the bottom are combined to form a smaller tree.

The dominant clusters for the $\mathcal{W}$ nodes are shown in Figure 6.9. This example has 2 clusters and the $\mathcal{W}$ and-or tree would add 2 terms to the function. The resulting and-or tree can be seen in Figure 6.10 (page 102) .

6.5.3 Simplification strategies

This subsection introduces new simplification techniques that were specifically developed as part of the current research. These simplification techniques eliminates branches from the decision trees using the dominant class and the certainty percentage values determined by the C4.5 program. The aim of the simplification techniques is to produce simpler feature expressions. An

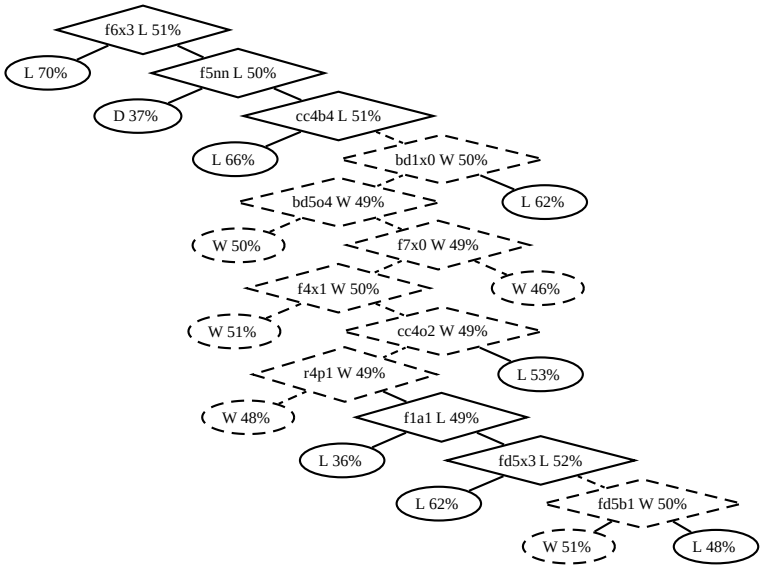


Figure 6.9: A decision tree with $\mathcal{W}$ dominant clusters

evaluation function is easier to understand and it consumes less computational resources when it has less complex expressions.

Term exclusion

The elimination of leaf nodes in the decision tree would result in less terms in the evaluation function. A reasonable strategy is to exclude all leaf nodes with a certainty less than 50%. In order to simplify discussions, this rule is referred to as the *exclude dubious rule (EDR).* EDR

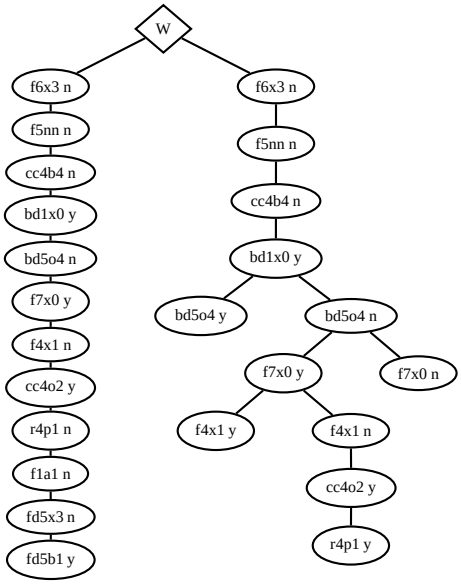


Figure 6.10: The $\mathcal{W}$ and-or tree derived from the decision tree in Figure 6.6 using dominant clusters

assumes that a decision is weak if it could not classify at least 50% of the test cases correctly.

The dominant class of a leaf node's parent could be different than the leaf node. In such an arrangement, the parent node had more instances of another class during testing, consequently a more accurate estimate of the performance of that parent's dominant class can be expected. The rule which will be referred to as the *exclude spurious rule (ESR)* eliminates a leaf node that has a parent with a disagreeing dominant class. This rule eliminates all dominant clusters that contains one node.

The effect of the term exclusion rules on the tree in Figure 6.6 is as follows: EDR will eliminate 2 `W` nodes and 3 `L` nodes. ESR will eliminate no `W` nodes, but it will eliminate 3 `L` nodes. In Figure 6.11, all the nodes affected are lined with dashes. Only the nodes with solid lines will remain after the application of both rules. In total, these rules eliminate 5 terms from the CNF-tree.

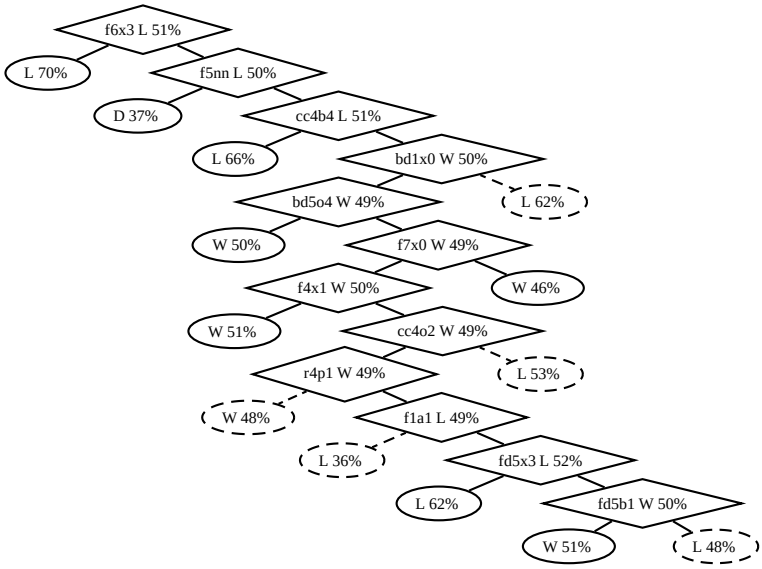


Figure 6.11: A decision tree with excluded leafs marked

Term cutting

Another method to simplify the evaluation function is to make the terms shorter. A term in the CNF-tree can be shortened by selecting a *cut node* in the term and removing the cut node along with all nodes between it and the root node. Because the term has less nodes, it is simpler; but this cutting process could increase the number of terms. The cut node is chosen by 'walking' from leaf to root. The first node encountered in this walk that matches the cut criterion becomes the cut node.

The *cut spurious rule (CSR)* chooses the first node with a dominant class that disagrees with the leaf node. When CSR is applied, every dominant cluster becomes a term; nodes not in a dominant cluster are eliminated. The *cut dubious rule (CDR)* cuts on the first node with a certainty less than 50%.

The effect these rules have on the $\mathcal{W}$ and-or tree can be seen in Figure 6.12 and Figure 6.13. Note that the CDR tree has less nodes, but more terms.

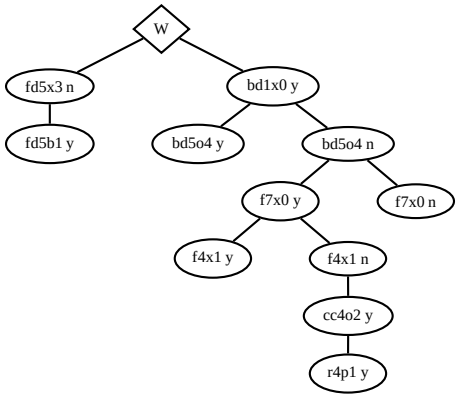


Figure 6.12: The $\mathcal{W}$ and-or tree derived from the decision tree in Figure 6.6 using the Cut Spurious Rule

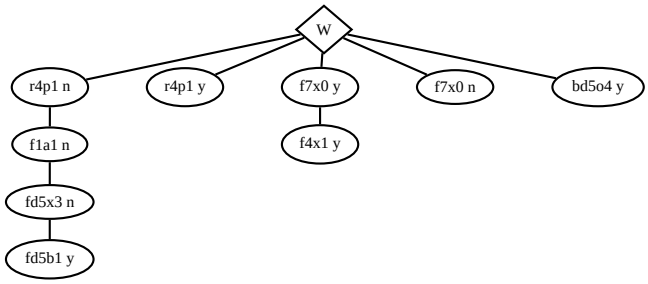


Figure 6.13: The $\mathcal{W}$ and-or tree derived from the decision tree in Figure 6.6 using the Cut Dubious Rule

Combinations

A simplification strategy can be obtained by choosing an exclusion rule and a cut rule to use. This choice provides a total of 9 different simplification strategies. A two letter acronym is assigned to each strategy - as listed in Table 6.2. For example `DS` identifies the strategy that chooses the exclude dubious rule and the cut spurious rule. In other words, only accurate leaf nodes and non-leaf nodes within a dominant cluster will be used to construct the terms for the evaluation function.

	Exclude rule		
Cut rule	None	Spurious	Dubious
None	NN	SN	DN
Spurious	NS	SS	DS
Dubious	ND	SD	DD

Table 6.2: Simplification strategies

6.6 Experiment: Performance comparison

Objective

The objective of the experiment is to investigate whether some of the simplification strategies defined in Section 6.5.3 provide stronger evaluation functions than others. In particular, the 9 strategies listed in Table 6.2 are used to construct evaluation functions; and the performance of these functions are compared.

Method

The quality of ID3 is dependent on the quality of the set of examples provided as input to the algorithm. In order to provide a good variety of this input, 30 example sets have been generated for the experiment. Each set contains all the unique positions encountered from 1000 play-lines that were randomly generated. The number of unique examples in each example set varied between 58 824 and 60 023.

Using a simplification strategy, one phased evaluation function is deduced for every input example set. That is 30 phased evaluation functions for each strategy.

The performance of a phased evaluation function was measured using the method described in Section 5.6. This is the same method used by Franken and Engelbrecht (in [25]), a value greater than 50 indicates more games are won than lost. For the measurement 15000 games were played against a random moving agent. The agent chose a move by selecting the move on the first ply that leads the best evaluation function score. If more than one move results in the same score, the decision is based on a uniform random selection amongst the best scoring candidates.

Results

The aggregation of the 30 performance measurements taken for each simplification strategy is given in Table 6.3. In this table, the best outcome for each measurement is highlighted using an

Table 6.3: The performance statistics of the strategies

Strategy (S)	Mean ($\overline{X}_S$)	Variance (s_S^2)	Maximum	Minimum
NN	59.91990	10.70934	65.3900	51.6567
SN	53.36689	19.92879	62.7067	45.3133
DN	59.89477	14.91340	68.7667	49.5867
NS	*60.84433	11.37601	67.2067	54.2900
SS	54.88689	19.63245	64.3733	45.1667
DS	60.08711	12.65979	67.6767	*54.8267
ND	59.39389	*10.40487	65.8500	53.3133
SD	53.05411	19.67373	61.8933	45.7567
DD	59.74778	14.72119	*68.8600	52.5133

asterisk ('*').

A performance greater than 50 indicates more wins than losses, and the mean values obtained indicate that the strategies are reasonable. The `NS` strategy has the best mean value; approximately 0.8 higher than the second best.

The randomly generated example sets is likely to produce good and bad input sets to ID3; and consequently high variance values should be expected. However, it is interesting to note the stronger strategies have lower variances. The correlation between mean and variance reinforces the interpretation of the mean results: it is reasonable to expect a stronger strategy to be more resilient to fluctuations in input quality.

The greatest maximum- and the greatest minimum performances were also obtained from the strategies that had a higher mean.

More detail of the results are presented in the box-and-whisker diagram in Figure 6.14. In the diagram the dotted line shows the mean of all the evaluation functions; and the crosses mark the mean of each strategy.

The 3 weak strategies (`SN, SS, SD`) are clearly isolated from the strong strategies: the lower quartiles of the stronger strategies are roughly aligned with the upper quartiles of the weak strategies. Apart from `DN`, only the weak strategies had performances of less than 50. The weak strategies all used the exclude spurious rule. This rule eliminates the leaf node if its parent has a dominant class that differs from the class of the leaf node. Clearly, the evidence shows that this strategy is not very effective.

From the obtained results, the distinction between the strong and the weak strategies are clear, but discerning the best strategy from the strong set is less obvious.

