Informed Search in Complex Games

Informed Search in Complex Games

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. mr. G.P.M.F. Mols,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op woensdag 1 december 2004 om 12:00 uur

door

Mark Henricus Maria Winands

SIKS Dissertation Series No. 2004-17
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 90-5278-429-9
Universitaire Pers Maastricht

Printed by Datawyse b.v., Maastricht, The Netherlands.

Preface

In 1999 I became acquainted with the game LOA (Lines of Action). To me it was a fascinating and attractive topic of games research and I chose it for my M.Sc. project on knowledge representation and search. It was a joy working on it and my interest in games and AI enlarged every day. During this project I was offered to continue my research as a Ph.D. student at the computer science department of the Universiteit Maastricht, part of the research institute IKAT (Institute for Knowledge and Agent Technology). After some thought I accepted this offer, a choice I still enjoy. Then, submitted papers (a new world to me) were accepted and led to this thesis. The result would not have existed without the support of a variety of people. In this preface I want to acknowledge them.

First of all, I wish to thank my supervisor professor Jaap van den Herik for his stimulating efforts to teach me scientific writing and thinking, for the many other lessons I learned from him, and also for providing me with the opportunity to broaden my horizon internationally. Next, many thanks go to my daily advisor dr. Jos Uiterwijk, who always had time to give me advice. Without the help of both of them this thesis would not have existed. Part of this research was done at the Computer Games Research Institute (CGRI) of the Shizuoka University. I want to acknowledge professor Hiroyuki Iida's advice and support during my stay in Japan.

Moreover, I would like to thank the members of the Search and Games group for their useful comments and collaboration. I enjoyed working with my former roommate Levente Kocsis on our various machine-learning experiments in LOA. Although we hardly agreed, he kept me sharp with his intriguing opinions on society and computer science. Erik van der Werf confirmed the performance of the relative history heuristic in his solver MIGOS. Besides our game-programming joint ventures, we made some (scientific) trips over the globe. I want to thank Jeroen Donkers for his enthusiasm to share his broad technical knowledge on a large range of topics. I do not know what I would have done without his help on LaTeX formatting. Last but not least, Tony Werten's chess programming tricks (or better hacks) turned out to be useful in LOA as well.

Then, I would like to thank my colleagues at IKAT for their cooperation and help. In particular the support offered by the members of the secretarial staff, Joke Hellemons, Marlies van der Mee, Martine Tiessen, and Hazel den Hoed is highly appreciated. Furthermore, I thank Peter Geurtz for his computer support and his willingness to replace the various hardware components I broke.

Beyond IKAT many other people contributed in one or another way to this thesis.

I thank Yngvi Björnsson and Darse Billings for sharing their thoughts about LOA in general, and LOA evaluation functions in particular. Further, I would like to acknowledge Lars Eijssen for proof reading this thesis.

In het bijzonder wil ik in mijn ouders bedanken voor de mogelijkheden die ze me hebben geboden om me te ontwikkelen.

Mark Winands, 2004

Acknowledgements

The research has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems. Moreover, I gratefully acknowledge financial support over the years by the Universiteitsfonds Limburg / SWOL.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis investigates how search can be guided by knowledge in such a way that the search space is traversed efficiently and effectively. For this task we focus on the question how to combine knowledge with search. The more adequate the knowledge is, the better the search. If a search process is sufficiently endowed with knowledge and as a consequence the search is (rather) successful, we call the search process an *informed-search* process. In this chapter we provide a brief introduction on games and Artificial Intelligence (AI) (Section 1.1) and then discuss the notion of informed search (Section 1.2). Subsequently, we formulate our problem statement together with four research questions (Section 1.3). Section 1.4 provides a thesis overview.