Consider the two strategies with the greatest sample mean values: $\overline{X}_{NS} = 60.84433$ and $\overline{X}_{DS} = 60.08711$. The statistical inference to determine whether the difference between these

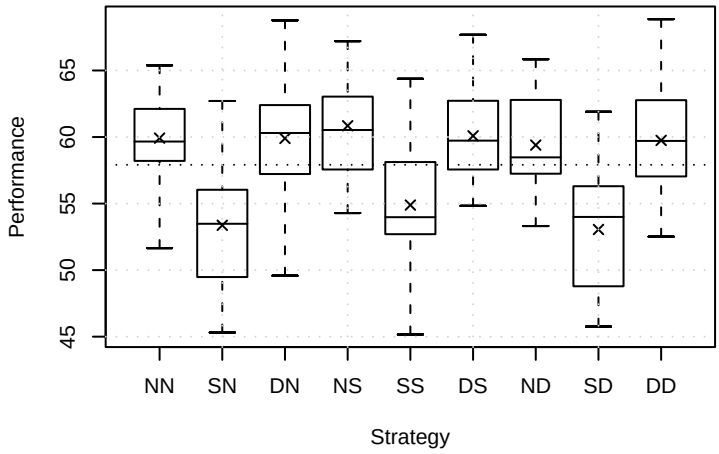


Figure 6.14: A box-and-whisker diagram of the simplification strategy performance

value are significant follows in the paragraphs below. First, the *F*-distribution is used to show that the sample variances are equal; and then the *t*-distribution is used to test the significance of the difference.

The hypotheses, $H_0 : \sigma^2_{\text{DS}} = \sigma^2_{\text{NS}}$ can be tested using the *F*-distribution. A sample size of 30 provides 29 degrees of freedom for both samples; and at a significance level of 5%, the critical value is $F^{(0.025)}_{29,29} = 2.1010$. The observed *F*-value is:

$$\begin{aligned} \frac{s^2_{\text{DS}}}{s^2_{\text{NS}}} &= \frac{12.65979}{11.37601} \\ &= 1.11284 \end{aligned}$$

The observed value is lower than the critical value ($1.11284 < 2.1010$), and H_0 is accepted. Thus, the variances are equal.

Because the population variance of the 2 samples are equal, the *t*-distribution can be used to test whether the population mean of `NS` is greater than the population mean of `DS`. Let μ_n denote the population mean of `NS` and μ_d for the population mean of `DS`. The null hypotheses: $H_0 : \mu_n - \mu_d = 0$ states that the population means are equal. The alternative hypotheses is that $\mu_n > \mu_d$, or $H_1 : \mu_n - \mu_d > 0$. For a significance of 5%, and 58 degrees of freedom, the one-sided critical *t*-value is 1.6716.

The pooled variance s^2 is used to calculate the pooled standard deviation s for the two

samples:

$$\begin{aligned} s^2 &= \frac{29\times(12.65979+11.37601)}{56} \\ &= 12.44711 \end{aligned}$$

$$s = 3.52804$$

The observed t-value is calculated as follows:

$$\begin{aligned} t_{56} &= \frac{\overline{X}_{\mathrm{NS}}-\overline{X}_{\mathrm{ND}}}{s\sqrt{\frac{2}{29}}} \\ &= \frac{60.84433-60.08711}{s\sqrt{\frac{2}{29}}} \\ &= 0.81732 \end{aligned}$$

Because $0.81732 < 1.6716$, H_0 cannot be rejected and with the results obtained from the experiment, it cannot be concluded that `NS` is a better strategy than `ND`.

6.7 Experiment: Complexity comparison

Objective

The objective of this experiment is to compare the complexity of the phased evaluation functions produced by the stronger strategies. A function that is more complex would not only be more difficult to understand but it would take longer to evaluate; and as such would negatively impact the learning rate of an agent.

Method

The method to produce the evaluation functions for this experiment is the same method as described in Section 6.6.

As an estimate of the complexity of the function, the size of the file to which the function was written is used. Two factors contributes to the complexity of the function: the number of terms in the function, and the length of the terms. A measure of these factors is proportional to the size of the file. Also, the implementation of the evaluation algorithm has a time-complexity linear with the file size.

Table 6.4: The file size (in kb) statistics of the strategies

Strategy (S)	Mean ($\overline{X}_S$)	Variance (s_S^2)	Maximum	Minimum
DD	65.34323	24.941943	75.10547	54.64844
DN	69.42389	23.384363	78.85742	59.40137
DS	*20.59616	*0.918905	*22.10352	*18.26074
ND	82.13887	35.069803	91.55859	68.47266
NN	93.20179	24.622274	100.66504	80.09277
NS	23.45993	1.039401	24.90918	20.66699

Results

The measurements obtained for the simplification strategies that proved to be stronger in the previous experiment is given in Table 6.4. In contrast with the results obtained from the previous experiment, the better values of every measured statistic belongs to `DS`. This makes `DS` the prime candidate strategy.

The box-and-whisker diagram in Figure 6.15 shows the results graphically. In a sense, the

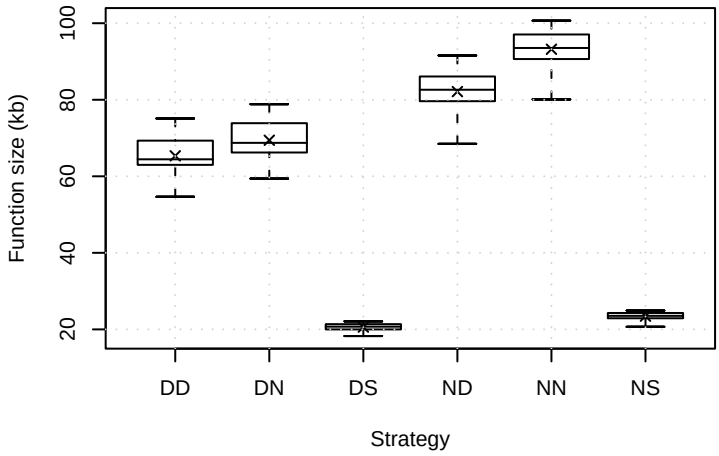


Figure 6.15: A box-and-whisker diagram of the simplification strategy file size

results are predictable: the strategy that does not simplify the function (`NN`) has the highest value; and the strategies that apply cutting and exclusions (`DD` and `DS`) produce simpler evaluation functions than most. However, the effect of cutting terms at a spurious node on the function complexity is very high. This indicates that spurious nodes are more frequently found than dubious nodes; an expected result that confirms ID3's ability to produce effective decision trees. Curiously, `NS` and `DS` also had the best mean performance; suggesting the possibility that simpler

evaluation functions perform better.

6.8 Conclusion

This chapter described the concept of decision trees and the details of the ID3 algorithm. A method to encode game positions in such a way that it can be used in ID3 was described. This method used the expected game outcome of a position as the class and feature expressions as the attributes. The need to identify a set of regions has been highlighted; and an example of the region set useful for games like C has been provided. Furthermore, the discovery algorithm has been described in detail; including the method used to deduce evaluation functions from decision trees. This method is based on and-or trees. Two simplification approaches were introduced that eliminates needless complexity from the evaluation function terms. The first approach is to exclude some terms and the second is to make the terms shorter. This gave rise to nine different simplification strategies.

The results of the experiment to compare the performance of the simplification strategies lead to the conclusion that the exclude spurious rule produces weak evaluation functions. Another experiment to compare the complexity of the evaluation functions produced by the stronger strategies concluded that `DS` and `NS` produce the least complex functions. However, `DS` could safely be isolated as the strategy that produced the simplest functions. Thus, `DS` is a reasonable choice for a simplification strategy to use in the learning framework.

In the next chapter, the Particle Swarm Optimisation (PSO) method is introduced. PSO is the optimisation algorithm that will be used to optimise the weights of the phased evaluation function produced by the discovery algorithm described in this chapter.

Chapter 7

Weight optimisation with PSO

The previous chapter described the method used by the learning framework to discover the terms of the evaluation function. This chapter considers the next stage of the learning framework: the problem of optimising the weights for these terms. The aim is to describe the Particle Swarm Optimisation (PSO) algorithm, and alterations made to the algorithm to optimise playing agents. A new form of PSO that uses tournament is described, along with options for tournament methods. Experiments in this chapter select the most viable option amongst alternative configurations of the new PSO.

7.1 Introduction

Chapter 5 provided a review of the various methods currently employed to optimise the weights for game playing agents. In this chapter, the practicalities of optimising the weights of the evaluation function described in Chapter 3 are considered. The optimisation of weights is a general problem that is independent of the representation scheme in which the weights are found; therefore the methods described in this chapter can also be applied to neural networks (see page 27). However, the optimisation for game playing agents is different from many other optimisation problems because playing agents need opponents. This difference leads to the need for innovative optimisation methods geared for *competitive* optimisation problems.

As explained in Section 5.5, the optimisation method chosen for the learning framework is Particle Swarm Optimisation (PSO). It was first introduced in 1995 by Kennedy and Eberhart [33]. Like genetic algorithms, PSO also emulates a concept from nature (see Section 5.4.4). The metaphor employed by PSO is not genetic, it is the social behaviour of a collection of individuals. Consider a single bird flying in a large flock: if it keeps an eye on its neighbours

while looking for food on the ground below, its chances of finding something to eat is better. The downside of flocking, is that it has to share the food it finds with those neighbours. As attested by many bird species, this is a good survival tactic. It is in this behaviour of flocking birds that the origin of the PSO algorithm is found.

This chapter starts with Section 7.2 that provides an overview of PSO; focussing on neighbourhood topologies and the trajectory of a particle. In Section 7.3, the influence of competition based learning on the PSO algorithm is discussed. It introduces the Competitive PSO that is used as the basis for a new PSO, called the Tournament PSO algorithm.

The last three sections describe experiments conducted to identify suitable configurations for the learning framework. In these experiments, the Tournament PSO is applied to find optimal weights for the evaluation function defined in Section 3.5. Section 7.4 determines whether it is best to include a particle in its own neighbourhood. Section 7.5 measures the performance of various tournament methods, and Section 7.6 considers the behaviour of these tournament methods when the PSO search is extended. Section 7.7 closes the chapter with a brief review and important conclusions.

7.2 Particle Swarm Optimisation

Kennedy and Eberhart [33] introduce *Particle Swarm Optimisation* (PSO) as an algorithm that searches through a multi-dimensional problem space for an optimal solution. An optimal solution is a vector in this search space that maximises (or minimises) a given function. This function, called a *fitness function*, maps a real-valued vector to a real value. During the search, a constant number of search locations, called *particles* are kept current. The search is conducted by changing the velocity of every particle at each iteration. An iteration is referred to as an *epoch*, and the collection of particles form a *swarm*. PSO is distinguished from other searches that keep multiple locations current by the dynamic influence other locations have on the trajectory of a particle [34].

A particle, identified by a search location, has a few key properties. At the start of every epoch, a particle is assigned a new velocity. This velocity vector is added to the current location of the particle to determine the new location. More detail regarding the movement of a particle is provided in Section 7.2.2. The *fitness value* of a particle is the value of the fitness function at the current location of the particle. The *personal best* is a location on the path of a particle at which the best fitness value was achieved. The *neighbourhood* is a static property of a particle; it

is the subset of all the particles that exert an influence on the velocity of a particle. Section 7.2.1 discusses different neighbourhood structures. The *local best* is the personal best of a particle in the neighbourhood, such that no other neighbour have a better personal best.

In addition to parameters that influence the neighbourhood structure and the particle trajectory (discussed in the subsections that follow), two general parameters apply to PSO. The first is the *swarm size* that determines the number of particles in the swarm, and the other is the *termination rule* that specifies when the PSO must stop its search. A typical termination rule is to specify an *epoch limit.* Alternatively, an *evaluation limit* can specify the maximum number of fitness function evaluations allowed. For some fitness functions, it is possible to search until the particle locations converge at some optimal location.

The outline of the PSO algorithm is shown in Figure 7.1. The notation for the location vector of particle p is $\vec{p}$. The personal best of p is $pbest(p)$. The i^{th} component of $\vec{p}$ is denoted as $p[i]$. The velocity of p is denoted as $v(p)$, with the i^{th} component of the velocity, denoted as $v(p)[i]$.

Algorithm Particle Swarm Optimisation
Input The fitness function, f that takes m arguments and the size of the swarm n
Output A vector $\vec{p}$ encountered during the search where $f(\vec{p})$ had the best value.

```
PSO(f,n)
    Create a swarm S such that S = {p_1, p_2, ..., p_n}.
    For each p in S
        For 1 ≤ i ≤ m
            p[i] = r | r is a random value
            v(p)[i] = 0
        pbest(p) = p

1:  While not end of search
        For each p in S
2:          Update v(p)
            For 1 ≤ i ≤ m
                p[i] = p[i] + v(p)[i]
3:          If f(p) > f(pbest(p))
                pbest(p) = p
```

Figure 7.1: Outline of the PSO algorithm

7.2.1 Neighbourhood topologies

A *neighbourhood strategy* describes how the particles that belong to a neighbourhood are chosen. The owner of the neighbourhood is called the *subject particle*. The neighbourhood strategy constructs a neighbourhood network in which every subject particle is connected to its neighbours. The general structure (or pattern) used to construct this network is called the *neighbourhood topology*.

Kennedy and Mendes [34] identify three topologies: global best, local best and Von Neumann. The *global best topology* includes as the neighbour of a particle every other particle in the swarm. The *local best topology* forms a regular ring lattice by arranging all particles in a ring and selecting the k closest particles on either side of the subject particle as neighbours. The third is the *Von Neumann topology* that places particles in a matrix configuration. A two-dimensional lattice is formed by connecting each element in the first row to the aligning element in the last row; and connecting each element in the first column to the aligning element in the last column. From this lattice, the k particles above and below, as well as the k particles to the left and to the right of the subject particle, are included in the neighbourhood.

The topology effects the performance of the swarm because it determines which particles have an influence on the trajectory of the subject particle. The magnitude of this influence depends on the fitness function [32]; therefore, a topology that is superior in general cannot be isolated. However, an empirical study conducted across a variety of fitness functions indicates that the Von Neumann topology is best for most functions [34]. Using a fitness function that relate closely to the current work, Franken and Engelbrecht [25, 26] apply PSO to optimise the weights of a neural network; and also concludes that the Von Neumann topology is superior when compared with the local- and global best topologies.

The function of the neighbourhood network is to propagate information regarding the best locations found, to the other particles in the swarm. Watts [74] postulates that the ability of a network to propagate information is influenced by the number of edges in the network, the clustering (or cliquishness) of the network and the average shortest path length between nodes in the network. Kennedy and Mendes [34] note that the number of edges in the neighbourhood topology is directly proportional to the number of particles in the neighbourhood, and that the cliquishness can be measured as the average number of shared particles in the neighbourhood of a node. A particle $\vec{q}$ in the neighbourhood of $\vec{p}$ is counted as a *shared particle* if and only if $\vec{p}$ is also in the neighbourhood of $\vec{q}$.

A Von Neumann topology with the same neighbourhood size as a local best topology have the same number of edges. The difference is that the particle in the Von Neumann topology is shared in twice as many neighbourhoods as the particle in local best topology with the same size. Also, the average shortest path between Von Neumann particles is far less than the average shortest path of the local best configuration. Thus, the measurements suggested by Watts [74] predicts the empirical results accurately.

As an option, Kennedy and Mendes [34] include the subject particle in its own neighbourhood. A neighbourhood strategy that excludes the subject particle is called a *self-excluding strategy*. In a *self-including strategy*, the subject particle is included in its own neighbourhood. In a self-excluding strategy, the best particle in the neighbourhood also contributes to the space exploration, because it is pulled towards the second best particle. A self-including strategy pulls the best particle towards its own personal best, and consequently this particle contributes little to the exploratory search. Therefore, there is reason to expect a self-excluding strategy to do better.

For the fitness functions used in the Kennedy and Mendes comparative study [34], it is only for the local best topology that the self-excluding strategy performed better than the self-including strategy. For global best and Von Neumann the performance of self-including strategy outperformed the self-excluding strategy. For the global best, and the local best topologies, these results are predictable. In the global best case, the behaviour of only one particle will be affected by the inclusion (i.e. the global best particle). For local best, more particles are constrained, and self-excluding is expected to perform better. However, it is not clear why Von Neumann would show better results for the self-including strategy. For this reason, Section 7.4 presents the results of an experiment conducted to decide whether or not a self-including strategy must be employed for the game learning framework instead of a self-excluding strategy.

7.2.2 Particle trajectory

A particle's trajectory is influenced by its personal best (or *pbest*()) and the best of the personal bests of the particles in its *neighbourhood.* The latter is called the *local best* (or *lbest*()) [17]. These two poles, local best and personal best, to which a particle is attracted are fundamental to the hypothesis from which PSO originates. According to this hypothesis, the ability of a species to survive lies with its ability to share information by social interaction [33]. During this interaction, individuals in the species share information gained by personal experience. This

experience based information is sometimes referred to as cognitive information. In PSO, the cognitive information is represented by the personal best, and the social information is represented by the local best. The strong relationship between the personal best and the local best is highlighted by the following definition:

Definition 7.1: *Local Best.* For particle p with neighbourhood $\mathbf{N}(p)$, and maximising fitness function f, $lbest(p)$ is an arbitrary selection from the local best set $\mathbf{L}(p)$, where

$\mathbf{L}(p) = \{pbest(q) \mid q \in \mathbf{N}(p) \ \wedge \ \forall_{x \in \mathbf{N}(p)} f(pbest(x)) \leq f(pbest(q))$

The movement of a particle is controlled by the *velocity update function* (line 2 of the PSO outline on page 113). This function calculates the new velocity by adjusting each component of the velocity vector separately. Two constants regulate the magnitude of the velocity change: the *cognitive acceleration constant*, c_1, constrains the change towards the personal best, and the *social acceleration constant*, c_2, constrains the change towards the local best. In addition, Shi and Eberhart [59] introduced an *inertia weight*, w, that restricts the influence the current velocity has on the new velocity. Choosing different values for c_1, c_2 and w has a measurable effect on the performance of PSO [63, 18, 60].

Stochastic elements, r and s, are introduced to the velocity update function. These elements facilitate an uncontrolled exploration of the search space. The values are from the uniform distribution, $U(0, 1)$. The values of r and s influence the rate of change toward personal best and local best, respectively. Thus, the acceleration constants, c_1 and c_2, essentially provide upper bounds for the change in the velocity.

The state of particle p at epoch t can be denoted as p_t. Using this notation the velocity at epoch $t + 1$ is $v(p_{t+1})$. This value is determined from the state of the particle at the preceding epoch:

$$\begin{aligned} v(p_{t+1})[i] \quad = \quad & w \times v(p_t)[i] + \\ & r_t \times c_1(pbest(p_t)[i] - p_t[i]) + \\ & s_t \times c_2(lbest(p_t)[i] - p_t[i]) \end{aligned} \tag{7.1}$$

where $r_t, s_t \sim U(0, 1)$. Note that these random values are also time-dependent.

After the velocity has been determined, *velocity clamping* can be applied. For a domain, a maximum velocity value, v_{max} is determined. The velocity is restricted, component by component to the values in the range $[-v_{max}, v_{max}]$. The value of v_{max} is dependent on the domain, usually chosen as a fraction of the maximum vector in the search space x_{max}. That is, for $0.1 \leq k \leq 1.0$, $v_{max} = k \times x_{max}$. Note that this restriction is not placed on the value of $\vec{p}$, only on

the distance a particle can move in one iteration $v(p)$.

Clerc and Kennedy [13] define constraint equations that improve the probability of convergence. These equations are combined with Equation 7.1 to derive the *constricted function* . Using this function for the PSO also has the advantage that the population converges without placing a v_{max} limit on the velocity. The constricted function uses a *constriction factor*, χ:

$$\begin{aligned} v(p_{t+1})[i] &= \chi \times (v(p_t)[i] + \\ &\quad r_t \times c_1(pbest(p_t)[i] - p_t[i]) + \\ &\quad s_t \times c_2(lbest(p_t)[i] - p_t[i])) \end{aligned} \tag{7.2}$$

The value of χ is determined as follows:

$$\chi = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \tag{7.3}$$

where $\phi = c_1 + c_2$. Clerc and Kennedy [13] found that ϕ must be greater than 4.0 to promote convergence.

A domain specific enhancement is required for the PSO that optimises the evaluation function. The weights of the evaluation function (described in Section 3.5) indicates the importance of the weight relative to the other terms in the function. The consequence of this is that there are an infinite number of functions that are equivalent. PSO currently searches an infinite space; and it is possible that many particles with equivalent functions could find themselves in vastly different locations in the search space. In an attempt to mitigate this problem, *value clamping* has been introduced to make the search space smaller. Two constants, val_{min} and val_{max}, specifies the minimum and the maximum value of particle components, respectively. After the velocity has been applied to a component, the value of that component is adjusted to ensure conformance to these limits. If the value is less than val_{min}, it is set to val_{min}. Likewise, if it is greater than val_{max}, it is set to val_{max}. For the current domain, val_{min} must be greater or equal to 0. A negative weight would lead to a a negative value for the evaluation function. As explained in Section 3.5, the evaluation function has a value 0 to indicate losing positions, and a value of 1 to indicate winning positions. Also, the weights estimate the importance of a term relative to the other terms in the function. From this vantage point, a zero indicates that a term is unimportant, and a negative value is meaningless.

7.3 Competitive Environments

In a competitive environment, the fitness function evaluates the learning agent's performance against a set of opponents. The opponents for a learning agent can be drawn from two sources. The first source is a set of random players. As described in Section 5.6, these players can form the foundation for an objective performance measurement. However, this objective measurement requires extensive computing resources and can therefore not be applied as a PSO fitness function. The second source of opponents is the population of the agents created during the learning process. In the case of PSO, this population is the swarm of particles. The use of other particles in the swarm as opponents encourages an evolutionary development that increases the ability of each participant until a near optimal configuration is achieved [4].

A *competitive fitness function* is any calculation that is dependent on the current population to some degree [4]. In a PSO swarm that uses a fixed fitness function, the learning environment is static; but when it uses a competitive fitness function that depends on the fitness of the other particles in the swarm, the learning environment becomes dynamic. In a dynamic environment, PSO is stable and efficient; and in many cases, such an environment helps to avoid local optima and locate a global one [47].

Coevolution is a learning process in which the learning environment changes as the process continues [48]. More specifically, *competitive coevolution* requires that the fitness of an individual is determined by the fitness of other individuals in the environment [76]. Therefore, a PSO that employs a competitive fitness function is an example of competitive coevolution.

Competitive coevolution is a variation of self-play learning, and as such it suffers from two problems that are common to all training methods that employ self-play. Firstly, self-play is very likely to get stuck on a self-consistent but a non-optimal strategy [66]. Secondly, there is no guarantee that the portions of the strategy space searched are the most significant ones [20]. These problems lead to strategies that perform poorly.

The problems related to self-play learning are addressed by ensuring that the population diversity is adequate to avoid local minima and to cover a larger search space. A method to increase diversity is simply to increase the population size. However, Tesauro [66] illustrates that the learning of does not require a large population size. In the case of -, and with stochastic games in general, the source of the diversity is the stochastic element present in the game rules. This diversity is a primary contributor to the success of Tesauro's player [48].

Angeline [4] observes that the stochastic elements present in the GA population, such as the application of the mutation operator and the probabilistic selection of parents, bring about a level of non-determinism that can be exploited to create a population that is diverse enough for learning through self-play. A similar argument can be put forward for PSO: it also has stochastic elements, and as such it has the potential to maintain a level of diversity that is adequate for self-play learning. It follows that PSO provides a training environment that is likely to mitigate the problems associated with training methods that employ a competitive fitness function.

7.3.1 The Competitive PSO algorithm

The *competitive PSO algorithm* optimises a competitive fitness function using the particles in the PSO swarm as opponents. The first application of competitive PSO, described in Messerschmidt and Engelbrecht [40], is an analysis that compares PSO to the Evolutionary Program (EP) described in Fogel [22]. This analysis indicates that competitive PSO obtains better T - - players than those obtained by competitive EP. Franken and Engelbrecht [24] extended the work of Messerschmidt and Engelbrecht on T -T - by analysing the effect of different PSO structures and neural network topologies on the learning performance. This work was also extended to C [25, 26].

The competitive PSO extends the original PSO outlined in Figure 7.1 (page 113) by introducing a *competition stage* at the start of each epoch. During this competition stage, the fitness of the particles are determined using the competitive fitness function.

The competitive fitness function used by Messerschmidt and Engelbrecht selects a constant number of opponents randomly from the swarm for each participant. The participants include all the current locations as well as the personal best of each particle. For each participant a score is kept. At the start of the competition stage, all the scores are initialised to zero. During the competition, the score increments when the participant beats an opponent, and decrements when the participant loses. In the same manner, the opponent's score is also adjusted. The number of matches played depends on the swarm size, n and the number of opponents, k. The number of matches played during the competition stage is $2 \times n \times k$.