1.1 Games and AI

Ever since humans achieved some degree of civilisation, they played games. The two most important reasons for games to be played are their intellectual challenge and their entertainment. For the first reason games are used as a testing ground for computational intelligence. Since the 1950s the AI community compares the computer performance with the human performance (Van den Herik and Iida, 2000), or otherwise stated: since the birth of AI computational intelligence is measured with respect to human intelligence. Shannon (1950) and Turing (1953) were the first to describe a chess-playing program, while Samuel (1959) wrote the first game-playing program in the domain of Checkers. In the beginning most AI research in games concentrated on abstract games like Chess and Checkers. Later on (in the 1970s) Backgammon and Bridge were added to this list, in particular since they possessed additional features, viz. stochastic information and incomplete information, respectively. All four games offer a pure abstract competition, with an exact closed domain (i.e., well-defined rules). The game state is easy to represent and the possible actions are known. Less abstract games like football, of which the domain is not properly described and the rules are more vague, did not attract any attention in the beginning of AI.

Since the 1950s there has been a steady improvement in the strength of game-playing programs. The quality of these programs can be roughly categorised into five classes (Allis, Van den Herik, and Herschberg, 1991a).

(1) **Solved games.** In certain games perfection is reached by solving them. This means that the program is always able to achieve the best (i.e., game-theoretic) value independent of the opponent. The last years quite a number of games have been classified in this category. We mention Connect-Four (Allis, 1988), Go-Moku (Allis, 1994), Nine Men's Morris (Gasser, 1995); and recently we saw the solution of Kalah (Irving, Donkers, and Uiterwijk, 2000), Renju (Wágner and Virág, 2001) and Awari (Romein and Bal, 2003).

(2) **Over-champion games.** This is a category of games where the programs are significantly stronger than the human world champion. The best example is the game of Checkers, where the game programs have achieved a level above human capacity since Chinook obtained the Man-Machine World Champion title against Tinsley in 1994 (Schaeffer, 1997). Other games where over-champion status has been reached are, for instance, Backgammon (Tesauro, 1994), Othello (Buro, 1997), and Scrabble (Sheppard, 2002).

(3) **Champion-level games.** In some games the top programs are on equal footing with the world champion. The most famous example is Chess. The victory of Deep Blue against the then reigning World Champion Kasparov in 1997 (Hsu, 2002) is a telling performance. Moreover, the tied matches of the current top chess programs Deep Fritz (Müller, 2002; Levy, 2003b) and Deep Junior (Müller, 2003) against the world-champion calibre players Kramnik and Kasparov, respectively, show that the level of computer Chess is equal to that of the world champion, but does not yet surpass them (Levy, 2003a).

(4) **Grandmaster-level games.** Then we have games where the programs can play on grandmaster level, but are still defeated by the world champion. The games of Poker (Billings *et al.*, 2000), Hex (Anshelevich, 2002), and Draughts (Guibert and Wesselink, 2003) are examples.

(5) **Amateur-level games.** Finally, there exists the category of games where the level of play is still at amateur level. In this category the level goes from strong amateur (e.g., Shogi) to weak amateur (e.g., Go) (Van den Herik, Uiterwijk, and Van Rijswijck, 2002).

We remark that comparing the quality of a game program against humans is somewhat arbitrary. Some games are less popular among humans than others. This means that goods results can be achieved quite easily by the lack of a decent human opponent. Nevertheless, a weakly-playing human might sometimes be the best available person to give an insight into the progress of the field. Summarising, we can state that for quite some games at least the grandmaster status has been reached.

It is only a matter of time before most of the game-playing programs will reach over-champion status. This does not mean that games, which fall in category 2, are no longer attractive for research. Below we provide four reasons that advocate further research. First, continuing research in this kind of games could lead to solving them. Second, continuing research in these games could generate new ideas, which could successfully be applied in games where the play is still weak. According to Fraenkel (1996) new ideas could even be used in mathematics and economics. Third, having the strongest program in a certain domain can still be an intellectual challenge for other researchers. For example, at the Computer Olympiads programs are competing against each other and the *auctores intellectuales* are eager to find out which program is the best. Fourth, an over-champion program could be a kind of almighty tutoring oracle, by which a grandmaster can improve his[1] play.