The Messerschmidt competitive function implements an evaluation method that is not fair and it could result in fitness values that are too coarse. The evaluation is not fair because, depending on the quality of the selected opponents, a good participant could be assigned a bad fitness value, and *vice versa*. The coarseness comes from the limited range of the fitness function

and it produces the situation that many participants share the same fitness value. Consequently, the local best of a particle becomes more often than not, an arbitrary choice between seemingly equal candidates in the neighbourhood.

A solution to this problem (of fairness and coarseness) is to allow the personal bests to compete in a tournament that aims to be a fair assessment of the participant's ability. In such a tournament, the fitness value is not a tally of match outcomes; it is a ranking. By using the results obtained by previously matched participants, the number of new matches conducted during the competition step to determine the ranking, is kept to an absolute minimum.

When ranking is used, particle p is considered better than particle q if and only if p has a higher rank than q. This comparison does not imply that there was a match between p and q. Indeed, it is possible that q will beat p, even if p is ranked higher. The fairness of the ranking depends on the tournament structure and the elimination strategy used during the competition. The next section provides some alternatives to consider in this regard.

7.3.2 Tournaments

In a tournament, a number of participants compete to decide which participant is the best. A tournament is one of three types: an *elimination tournament*, a *scoring tournament* or a *hybrid tournament*. In an elimination tournament, participants are removed from the tournament until the winner remains. In a scoring tournament a score is given to each competitor after a match, and the winner is the participant with the highest score. A hybrid tournament contains elimination and scoring stages.

Different types of scoring tournaments are available. The tournament used in Messerschmidt and Engelbrecht [40] and in Fogel [22] is a *random subset tournament*. In a *fixed subset tournament*, the set of competitors for a participant does not change every epoch. In a *round robin tournament*, every participant is matched against every other participant. Because of the high likelihood that more than one participant will have the same score, a *tie breaking procedure* is required to identify the winner. During this procedure, criteria derived from the tournament can be considered, such as: the results of the matches between winning participants; the ratio of number of wins against the number of losses; the difference between the number of wins and number of losses; and simply the number of wins.

Two types of elimination tournaments are common. In a *knockout tournament*, the loser of a match is eliminated, and the winner continues to the next round. This elimination process

continues until one participant remains. In a *double-elimination tournament*, the participant is eliminated after he loses a second time. This second chance is implemented by creating two brackets: a winners bracket, and a losers bracket. The process followed in the winners bracket is the same as the knockout tournament. However, when a participant in this bracket loses a match; the participant is moved to the losers bracket. The losers bracket has two stages to every round: firstly the winners of the previous losers bracket round (or the losers of the very first winners bracket round) compete. In the second stage, the winners of the first stage compete against the losers of the same round in the winners bracket. The losers of the second stage are eliminated from the tournament, and the winners remain in the losers bracket. This process continues until both brackets have only one remaining participant. The champion of the losers bracket would have lost one match, and the champion of the winners bracket is undefeated. These two champions then compete to determine the winner of the tournament.

In the elimination tournaments, the *pairing procedure* decides which participants should compete. If unlucky, the second best participant can be paired with the best participant and eliminated at the first round of the tournament. Although the double-elimination tournament mitigates the problem, unfair pairing remains an issue. The simplest pairing procedure is to select the opponents at random. However, if the ability of the participants are known, the pairs can be organised such that the best players are likely to compete in the final rounds. A process called *seeding* orders the participants according to previous performance from best to worst. From this sequence, pairs are formed by repeatedly removing the first and the last seeded participant. If the number of participants in a round is not even, one participant receives a *bye*, and moves to the next round without competing.

7.3.3 The Tournament PSO algorithm

A characteristic of PSO that is not available to standard GA optimisation algorithms is that PSO has memory. Every PSO particle ‘remembers’ its own personal best location. Typically, an individual in a standard GA population is replaced in every generation with a new individual. Therefore, every new individual must be re-evaluated for every GA generation. However PSO does not need to follow suit: using the personal best, it is possible to avoid the re-evaluation of every particle in the swarm at the start of each epoch. If the re-evaluation does not apply to each particle, it becomes feasible to use the more resource intensive, but more fair tournament methods described in the previous section. It is from this idea that the *Tournament PSO* is

developed in this thesis.

The key difference between *Tournament PSO* and the competitive PSO described in Section 7.3.1, is that the particles in tournament PSO compete against their own personal best. Tournament PSO elaborates on competitive PSO by splitting the competition stage into two smaller stages: the *personal competition stage* and the *tournament competition stage*. In the personal competition stage each particle's current location competes against its own personal best. If the personal best is beaten, it is replaced with the current location. During the tournament competition stage, the personal bests of all the particles compete in a tournament. The tournament establishes a partial order ranking that is used to determine the local best particle in the neighbourhood.

The first stage of Tournament PSO requires a constant number of matches: for a swarm size of n, n matches are held during the personal competition stage. If, for a given epoch, no new personal bests are identified, the results of competition phase for the previous epoch is used again to determine the fitness of the particles in the swarm. The number of matches held during the tournament competition stage depends on the tournament method.

Tournament PSO also makes use of a *match cache*. The match cache is a memory structure used during the tournament competition phase. Although this memory structure increases the memory requirement of the PSO, it does not affect the trajectory of any particle. Its purpose is to avoid needless matches between particles. The match cache keeps the result of the match between the pairs of personal best locations. When the tournament demands a match between two locations that have already been matched, the result is retrieved from match cache. If the result is not available in the cache, the particles compete, and the result of the competition is added to the cache. Whenever a new personal best is identified, all matches in which the personal personal best participates are removed from the match cache. The number of new personal bests are likely to become fewer as the particles approach optimal configurations, and more matches will be replaced by cached results. The net effect is that more computing resources are spent on search space exploration, and less on the evaluation of personal bests.

The self-excluding Von Neumann neighbourhood topology is chosen to identify neighbours. If more than one neighbour has the same ranking, either the neighbours are chosen randomly, or a knockout tournament is used to select the local best. These two approaches are called *random best* and *knockout best* respectively.

A tournament match between two participants consists of two games; each participant gets

a turn as the first player. For these games, 2 points are awarded to the winner, 1 point each for a draw and 0 points to the loser. The participant with the most number of points after the two games wins the match. If the score is equal, the winner is chosen at random.

The local match is more strict - in order to replace the personal best, the current particle location must win as first and as second player. This sterner rule aims to prevent situations in which a tested champion is replaced with a 'lucky' novice [48].

If a tournament requires pairing, either *random pairing* or *rank seeding* can be employed. In random pairing, opponents are paired by random selection. In rank seeding, the previous rank of the particle is used to order the participants. For a particle that obtained a new personal best, the rank obtained by the previous personal best is used for seeding. From this ordered list, pairs are chosen such that the best seeds are likely to compete in the final rounds.

For the tie breaker procedure, a knockout tournament is held that includes all the winners as competitors. If there are still ties, a winner is chosen randomly.

The tournaments (subset and elimination) can be combined with the random or knockout local best. For elimination tournaments, either random pairing or seeding can be applied. These choices lead to fourteen different permutations, each permutation is a different *tournament method* that can be used in the Tournament PSO. These methods are summarised and labelled in Table 7.1. and the constricted function (Equation 7.2 on page 117) is used to update

Tournament	**Random best**		**Knockout best**	
Random subset	RSR		RSK	
Fixed subset	FSR		FSK	
Round robin	RRR		RRK	
	Random pairing	**Seeded**	**Random pairing**	**Seeded**
Knockout	KRR	KSR	KRK	KSK
Double elimination	DRR	DSR	DRK	DSK

Table 7.1: The tournament method matrix

the velocity.

7.4 Experiment: Self in neighbourhood

Objective

For the learning framework, either a self-excluding or a self-including strategy must be used. This experiment investigates the influence this choice has on the Tournament PSO. Every tournament strategy defined in Section 7.3.3 is used to gain a general result.

Method

Because of the decision made in the previous chapter, a phased evaluation function built from the DS strategy (exclude dubious and cut spurious branches) is used for this experiment. From the 30 DS functions created by the experiment in Section 6.6, the best performing function has been selected as the function to optimise for this experiment. Call this function F_b.

For the floating-point PSO parameters the following values were chosen: $w = 1$, $c_1 = 2.1$, $c_2 = 2.1$, $\text{val}_{min} = 1$ and $\text{val}_{max} = 99$. The values for c_1 and c_2 were chosen to adhere to Equation 7.3. The swarm consisted of 25 particles arranged in a 5×5 lattice. The smallest Von Neumann topology was used, containing the 4 particles that surround the subject particle in the lattice.

The initial locations of the particles were set randomly. The level of consistency for the experimental conditions was improved by keeping the initial particle locations the same for every PSO run.

The PSO was terminated when 30 000 matches were played. During the tournament competition phase, the match count is incremented only when a new match was added to the match cache. The matches were played without searching the game tree. The move choice fell on the next position in the play line with the highest evaluation. In the case where more than one move with the best evaluation were encountered, the next move would be randomly chosen from amongst these moves.

Using the Tournament PSO algorithm, each of the fourteen tournament methods were used in twenty runs. Of these runs, ten included the subject particle in the neighbourhood, and ten excluded the subject particle from the neighbourhood. At the end of each run, the performance of the best particle in the swarm was measured. For this measurement Equation 5.7 on page 84 was used with a game count of 15 000.

In order to determine which of the two strategies is best, the mean performance of the strategies are compared for every tournament method. Because of the small size of the sample, it cannot be assumed that the sample variance is an adequate estimate of the population variance; and consequently the usual approach to compare means using confidence intervals cannot be applied. For such small sample sizes, the Student's *t*-distribution must be used to obtain a reliable comparison [71].

More specifically, the comparison of two sample means can be done using the *two-sample*

t-test. The formula for this test is:

$$t_{n_1+n_2-2} = \frac{\overline{X}_1 - \overline{X}_2 - (\mu_1 - \mu_2)}{s \times \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{7.4}$$

where n_1 and n_2 are the sample sizes, $\overline{X}_1$ and $\overline{X}_2$ are the sample means, the values of μ_1 and μ_2 are determined by the test hypothesis, and s is the pooled standard deviation of the sample sizes.

The pooled variance is obtained from the following formula:

$$s^2 = \frac{s_1^2 \times (n_1 - 1) + s_2^2 \times (n_2 - 1)}{n_1 + n_2 - 2} \tag{7.5}$$

where s_1^2 is the variance of the sample with size n_1 and s_2^2 is the variance of the sample with size n_2.

The value $n_1 + n_2 - 2$ is called the *degrees of freedom.* From this value, the cut-off t-value is obtained from the t-distribution. If the observed t-value (that is Equation 7.4) falls in the reject region (determined from the cut-off value) the test hypothesis is rejected.

Unfortunately, matters are complicated by the fact that the two-sample t-test assumes that the variances of the two populations are equal. If they are not equal, it is necessary to use the *approximate t-test.* The approximate t-test does not use a pooled variance, and adjusts the degrees of freedom. The approximate t-test is given by the following formula:

$$t^* = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{7.6}$$

The degrees of freedom used to get the cut-off value of t^* from the t-distribution is given by:

$$n^* = \left\{ \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{\frac{(s_1^2/n_1)^2}{n_1+1} + \frac{(s_2^2/n_2)^2}{n_2+1}} \right\} - 2 \tag{7.7}$$

In order to determine whether of not the two variances are equal, the F-distribution is used. The formula to determine the F-value is simply:

$$F = \frac{s_1^2}{s_2^2} \tag{7.8}$$

The F-distribution has two degrees of freedom, $n_1 - 1$ and $n_2 - 1$. Like the t-test, these degrees of

freedom are used to determine the cut-off value for the F-test. If the observed *F*-value (Equation 7.8) exceeds the cut-off value, the hypothesis that the two variances are equal is rejected. In order for this one-sided test to work, s_1^2 must be substituted with greatest variance, and s_2^2 with the other variance.

In order to use the *F*-distribution or the *t*-distribution it is assumed that the data is from the normal distribution. The *central limit theorem* [71] states that if a random variable is the sum of a large number of random increments, then it has the normal distribution. The performance measurement is an accumulation of 15 000 smaller measurements, and as such it adheres to this theorem, and can therefore be assumed to be a variable with a normal distribution.

This concludes the overview of statistical theory, now the equations above can be simplified given the circumstances of the current experiment. Clearly, the first problem is to determine the *F*-values using Equation 7.8, and compare that to the *F*-distribution cut-off. For a 5% significance level, this cut-off value for two variances obtained from 10 samples each is as follows:

$$F_{n_1-1,n_2-1}^{(0.025)} = F_{9,9}^{(0.025)} = 4.0260 \tag{7.9}$$

If the observed *F*-value obtained from Equation 7.8 exceeds the value of Equation 7.9 the variances are assumed to be different. In this case, the approximate degrees of freedom, n^* must be calculated using Equation 7.7 and cut-off value must be compared with the observed approximate t-value. Substituting the appropriate values in Equation 7.7 gives:

$$n^* = \left\{ \frac{(s_1^2/10 + s_2^2/10)^2}{\frac{(s_1^2/10)^2}{11} + \frac{(s_2^2/10)^2}{11}} \right\} - 2 \tag{7.10}$$

On the other hand, if the observed *F*-value does not exceed the cut-off value, the variances are assumed to be equal, and the *t*-distribution can be applied. The interest is to show that the two means are equal; that is $\mu_1 = \mu 2$, and consequently $\mu_1 - \mu_2$ in equation Equation 7.4 is 0. Further substitution simplifies this equation to:

$$t_{10+10-2} = t_{18} = \frac{\overline{X}_1 - \overline{X}_2}{s \times \sqrt{0.2}} \tag{7.11}$$

where *s* is the standard deviation. This value comes from the pooled variance calculated from Equation 7.5:

$$s^2 = \frac{s_1^2 \times 9 + s_2^2 \times 9}{18} = \frac{9 \times (s_1^2 + s_2^2)}{18} = \frac{(s_1^2 + s_2^2)}{2} \tag{7.12}$$

The final detail is the cut-off value used to determine the result of the two sample test. For a significance level of 5%, the cut-off value is:

$$t^{(0.025)}_{10+10-2} = t^{(0.025)}_{18} = 2.101 \tag{7.13}$$

Results

The performance statistics for the self-excluding and the self-including strategy are shown in Table 7.3 and Table 7.2 respectively. These tables show the sample mean, sample variance, minimum and maximum for each run.

S	$\overline{X}_S$	s^2_S	Max	Min
DRK	65.5266	2.2152	67.0433	62.6933
DRR	64.9607	3.6231	67.8300	61.5700
DSK	65.1237	2.0082	69.0767	64.2767
DSR	64.9407	0.8509	65.8167	62.5000
FSK	66.1740	1.6380	67.8967	64.1333
FSR	65.7873	1.9372	67.5900	64.1133
KRK	65.5377	1.3065	67.6200	63.8800
KRR	66.1990	2.4128	68.4833	63.2367
KSK	64.9130	0.2899	65.5967	63.5433
KSR	65.3693	0.3255	66.4967	64.7000
RRK	65.6947	3.1974	68.9167	63.6767
RRR	65.6273	1.5950	67.4167	63.5500
RSK	65.9980	2.6762	68.0833	63.7233
RSR	65.4710	2.7587	67.4300	62.2733

Table 7.2: Statistics for self-including stategies for F_b

S	$\overline{X}_S$	s^2_S	Max	Min
DRK	65.4016	1.1772	66.9133	63.3167
DRR	65.0917	2.6561	67.4033	62.2300
DSK	66.7290	2.5015	69.3867	64.5700
DSR	65.4183	1.7178	67.6000	63.0700
FSK	65.9664	1.4284	68.1233	64.0267
FSR	65.2980	2.7477	68.2867	63.4667
KRK	65.7903	3.4113	68.9633	62.7800
KRR	66.0460	1.5852	68.5467	64.5333
KSK	64.9390	0.6566	66.9633	64.2367
KSR	66.2663	2.9189	68.4400	63.4600
RRK	65.5190	3.5560	68.4367	63.3900
RRR	66.1187	3.7925	68.7533	63.7033
RSK	65.9110	2.3872	67.6500	62.5667
RSR	65.5840	1.5597	67.8100	63.5333

Table 7.3: Statistics for self-excluding strategies for F_b

Of the 14 methods, 10 produced a maximum for the self-excluding strategy that is greater

than the maximum for the corresponding self-including strategy. The mean of the maximums for the self-excluding strategies is 68.09119, compared to a 67.5212 mean for the maximums of self-including strategies. Thus, the maximum performance values indicate self-excluding is better. Insofar the minimums are concerned, only 8 methods produced a greater minimum for the self-excluding strategy than the minimum obtained for the corresponding self-including strategy. The mean of the minimums for the self-excluding strategies and self-including strategies is 63.49167 and 63.41928, respectively. Although, the contest is much closer than the maximums, the minimums also indicate that self-excluding is better.

The result of the calculations for the formal comparison of the observed measurements are shown in Table 7.4. Except for KSR all the tournament methods produced an *F*-value well below the cut-off value specified in Equation 7.9. Consequently, except for KSR, the two-sample test applies to all tournament methods. For all these, the *t*-value have been computed using Equation 7.12 and Equation 7.11. Almost all the calculated *t*-values were in the acceptance region, $-2.101 < t_{18} < 2.101$ and the conclusion is that most means are equal. The only tournament method that shows a better mean for the self-excluding strategy is DSK.

Table 7.4: The computed values for the self-including and self-excluding mean comparison for F_b

	Self-including		Self-excluding				
Tourn.	Mean	Variance	Mean	Variance	*F*-value	*t*-value	Comparison
DRK	65.5266	2.2152	65.4016	1.1772	1.8817	0.2146	Equal
DRR	64.9607	3.6231	65.0917	2.6561	1.3640	0.1653	Equal
DSK	65.1237	2.0082	66.7290	2.5015	1.2456	2.3905	Excluding is better
DSR	64.9407	0.8509	65.4183	1.7178	2.0188	0.9425	Equal
FSK	66.1740	1.6380	65.9664	1.4284	1.1467	0.3750	Equal
FSR	65.7873	1.9372	65.2980	2.7477	1.4184	0.7149	Equal
KRK	65.5377	1.3065	65.7903	3.4113	2.6110	0.3678	Equal
KRR	66.1990	2.4128	66.0460	1.5852	1.5221	0.2420	Equal
KSK	64.9130	0.2899	64.9390	0.6566	2.2652	0.08458	Equal
KSR	65.3693	0.3255	66.2663	2.9189	8.9671	*NA*	Equal (via t^*-test)
RRK	65.6947	3.1974	65.5190	3.5560	1.1122	0.2138	Equal
RRR	65.6273	1.5950	66.1187	3.7925	2.3777	0.6694	Equal
RSK	65.9980	2.6762	65.9110	2.3872	1.1210	0.1223	Equal
RSR	65.4710	2.7587	65.5840	1.5597	1.7688	0.1720	Equal

For the KSR tournament method $n^* = 11.42329$ and $t^* = 1.57479774$. Using the nearest integer value gives the cut-off value for the approximate t-test at a significance level of 5% as:

$$t_{11}^{(0.025)} = 2.201 \tag{7.14}$$

The measured value, 1.57479774, is below the cut-off, and therefore the hypothesis that the two means of KSR are equal is accepted.

Although the minimum and maximum measurements hint toward the conclusion that self-excluding strategies are better, this conclusion would not be statistically sound. A more accurate general conclusion is that the performance of a specific Tournament PSO that uses the self-excluding strategy is no worse than the performance of a PSO that is the same as the specified PSO in other respects but uses a self-including strategy. However, a specific conclusion can be stated for the DSK tournament method – for this method, the self-excluding strategy outperforms the self-including strategy.

7.5 Experiment: Tournament method performance comparison

Objective

This experiment applies the self-excluding strategy to each of the tournament methods identified in Section 7.3.3 and it compares the performance of the optimised playing agents.

Method

The method described in Section 7.4 is also used for this experiment. The subsection below provides a comparative analysis of the statistics obtained from the self-excluding runs conducted during the previous experiment.

Results

The results for F_b is shown in Table 7.3, and depicted as a box-and-whisker diagram in Figure 7.2. Taken as a whole, the mean performance of all tournament methods is 65.71995. In the experiment described in Section 6.6, the performance of F_b with all weights equal to 1.0 was measured as 67.6767.

Only DSK, KSR, RRK and RRR have an upper quartile higher than 67.6767. Some of the strategies were not able to reach this value even as a maximum. From the Figure 7.2, it is clear that no single strategy can be isolated as the best strategy, but it seems very plausible that KSK is the worst tournament method for Tournament PSO.

In order to identify the better strategies accurately, the mean value of every strategy is tested for equality against the mean value of the strategy that obtained the greatest mean value. The

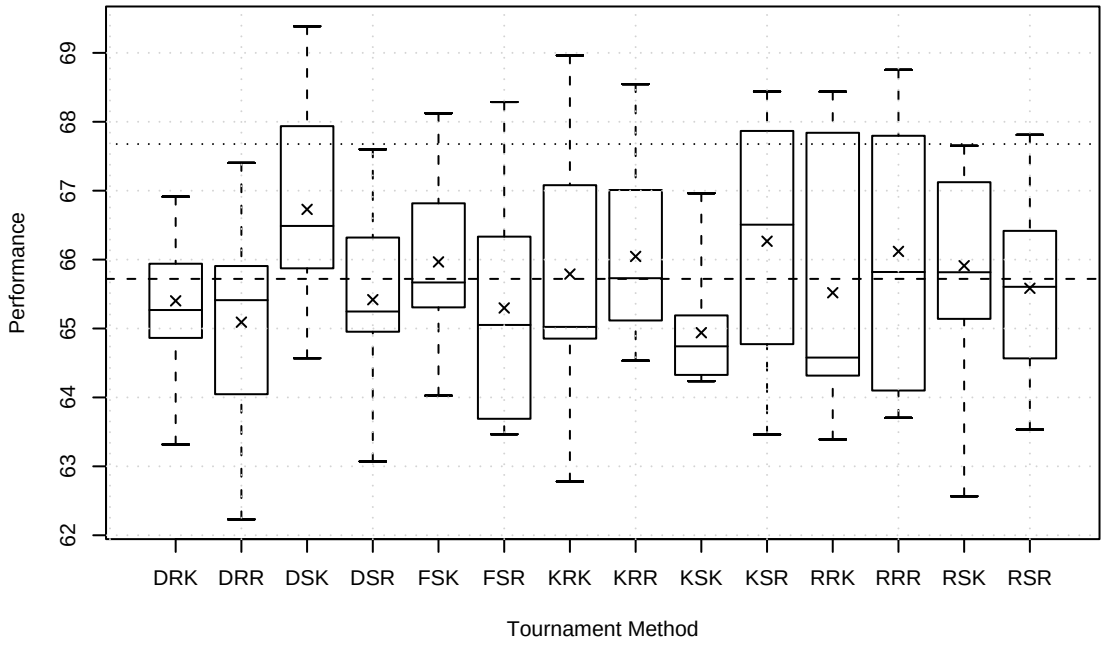


Figure 7.2: The performance of the tournament methods

best performer was DSK with a mean value of 66.7290. As before, the *F*-value and the *t*-value are used (see page 125) to determine the equality of the two means. The results of these calculations are shown in Table 7.5.

Table 7.5: Comparing other method means with the mean of DSK

	Mean ($\overline{X}_S$)	**Variance (s_S^2)**	**F-value**	**t-value**	**Result**
DRK	65.4016	1.1772	2.1249	2.1885	Not equal
DRR	65.0917	2.6561	1.0618	2.2799	Not equal
DSK	66.7290	2.5015	1.0000	0.0000	Equal
DSR	65.4183	1.7178	1.4562	2.0178	Not equal
FSK	65.9664	1.4284	1.7512	1.2166	Equal
FSR	65.2980	2.7477	1.0984	1.9751	Not equal
KRK	65.7903	3.4113	1.3637	1.2207	Equal
KRR	66.0460	1.5852	1.5780	1.0684	Equal
KSK	64.9390	0.6566	3.8100	3.1852	Not equal
KSR	66.2663	2.9189	1.1669	0.6284	Equal
RRK	65.5190	3.5560	1.4216	1.5547	Equal
RRR	66.1187	3.7925	1.5161	0.7693	Equal
RSK	65.9110	2.3872	1.0479	1.1699	Equal
RSR	65.5840	1.5597	1.6038	1.7967	Not equal

Only six of the fourteen tournament methods were found to be less effective than DSK.

Table 7.6 emphasises the stronger tournament methods by marking the places of the six weaker

Table 7.6: The dominant tournament method matrix

Tournament	**Random best**		**Knockout best**	
Random subset	***		RSK	
Fixed subset	***		FSK	
Round robin	RRR		RRK	
	Random pairing	**Seeded**	**Random pairing**	**Seeded**
Knockout	KRR	KSR	KRK	***
Double elimination	***	***	***	DSK

methods with asterisks ('***').