In the last years we observed two shifts in the domain of game-playing programs. First, there is the research shift from cognition towards perception, which resulted in more attention to the game of football (e.g., the RoboCup). Second, a shift was seen from the classic abstract games towards the more commercial games. Role-playing, adventure, and sport games have become increasingly more popular as a test domain for AI research. This is partly so because these games offer new challenges, such as real-time pathfinding and adversarial real-time planning. But it also stems from the fact that this subdomain has more resources (i.e., these games constitute a multi-billion enterprise) to do research. Whatever the case, all games will remain an intriguing subject for AI research in the future. Moreover, informed search is a topic that will be beneficial for all types of game research mentioned above.

1.2 Informed Search

In complex games a program designer usually lacks the knowledge to let the program play the game perfectly. For many years, search has been an adequate answer to bridge (partially) the knowledge gap. As is well known, most games cannot be played at an acceptable level without using domain knowledge, because the corresponding state space is too large to be searched completely in a reasonable amount of time. So, most games can neither be played by using knowledge nor by using search only. Both search and knowledge have their advantages and drawbacks (see, e.g., Breuker, 1998). A logical inference of this observation is attempting to combine the advantages from both sides into a well-functioning procedure. For a good understanding of the possibilities we examine the trade-off between the amount of knowledge used and the amount of search performed. Two well-known characteristics frequently used to measure a position in the trade-off spectrum are the cost of *time* and the cost of *memory*.

First, we look at the time characteristic: evaluating a position by using (some amount of) knowledge consumes time. On the one hand a program that uses sophisticated but time-consuming knowledge determines more accurately the merit of each node visited. It searches fewer nodes to find the best move or to solve the problem. On the other hand, a program that uses less detailed (and therefore less time-consuming) knowledge will search more nodes and will be able to perform a deeper search, thereby showing an adequate short-term tactical ability. A considerable amount of research interest has arisen in the last years to study this phenomenon (Schaeffer, 1986; Berliner *et al.*, 1990; Junghanns and Schaeffer, 1997; Heinz, 2000).

Second, we look at the memory characteristic: storing knowledge gathered during a game requires *memory*. The purpose of storing knowledge acquired during the search process is to re-use it at later times. According to Breuker (1998) there are two points of view: (1) we may reduce the search by using more memory (e.g., transposition tables in depth-first search) and (2) we may reduce the need for memory at the cost of additional searching (e.g., two-level best-first search).

[1]In contexts where the gender of a non-neutral third person is irrelevant, we will always use "he" and "his" to avoid the more cumbersome "(s)he" and "her/his".

The trade-off is dependent on the type of knowledge used. According to Berliner (1984) there are two basic types of knowledge available in the search: terminal knowledge and directing knowledge.

Terminal knowledge is applied to the leaf nodes of the search tree. If the leaf node represents a terminal position, it produces an exact value (win, loss, or draw). Otherwise a heuristic evaluation value is computed for the position represented by the leaf. This value can be interpreted in three different ways (Donkers, 2003): (1) it is a *prediction* of the game-theoretic value, (2) it measures the *probability* to win, and (3) it measures the *profitability* of the position. The third interpretation is mostly used in the evaluation function of game programs. The profitability of a position is only partly based on the prediction of the game-theoretic value. It may include also the strengths and weaknesses of the program. The profitability of a position is independent of the actual opponent; this is in contrast to the probability to win the position. In machine-learning techniques the probability is used; in practice it is dependent on the specific opponent(s). The profitability of an evaluation function is usually determined by the number of games that are won by letting the program play against itself or against other programs.

Directing knowledge guides the way the search tree is being built. In best-first search the added domain-specific information selects which node to expand next. In selective depth-first search methods knowledge is used to decide which branches to abandon (forward pruning) or to extend beyond the nominal depth (search extensions).

Search needs at least terminal knowledge to solve a problem. If the search is also using directing knowledge it is called *informed search*.

1.3 Problem Statement and Research Questions

In the previous section we discussed the relevance of informed search to improve computer-game play. That is precisely the topic of this thesis. The following problem statement guides our research.

> **Problem statement**: *How can we develop informed-search methods in such a way that programs significantly improve their performance in a given domain?*

In order to formulate an answer on the problem statement we have to identify a test domain in which we can test our informed-search methods. The domain has to fulfil the following three conditions: (1) there should be a complex balance between the search and knowledge needed; neither search (as in Awari) nor knowledge (as in Hex) should be the dominant component; (2) the domain should be sufficiently complex with respect to tactics and strategy; it should not be possible in the near future (say five years) to solve the game by knowledge or by search, or by a combination of both of them; (3) it should be a relative unexplored domain, which provides ample room for new ideas. For instance, in Checkers or Othello almost perfection is reached with current search methods and "pre-cooked" knowledge.