The first deduction that can be made from these results is a predictable one: random best is weaker than knockout best because the latter was used by only two of the six eliminated methods. The second deduction is that double elimination is less effective than knockout tournaments. Double-elimination takes more rounds to determine the winner, and more epochs can be achieved with the same number of games using the knockout tournaments. It is therefore reasonable that knockout could be better. However, DSK is a double-elimination strategy that did very well. The reason for this could be that DSK also used the two other aspects that intuitively introduce more fairness: that is the use of seeding for tournaments and the use of a knockout tournament to determine the best of equal particles in the neighbourhood.

7.6 Experiment: Tournament method interval analysis

Objective

In the previous experiment, eight tournament methods where isolated according to their performance. This experiment aims to identify the better methods among these by considering the evaluation functions created at various intervals during the Tournament PSO runs.

Method

In this experiment, the Tournament PSO was run for ten times for the following tournament methods: KSR, RRR, RRK, FSK, KRK, KRR and RSK. The parameters for these runs did not deviate from the runs described in Section 7.4. During each run, 31 evaluation functions were written to files: the initial function and 30 functions taken from the swarm at a 1 000 match intervals. The initial function is taken right after the first tournament. Every time a function was taken the current swarm champion was chosen. In total 2 400 interval evaluation functions were

produced. For each one of these functions, a performance measurement was obtained using 15 000 matches. The performance function described in Section 5.6 was used to obtain the measurements.

For the analysis the measurements were aggregated into interval means. The interval mean is the mean of the ten measurements taken at an interval for a tournament method. Thus, for each tournament 31 interval means were obtained.

A formal statistical analysis of the interval means is not done for this experiment. The reason of this is that the previous experiment concluded that the performance of the final interval mean is statistically equal. There is no reason to expect the sample variances of the other intervals to be less than the variances of the final intervals. Therefore, a statistical analysis is likely to conclude that all the interval means are equal. The results of this experiment undergoes a less formal, but still a reasonable analysis with two steps.

The first analysis is simply a count of the number of interval means that is greater than the initial interval mean for that method. If this count is thirty, it means that the method constantly produces a function that performs better than the initial function. However, a count less than 15 indicates that the method is likely to produce functions that perform worse than the initial function. In such case, the tournament method fails to identify and to promote better functions. It could also indicate that the method is susceptible to local minima.

The second part of the analysis compares the interval means of each tournament at the intervals. If one of the methods consistently produces a mean greater than the other methods, it is likely that this method is better than the others.

Results

Figure 7.3 shows the number of interval means that are greater than the mean of the performance of the initial evaluation function. Only DSK and KSR have the maximum count of thirty. KRK and FSK have a count lower than fifteen. The conclusion is that these two methods fail to produce functions that consistently preform better as the PSO search continues.

The second analysis excludes KRK and FSK. Figure 7.4 shows the interval means for each of the six remaining tournament methods. The greatest values are amongst the means of the following methods: KSR, DSK, KRR, RSK and RRR. The only tournament method that has no interval with the greatest mean is RRK. The methods that has intervals with the smallest mean are restricted to RRK, RSK and KRR.

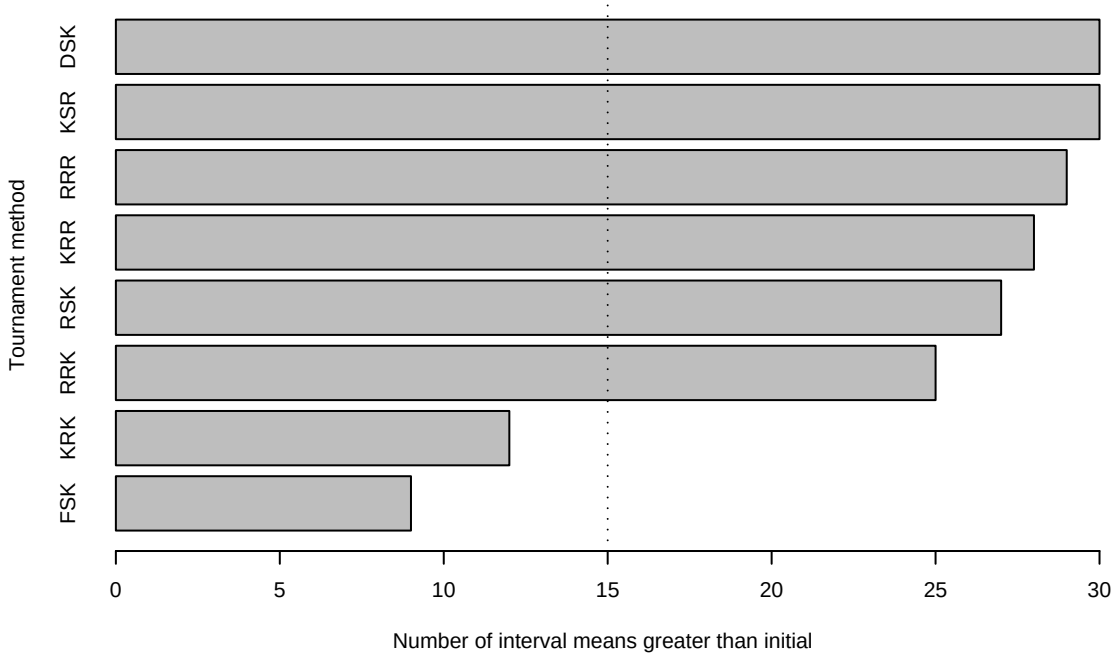


Figure 7.3: Tournament method interval performance

From Figure 7.4 the dominance of DSK as the maximum mean for each interval is clear. Here follows a the count of intervals for which each strategy achieved the maximum interval mean: KRR = 1, RSK = 2, KSR = 6, RRR = 4 and finally DSK had 17 of the maximum intervals.

For DSK all the intervals produced a better performing function than the initial function, and DSK had the best performing function in more than half of the measured intervals. Thus according to this experiment, DSK performs better than the other tournament methods.

7.7 Conclusion

PSO is an algorithm that finds optimal values of the arguments of real-valued functions using a multi-location search. The velocity of a location, or particle, drives it through the search space. This velocity is influenced by the neighbours of a particle. These neighbours are selected from other particles in the swarm using one of three typical neighbourhood topologies. Of these, the Von Neumann topology has been chosen as the topology to be used in the learning framework. The constricted function has been chosen as the update function guarantees convergence, and has been chosen as the function to update the velocities of the particles that optimise the weights

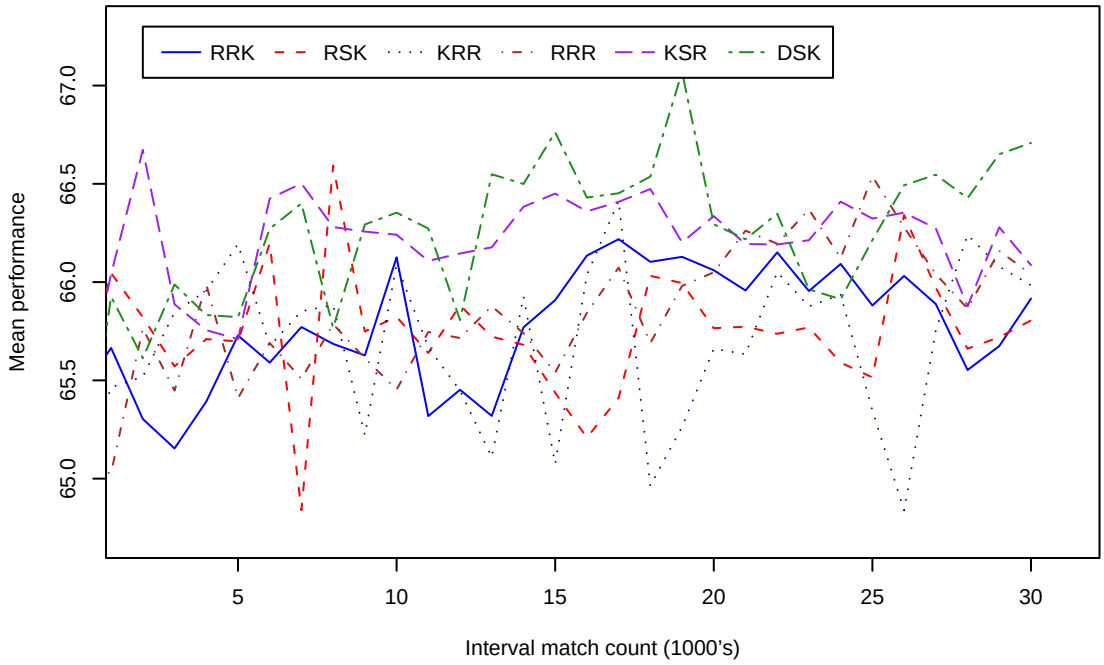


Figure 7.4: Tournament method convergence

of the evaluation function.

In a competitive environment, a static fitness function is likely to prove impractical. A competitive fitness function uses the other particles in the swarm to determine a particle's fitness. The use of this type of competition during the optimisation process creates a co-evolutionary environment in which a particle gains strength by playing against the rest of the swarm. The Competitive PSO selects opponents randomly from the swarm.

The Tournament PSO introduces the idea of ranking all particles according to a tournament method, such that the fitness value of the particle is its ranking. In addition, this new PSO redefines the personal best: if the current location beats the personal best in a match, it becomes the new personal best.

The new PSO brings with it the question of which Tournament method is better. Experimental results indicate that the most fair and least coarse method does perform better; and it is less likely to converge early. This method, labelled `DSK` uses the double-elimination tournament with seeding, and the knockout tournament to select the local best amongst equals.

In the next chapter, the `DSK` strategy is incorporated with the conclusions from the previous chapter to create a C player that learns from zero knowledge. Additional aspects, such

as game-tree search and increasing the swarm size are also considered.

Chapter 8

The automated learning of CHECKERS

In the three chapters that precede this one, the learning framework and its detail has been described. This chapter is an empirical investigation into some aspects of this framework. The game of Checkers is used as the subject of this investigation. In addition, the learning performance of the framework when applied to a simpler game is measured.

8.1 Introduction

This short chapter presents the findings of a few experiments conducted with the learning framework. These experiments do not constitute a full experimental analysis of the framework. Rather, the experiments were devised to investigate some aspects of the framework that were deemed likely to effect the performance of the learning agent.

First, Section 8.2 describes the rules of C used for the learning experiments. Section 8.3 considers the idea of using examples in more than one phase when discovering new evaluation functions. Section 8.4 explores the discovery process further by investigating whether more complex evaluation functions lead to better learning. Section 8.5 assesses the influence of different schemes to produce the example play-lines. Section 8.6 measures the learning performance for a game that is less complex than C . Finally, Section 8.7 concludes this chapter.

8.2 The rules of CHECKERS

This section lists the rules used by the playing agent. These rules are based on the standard rules of C [39]:

* *Checkerboard.* The board is a square that is divided into 64 small squares arranged on an 8×8 grid. The squares alternate between a light and a dark colour. The game is played on the dark squares. The board is placed such that each player has a light square on the far left.

* *Initial position.* The playing pieces are Red and White. Each player has 12 pieces of the same colour placed on the 12 dark squares closest to his edge of the board. The player with the Red pieces makes the first move.

* *Moving pieces.* A *checker* is a piece placed on the board at the starting position. A checker can move one square diagonally forward to a vacant square. When a man reaches the last row, called the *King Row*, it becomes a *king.* This process is called *crowning.* A king can move forward as well as backward.

* *Objective.* The game is lost by the first player that cannot move, either because he has no pieces or because all of his pieces are blocked. At any time a player can resign (and lose), or both players can agree to end the game in a draw. For the current research, a limit of 100 moves were placed on every game. When the move count reaches 100, the game ends in a draw.

* *Capture.* A opponent's piece can be captured by diagonally jumping over it to an adjacent vacant square. A checker captures forward only, while a king is allowed to capture backwards. Several pieces can be captured in one turn using the same piece. Captured pieces are removed from play. When a checker is crowned the move ends - even if the new king can continue a capture. When there is an opportunity to capture, the active player is forced to do the capture.