The game Lines of Action (LOA), which will be explained in the next chapter, satisfies these three conditions and will be used as test domain. As a guideline to our research we have formulated four explicit research questions, in which we investigate certain topics of informed search. They deal with (1) the evaluation function, (2) competitive proof-number search algorithms, (3) forward pruning, and (4) move ordering.

> **Research question 1**: *How can we build a strong evaluation function for Lines of Action?*

Informed search cannot exist without a decent evaluation function (terminal knowledge). For LOA, it is a challenge to build such an evaluation function, since it should incorporate the basic principles of the game and simultaneously increase the profitability. The difficulty lies in the fact that knowledge about LOA evaluation functions is not well developed, although some material on the basic principles has been published recently (Handscomb, 2000a, 2000b, 2000c; Chaunier and Handscomb, 2001; Billings and Björnsson, 2003).

> **Research question 2**: *How can we develop a proof-number search algorithm, which is competitive in speed and not restricted in working memory?*

The original Proof-Number (PN) search method (Allis, Van der Meulen, and Van den Herik, 1994) is formulated as a best-first search algorithm. It has the disadvantage that the whole search tree has to be stored in memory. Then the search can end prematurely because of memory exhaustion. Recently, some PN variants have been constructed as depth-first search algorithms; yet they behave as their corresponding best-first search algorithms. The advantage is that there is no longer a need to store the whole tree in memory. The disadvantage is that the PN variants have to re-generate the tree in each iteration. In this thesis we investigate a new PN variant, called PDS-PN, which does not suffer from the drawbacks mentioned above. We will compare its performance and memory requirements with enhanced $\alpha\beta$ search and state-of-the-art PN variants.

> **Research question 3**: *How can we improve forward-pruning methods in the Principal-Variation-Search framework?*

For long time brute-force $\alpha\beta$ search was the standard procedure in games like Chess (Marsland and Björnsson, 2001). Enhancing the search with forward pruning has improved game-playing performance during the last ten years. In this thesis we will look whether it is beneficial to improve forward-pruning methods in the Principal-Variation-Search (PVS) framework. PVS is in general more efficient than the original $\alpha\beta$. The forward-pruning methods under consideration are *null move* and a relatively new one, called *multi-cut*.

> **Research question 4**: *How can we use information gained during the search to improve move ordering?*

Move ordering (directing knowledge) is one of the main techniques to decrease the size of the $\alpha\beta$ search tree. There exist several move-ordering techniques, which can be qualified by their dependency on the search algorithm (Kocsis, 2003). *Static* move ordering is independent of the search. These techniques rely on game-dependent knowledge. The ordering can be acquired by using expert knowledge (e.g., favouring capture moves in Chess) or by learning techniques (e.g., the Neural MoveMap Heuristic (Kocsis, Uiterwijk, and Van den Herik, 2001b)). *Dynamic* move ordering is dependent on the search. These techniques rely on information gained during the search. The transposition-table move (Breuker, Uiterwijk, and Van den Herik, 1996), the killer moves (Akl and Newborn, 1977), the history heuristic (Schaeffer, 1983), and the butterfly heuristic (Hartmann, 1988) are well-known examples. In this thesis we present a new dynamic move-ordering variant, called the relative history heuristic, to replace the history heuristic.

1.4 Thesis Overview

The contents of the thesis is as follows. Chapter 1 contains an introduction, the problem statement, four research questions, and an overview of the thesis.

Chapter 2 introduces the test environment. It explains the game of Lines of Action (LOA), which will be used as test domain in this thesis. We provide some background information, the rules of the game, a variety of game characteristics, seven basic principles, and a review of the role of LOA in the AI domain. The search engine of the LOA tournament program MIA, used as test vehicle for all experiments in this thesis, is described.