8.3 Experiment: Overlapping game phases

Objective

The first step of the discovery stage (introduced on page 96) of the learning framework organises example positions into phase sets. Every phase set contains all the board positions from the examples that are in the given phase. In this approach, a position belongs to one and only one phase set. The phase sets proposed by Lee and Mahajan [38] also include the examples from positions in the phases that are adjacent to the phase that identifies the set. Lee and Mahajan used this approach in which phases overlap to train an O player. However, they did not

show empirically that phase overlapping is an improvement over non-overlapping game phases. The purpose of this experiment is to ascertain whether such an overlap is advantageous for the C learner.

Method

The cardinality of each phase set is determined by the total number of play-lines used to generate the examples. More play-lines produce larger phase sets. Large phase sets may lead to a better classification, but there is a trade-off. Larger sets produce larger decision trees that lead to evaluation functions that contain more terms. A function with more terms is harder to optimise because the number of dimensions is greater than the number of dimensions of a function with less terms. Also, larger functions consume more computational resources to evaluated game positions, ultimately leading to a slower learning rate.

During the discovery stage, an on-line pruning cut-off value is supplied to C4.5 to specify the minimum number of examples the branches of a sub-tree should have. If a sub-tree with less examples than this minimum is reached, the classification process terminates. Thus, when fixing the cut-off value, this pruning strategy produces larger decision trees when it is provided with a larger example set.

The experiments conducted in Chapter 6 used a C4.5 cut-off value of 32 for 1000 play-lines. Using the same cut-off to play-line ratio, this experiment uses a cut-off value of 144 for 4500 example play-lines. As an alternative configuration, the experiment also uses a cut-off value 32 for 4500 play-lines to see whether more complex evaluation functions lead to better players. A comparison of the performance at these two cut-off values provides some insight into the effect of the evaluation function complexity on the learning performance.

In order to investigate the effect of phase overlapping, both configurations (144 and 32 cut-off) are subjected to three different overlaps: 0, 1 and 2. The 0 overlap corresponds to the original algorithm. An overlap of 1 places an example position in the phase sets next to the phase of the position. In an overlap of 2, an example at phase η is placed in the following phase sets: $\{\eta-2, \eta-1, \eta, \eta+1, \eta+2\}$.

A play-line count that creates a phase set size of n for an overlap value of 0, leads to a phase set size of $3\times n$ when the overlap count is 1, and to $5\times n$ when the overlap count is 2. In order to ensure that the same number of examples are available for classification, the experiment uses a play-line count of 4500 for the 0 overlap, a 1500 play-line count for a 1 overlap and a 900 count

for a 2 overlap.

For each overlap and cut-off combination, the learning process was run 30 times. Each run started with a play stage that produced the number of play-lines associated with the overlap. These play-lines and the cut-off value were used during the discover stage to deduce the evaluation function. The DS pruning strategy was employed for this deduction. Then, this new evaluation function was optimised with the Tournament PSO using the DSK tournament method. The PSO was run until 10000 matches were held, and a swarm size of 49 was used. Other PSO parameters were $w = 1.0$, $c_1 = c_2 = 2.1$ and $v_{max} = \infty$. No tree searching was used for this experiment. The performance of the optimised functions were determined by Equation 5.7 (page 84) with 15000 games.

In order to compare the results, the 95% confidence interval is applied to an observed mean value, $\overline{X}$ using the observed standard deviation, σ. This interval is calculated as follows,

$$(\overline{X} - 1.96 \times \frac{\sigma}{\sqrt{30}}, \overline{X} + 1.96 \times \frac{\sigma}{\sqrt{30}}) \tag{8.1}$$

Results

Table 8.1 shows the results of the experiment. For both C4.5 cut-off values, the results show a decrease in performance as the overlap increases. However, the high variance makes it impossible to conclude that this trend is an accurate assessment. Nonetheless, the conclusion is that phase overlapping does not improve the performance of the learning framework.

8.4 Experiment: Function complexity

Objective

The number of terms in an evaluation function determines its complexity. The ideal complexity of a function balances classification accuracy with optimisation. A more complex function improves the ability of a function to discern between winning and loosing positions. However,

Table 8.1: Learning with overlapped game phases

Overlap	Cut-off = 32	Cut-off = 144
0	64.31774 ± 1.084966	59.40155 ± 1.498132
1	63.63928 ± 1.240780	59.29722 ± 1.672348
2	63.22548 ± 1.486220	57.93188 ± 1.702703

more terms introduce more weights that are more complex to optimise. Simpler evaluation functions are also easier to understand and faster to evaluate during play.

In the previous experiment, a notable performance increase was measured for the more complex function. This experiment compares the learning performance of the following C4.5 cut-off values: 32, 64, 96 and 144.

Method

For each cut-off value 30 learning runs were executed. Each run started with a play-stage that produced 4500 play-lines. Using no phase overlapping, the discovery algorithm was applied to the play-lines. The discovered evaluation was optimised using a match count of 10000. The PSO did not use game tree searching and a swarm size of 49 was used. Similar to the previous experiment, the DS and DSK strategies were used, the PSO parameters were $w = 1.0$, $c_1 = c_2 = 2.1$ and $v_{max} = \infty$, and Equation 5.7 (page 84) with 15000 games was used to measure the performance.

The statistical parameters are the same as the previous experiment, therefore Equation 8.1 is also applied to the results obtained from this experiment.

Results

Table 8.2 shows the measurements taken for this experiment. A cut-off value of 64 produce slightly better results than 32. Although the improvement is not statistically significant, a value of 64 is preferred because it produces less complex evaluation functions that evaluate faster. A value of 96 is significantly lower than 64 and 144 even more so.

Table 8.2: Learning with cut-off values

Cut-off = 32	Performance
32	64.31774 ± 1.084966
64	65.46011 ± 1.379469
96	61.31507 ± 1.612758
144	59.40155 ± 1.498132

8.5 Experiment: Champion contribution

Objective

At the end of each cycle, the new optimised evaluation function is merged with the champion list. During this merge process, the new evaluation function is ranked. The new function precedes the first champion that it beats in a two-game match. In this way the champions are always kept in rank order. This champion list is used to generate the next set of example play-lines. In this experiment, the contribution of the top champions to produce the example set is varied, and the effect of this variation on the learning performance is measured.

Method

For each learning cycle, 4500 play-lines are produced during the play phase. Starting with the top champion, the contribution are varied by specifying how many of these play-lines must be produced by each champion. Two per-champion values are considered: 4500 and 1500. In the 4500 case, only the first champion contributes to the play-lines. In the 1500 case, the first three champions have an equal contribution to the play-lines.

The experiment is run using the same settings used in the previous experiment (for the cut-off value of 64) with two exceptions. The first exception is that this experiment uses a search span of 20, and the second is that it learns for 5 macro learning cycles. For each contribution case, 30 5-cycles runs were executed.

Results

The resulting measurements are shown in Table 8.3. Although the 4500 contribution is slightly better, the high variance makes it impossible to decide which option is better. The conclusion is that the relative contribution of each champion has no effect on the learning performance.

Table 8.3: Learning with different contributions

Per champion contribution	Performance
1500	72.78122 ± 1.587015
4500	72.89189 ± 1.463084

8.6 Experiment: Game complexity

Objective

In this experiment, the learning framework is subjected to a simpler game to see whether the framework is more effective at learning simpler games.

Method

A game, called R C is created by changing the objective of C . The objective of R C is to get a king: the first player to crown a checker wins the game. This game is simpler because it eliminates the complexity of learning how to corner the opponent's kings during the end-game of C .

This experiment employed the same method as the previous experiment with a champion contribution of 4500. It uses a search span of 20 and a learning cycle count of 5. The performance measurement was taken for 30 different runs.

Results

The measured performance for R C was 95.69423 ± 0.2978747. This is a considerable improvement over the performance of 72.89189 obtained for standard C during the previous experiment. The conclusion is that the framework does indeed perform better when subjected to simpler games. This experiment also shows that the learning framework is applicable to more than just C .

8.7 Conclusion

The performance of the learning framework is largely unaffected by the use of overlapped phases and the contribution of the reigning champions. However, the C4.5 cut-off value that affects the function complexity has a definite effect on the learning performance. As expected, the learning framework performs much better when presented with a game that is less complex than C .

Chapter 9

Conclusion and future work

This chapter provides a summary of the findings of this work. It also provides a list of suggested research topics that would extend the work presented in this thesis.

9.1 Conclusion

The aim of this study was to introduce a learning framework that improves the skill of an agent equipped with zero knowledge. This framework must be applicable to a significant class of games. As constituents of this aim, a number of objectives were identified. These objectives entail reviews of game concepts as well as learning concepts. In addition, a representation scheme had to be introduced that is suitable for the representation and the learning of game knowledge. As a secondary aim, the current study also investigated the possibility of using tournament methods for particle swarm optimisation (PSO). This secondary aim has been achieved by incorporating the Tournament PSO as a key stage in an iterative learning process. This process embodies the learning framework. In the rest of this section, the manner in which the various objectives were achieved is described.

The review of game concepts highlighted that the game playing agent requires a high level of interaction between the components of game tree search and game knowledge. A game playing agent needs both components to play effectively. However, typical approaches to game tree search, such as alpha-beta, require a high quality evaluation function. Such a function is not available to a learning agent that starts its learning from zero knowledge.

A selective search algorithm uses game knowledge to guide the game tree search. This approach is more suitable for learning because the allows conceptually for the game tree search

to improve as the knowledge becomes better. The selective search algorithm developed by Korf and Chickering has been shown to be an improvement of alpha-beta, but this algorithm also requires strong game knowledge. The best-first shallow search algorithm introduced in this thesis performs better than best first minimax when knowledge is weak, and performs no worse than best first minimax when strong knowledge is available. Therefore, this new algorithm is a suitable game tree search algorithm to use when the intention is to learn from little or from zero knowledge.

Armed with a selective search technique, the focus of a learning agent is purely to gain knowledge. A structured representation for knowledge was preferred over neural networks because the aim is to obtain knowledge for use in a linear evaluation function, but also because structured knowledge is more readable by humans. In order to establish a common starting point for learning agents, a definition of zero-knowledge has been developed. According to this definition, the learning should be provided only with the game rules and with an interpretor that translates the observable elements in a game state to concepts described in the game rules.

A representation scheme for game knowledge is required that is flexible enough to represent the observable units of knowledge as well as the complex knowledge that aids the decision making process. The knowledge representation language developed as part of this has been shown empirically to be able to represent complex knowledge, and to be a practical tool for learning. In addition, this representation language can be applied to many different board games.

The process for deducing knowledge from example game positions has the C4.5 program as a cornerstone. The method introduced by this thesis to derive knowledge expressions from ID3 decision trees demonstrates the utility of the representation language. As a refinement to this process techniques to simplify the resulting knowledge expression has also been introduced. Experiments show that a particular combination of these simplification techniques called `DS` produces the simplest evaluation functions with the highest playing performance.

The scheme chosen to optimise the floating-point elements of the evaluation function employs the Particle Swarm Optimisation (PSO) algorithm. A number of tournament methods were introduced as alternative approaches to determine the relative fitness of a particle in the swarm. These alternatives are applied to a new PSO, called the Tournament PSO that uses the competition between a particle and its personal best to limit the number of competitions required at every epoch. Amongst the alternative methods, a tournament called `DSK` is empirically identified as the best method to use with Tournament PSO.

The learning framework combines the knowledge deduction process that employs DS with the Tournament PSO that uses DSK into an iterative learning cycle. This cycle starts with zero knowledge and uses examples produced as part of the learning process to build new knowledge. Experiments conducted with C show that this learning approach does lead to players that have more knowledge than random players. It has also been shown that the framework is more capable when it is subjected to a simpler game.

9.2 Future work

The learning framework introduced in this thesis presents many opportunities for further research. One reason for this is that the framework is a multi-stage approach to learning that allows the researcher to select alternative techniques within each stage. In particular, the discovery stage and the optimisation stage are primary targets for this kind of exploration. Another research avenue could be to discard the zero-knowledge constraint. Without this constraint many opportunities arise, such as the development of new methods that exploit rich domain knowledge within the learning framework. Such endeavours, as fruitful they may prove to be, have the potential to digress significantly from the design of the proposed framework.

However, the current framework is far from perfect. The best C performance levels obtained indicate that the framework could do much better (especially when compared with the R C results). Many research opportunities lie within the scope of refining and analysing the learning framework without affecting the design. The aim of such endeavours could be to identify weaknesses in the framework as a whole (or in its parts), and to introduce specific improvements. In the paragraphs below key ideas that provide direction for this kind of work are suggested.

Define a symbol definition method

A game specific parameter for the knowledge representation language is a set of symbols expressed as square states. These square states are then grouped to form the occupation state symbols that are used in the knowledge expressions. In this work, the process to derive the occupation states for C is described, but a more general set of rules and procedures need to be developed such that the occupation states for any given board game is deterministic. The methodology should ensure that the zero-knowledge principle is upheld.

Determine the effect of alternative regions

In addition to the symbols, the knowledge representation language also requires the demarcation of regions on the game board. The current research used a crude set of regions. This has been done to avoid breaching the zero-knowledge principle. It is possible that other general region configurations produce better knowledge (for example regions that do not overlap). Also, it is likely that important regions on the board can be identified as important from the game rules; or from game literature. An investigation into the effect that these alternative regions have on the learning performance could highlight the need to extend the framework to automatically discover optimal region configurations.

Use random trees to evaluate best first shallow search

The random game tree evaluation technique is used by game tree algorithm researchers to determine the effectiveness of the game tree algorithm. The application of this technique to evaluate the effectiveness of best first shallow search provides a domain independent assessment of the algorithm. In order to do this comparison, the random game tree must be constructed such that multiple nodes with the same value exists in the tree: in other words, these trees must be representative of trees produced by low quality knowledge. This assessment could also lead to the discovery of other circumstances in which this new method performs well.

Use logic to refine and simplify knowledge expressions

The knowledge expressions discovered by the induction process possibly contains rules that are redundant or contradictory. A logic processor for these expressions is needed to eliminate redundancies and find inconsistencies. This processor could also reduce the complexity of an expression without changing the intent of the expressions (such as removing two consecutive ‘negate’ operations). These improvements could lead to simpler expressions, and also to expressions that lead to better decisions.

Apply Tournament PSO to the training of neural networks

The utility of Tournament PSO can be explored further by using this PSO to train the same neural networks that have already been trained with other PSO algorithms. Such an investigation could provide insight into whether Tournament PSO has value in a more general context. Facets of Tournament PSO not covered by the experiments described in this thesis can be subjected to an

experimental treatment. In particular, alternative values for the PSO parameters, c_1, c_2 and v_{max} can be explored.

Find aspects that effect a game's learning potential

The question of whether or not the framework is generally applicable can be verified by using the framework to learn other games. One approach can involve the definition of more C variants, such that each variant modifies one or two of the game rules. The use of variants eliminates the need to define new game symbols, and provides an opportunity to identify which properties of a game make it more suitable as a learning subject for the framework.

Bibliography

[1] L. V. Allis. Searching for solutions in Games and Artificial Intelligence (PhD. Thesis). Rijks University, Maastricht, 1994.

[2] L. V. Allis. *Which games will survive?*, pages 155–184. In *Searching for solutions in Games and Artificial Intelligence (PhD. Thesis)* [1], 1994.

[3] L. V. Allis, H. J. van den Herik, and I. S. Herschberg. *Which games will survive*, pages 232–243. Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad. Ellis Horwood, Chichester, England, 1991.

[4] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 264–270, 1993.

[5] J. Baxter, A. Tridgell, and L. Weaver. *Machines that Learn to Play Games (Ed. J. Fürnkranz and M. Kubat)*, chapter 5, Reinforcement learning and chess. Nova Science Publishers, Huntington, NY, 2001.

[6] H.J. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14(2):205–220, September 1980.

[7] J.A. Boyan. *Modular Neural Networks for Learning Context-dependent Game Strategies*. University of Cambridge, Cambridge, August 1992.

[8] I. Bratko. *Prolog Programming for Artificial Intelligence Second Edition*, chapter 18. Machine Learning, pages 459–499. Addison-Wesley, 1990.

[9] M. Buro. Statistical feature combination for the evaluation of game positions. *Journal of Artificial Intelligence Research*, 3:373–382, 1995.

[10] M. Buro. From simple features to sophisticated evaluation functions. 1558, 1998.

[11] K. Chellapilla and D. Fogel. Evolving neural networks to play checkers without expert knowledge. *IEEE Transactions on Neural Networks*, 10(6):1382–1891, 1999.

[12] K. Chellapilla and D. Fogel. Anaconda defeats Hoyle 6-0: A case study competing an evolved checkers program against commercially available software. Proceedings of IEEE congress on Evolutionary Computation, pages 857–867, La Jolla Marriot Hotel, La Jolla, California, USA, July 2000.

[13] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6:58–60, February 2002.

[14] J.E. Davis and G. Kendall. An investigation, using co-evolution, to evolve an Awari player. Proceedings of the 2002 Congress on Evolutionary Computation, pages 1408–1413, 2002.

[15] M. Donskoy and J. Schaeffer. Perspectives on falling from grace. *International Computer Chess Association Journal*, 12(3):155–163, 1989.

[16] W.H. Duminy and A.P. Engebrecht. Title: Composing linear evaluation functions from observable features. *South African Computer Journal*, (35):48–58, December 2005.

[17] R.C. Eberhart and J. Kennedy. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[18] R.C. Eberhart and Y.H. Shi. Comparison between genetic algorithms and particle swarm optimization. In *Annual Conference on Evolutionary Programming*, San Diego, 1998.

[19] A.P Engelbrecht. *Computational Intelligence: An introduction*, chapter 15: Coevolution, pages 177–181. Wiley, 2002.

[20] S. L. Epstein. Toward an ideal trainer. *Machine Learning*, (15):251–277, 1994.

[21] D. B. Fogel. Using evolutionary programming to construct neural networks that are capable of playing Tic-tac-toe. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 875–879, San Francisco, 1993.

[22] D. B. Fogel. Blondie24: Playing at the edge of AI. 2001.

[23] D. B. Fogel, Timothy J. Hays, Sarah L. Hahn, and James Quon. A self-learning evolutionary Chess program. In *Proceedings of the IEEE*, volume 92, pages 1947–1954, 2004.

[24] N. Franken. *PSO-Based Coevolutionary Game Learning (M.Sc. dissertation)*. 2004.

[25] N. Franken and A.P. Engelbrecht. Comparing PSO structures to learn the game of checkers from zero knowledge. In *Proceedings of IEEE congress on Evolutionary Computation*, Canberra, Australia, 2003.

[26] N. Franken and A.P. Engelbrecht. Evolving intelligent game-playing agents. Proceedings of SAICSIT, pages 111–113, 2003.

[27] J. Fürnkranz. Machine learning in games: A survey. Technical Report OEFAI-TR-2000-31, Austrian Research Institute for Artificial Intelligence, 2000.

[28] A.G. Hammilton. *Logic for Mathematicians*. Cambridge University Press, 1988.

[29] T. P. Hart and D. J. Edwards. The alpha-beta heuristic. Technical Report 30, October 1963.

[30] IEEE. Washington DC, USA, July 1999.

[31] Peter Jackson. *Introduction to Expert Systems Third Edition*, chapter 20. Machine Learning, pages 380–401. Addison-Wesley, Essex, Harlow, 1999.

[32] J. Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proceedings of the Congress on Evolutionary Computation*, pages 1931–1938, 1999.

[33] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, Perth, Australia, Nov/Dec 1995.

[34] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In *Proceedings of IEEE congress on Evolutionary Computation*, volume 2, pages 1671–1676, May 2002.

[35] Richard E. Korf and David W. Chickering. Best-first minimax search. *Artificial Intelligence*, 84:299–337, 1996.

[36] Clifford Kotnik and Jugal Kalita. The significance of temporal-difference learning in self-play training TD-rummy versus EVO-rummy. In *Proceedings of the Twentieth International Conference on Machine Learning*, Washington DC, 2003.

[37] M. Kubat, I. Bratko, and R. Michalski. *Machine Learning and Data Mining: Methods and Applications*, chapter 1. A review of Machine Learning Methods, pages 1–72. John Wiley & Sons Ltd, 1996.

[38] K. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1 – 25, August 1988.

[39] Jim Loy. *The standard laws of Checkers*. www.jimloy.com/checkers/rules.htm.

[40] L Messerschmidt and A.P Engelbrecht. Learning to play games using a PSO-based competitive learning approach. In *Proceedings of the 40th Asia-Pacific Conference on Simulated Evolution and Learning*, Singapore, 2002.

[41] D. Michie. Experiments on the mechanization of game-learning - part i. characterization of the model and its parameters. *The Computer Journal*, (6):232–236, 1963.

[42] D. Michie. Machine learning in the next five years. Proceedings of the third European Working Session on Learning, Glasgow, 1988. London: Pitman.

[43] M. Minsky. chapter Steps toward artificial intelligence. McGraw-Hill, New York, 1963.

[44] D.H. Mitchell. *Using features to evaluate positions in experts and novices Othello games. Masters thesis*. Northwestern University, Evanston, IL, 1994.

[45] David Moriarty and Risto Miikkulainen. Evolving complex othello strategies using marker-based genetic encoding of neural networks. Technical Report AI93-206, 1, 1993.

[46] N. J. Nilsson. *Principles of artificial intelligence*. Springer Verlag, New York Berlin Heidelberg, 1982.

[47] K.E. Parsopoulos and M.N. Vrahatis. *Artificial Intelligence and Soft Computing*, chapter Particle Swarm Optimizer in Noisy and Continuously Changing Environments, pages 289–294. 2001.

[48] J. B. Pollack, A. D. Blair, and M. Land. Coevolution of a backgammon player. In *Proceedings of the Fifth International Conference on Artificial Life*, Nara, Japan, May 1996.

[49] David Poole, Alan Mackworth, and Randy Goebel. *Computational intelligence: a logical approach*. Oxford University Press, New York, Oxford, 1998.

[50] J. R. Quinlan. Induction of decision trees. *Machine Learning*, (1):81–106, 1986.

[51] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[52] E. Rich. *Artificial Intelligence*. McGraw-hill, 1983.

[53] A.L. Samuel. Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[54] J. Schaeffer. The role of games in understanding computational intelligence. *IEEE Intelligent Systems Journal*, pages 10–11, November/December 1999.

[55] J. Schaeffer, P. Lu, D. Szafron, and R. Lake. A re-examination of brute-force search. In *Games: Planning and Learning, Papers from the 1993 Fall Symposium*, pages 51–58, AAAI Press, 1993.

[56] J. Schaeffer and H. J. van den Herik. Games, computers, and artificial intelligence. *Artificial Intelligence*, (134):1–7, 2002.

[57] Jonathan Schaeffer and Robert Lake. Solving the game of checkers. *Games of no chance*, 29:119–133, 1996.

[58] C.E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, (27):623656, 1948.

[59] Y. Shi and R.C. Eberhart. A modified particle swarm optimizer. In *Proceedings of IEEE congress on Evolutionary Computation*, volume 3, Anchorage, AK, May 1998.

[60] Y. Shi and R.C. Eberhart. Empirical study of particle swarm optimization. In *Proceedings of IEEE congress on Evolutionary Computation* [30], pages 1949–1950.

[61] Y. Shoham and S. Toledo. Parallel randomized best-first minimax search. *Artificial Intelligence*, (137):156–196, 2002.

[62] J. Sonas. *Have chess computers surpassed the strongest grandmasters?* World Wide Web, http://www.chessbase.com/newsdetail.asp?newsid=1229, 2003.

[63] P.N. Suganthan. Particle swarm optimizer with neighbourhood operator. In *Proceedings of IEEE congress on Evolutionary Computation* [30], pages 1958–1961.

[64] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[65] R.S. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[66] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[67] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[68] G. Tesauro. *Machines that Learn to Play Games*, chapter 6: Comparison training of chess evaluation functions, pages 117–130. Nova Science Publishers, 2001.

[69] G. Tesauro and T.J. Sejnowski. A parallel network that learns to play backgammon. *Artificial Intelligence*, 39:357–390, 1989.

[70] W. Tunstall-Pedoe. Genetic algorithms optimizing evaluation functions. *International Computer Chess Association Journal*, 14(3):119–128, 1991.

[71] L. Underhill and D. Bradfield. *Introstat, Second Edition*. Juta & Co., Ltd, 2001.

[72] H. J. van den Herik, Jos W.H.M. Uiterwijk, and Jack van Rijswijk. Games solved: Now and in the future. *Artificial Intelligence*, (143):277–311, 2002.

[73] P.W. Wagacha. Induction of decision trees (lecture note*s), 2003.

[74] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small world' networks. *Nature*, (393):440–442, 1998.

[75] B. Widrow and M.E. Hoff. Adaptive switching circuits. In *WESCON Convention Record Part IV*, pages 96–104, 1960.

[76] R. P. Wiegand, W. C. Liles, and K. A. De Jong. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 1235–1242, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.