Chapter 3 answers the first research question by investigating which features are important for a LOA evaluation function. The features are based on the basic principles described in Chapter 2. It turns out that the following nine features are important: concentration, centralisation, centre-of-mass position, quads, mobility, walls, connectedness, uniformity, and player to move. These features have resulted in the evaluator MIA IV.[2] The evaluator is tested in a tournament against other LOA evaluators, which have performed well at the previous Computer Olympiads. Experiments show that MIA IV defeats them with large margins.

In Chapter 4 we start by providing a short description of the original PN-search method, and two main successors of the original PN search, i.e., PN^2 search and depth-first variants of PN search such as *Proof-number and Disproof-number Search* (PDS). A comparison of the performance between PN, PN^2, PDS, and $\alpha\beta$ is given. It is shown that PN-search algorithms clearly outperform $\alpha\beta$ in solving endgame positions in LOA. However, the memory problems make the plain PN search a weaker solver for the harder problems. PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$. But PN^2 is restricted by its working memory, and PDS is considerably slower than PN^2. A solution to this is offered in the next chapter. Finally, we will have a brief look on real-time applications of PN search.

[2]The program development of our research investigations is done under the name MIA (Maastricht in Action). Hence, the general name of the LOA program participating in tournaments is MIA. To distinguish between versions with different evaluators we sometimes identify the name of the program with the name of its specific evaluator. During the research we have developed four evaluators, viz. MIA I, MIA II, MIA III, and MIA IV.

Chapter 5 answers the second research question and presents a new proof-number search algorithm, called PDS-PN. It is a two-level search (like PN^2), which performs at the first level a depth-first PDS, and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS. Experiments show that within an acceptable time frame PDS-PN is more effective for really hard endgame positions than $\alpha\beta$ and any other PN-search algorithm.

Chapter 6 answers the third research question. Forward-pruning methods, such as multi-cut and null move, are tested at so-called ALL nodes. Principal Variation Search (PVS) is improved by four small but essential additions. The new PVS algorithm guarantees that forward pruning is safe at ALL nodes. Experiments show that multi-cut at ALL nodes (MC-A) when combined with other forward-pruning mechanisms gives a significant reduction of the number of nodes searched. In comparison, a (more) aggressive version of the null move (variable null-move bound) gives less reduction at expected ALL nodes than our algorithm. Finally, it is demonstrated that the playing strength of the LOA program MIA is significantly increased by MC-A.

Chapter 7 answers the fourth research question by describing a new method for move ordering, called the relative history heuristic. It is a combination of the history heuristic and the butterfly heuristic. Instead of only recording moves which are the best move in a node, we also record the moves which are applied in the search tree. Both scores are taken into account in the relative history heuristic. In this way we favour moves which on average are good over moves which are sometimes best. Experiments show that this method gives a reduction between 10 and 15 per cent of the number of nodes searched. Preliminary experiments in Go confirm this result. The relative history heuristic seems to be a valuable element in move ordering.

The research conclusions and recommendations for future investigations are given in Chapter 8.

Appendix A provides information on the performance of MIA at the LOA tournaments of the various Computer Olympiads. In Appendix B the pseudo code for PN, PN^2, and PDS is given.

Chapter 2

Test Environment

This chapter is based on the publication:
Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001a). The Quad Heuristic in Lines of Action. *ICGA Journal*, Vol. 24, No. 1, pp. 3–15.[1]

The chapter describes the test environment used to answer the problem statement and the four research questions formulated in Chapter 1. A test environment consists of a game and a game program. The game under consideration is the game of Lines of Action. In Section 2.1 we give some background information of the game. The rules are explained in Section 2.2. The characteristics and basic principles of LOA are given in Sections 2.3 and 2.4. Section 2.5 gives a short review of the role of LOA in the AI domain. The search engine of the LOA-playing program MIA (Maastricht in Action) is briefly described in Section 2.6. Finally, in Section 2.7 we will provide the conditions which LOA test sets have to fulfil.

2.1 Lines of Action

Lines of Action (LOA) is a two-person zero-sum game with perfect information; it is a chess-like game with a connection-based goal, played on an 8×8 board. LOA was invented by Claude Soucie around 1960. Sid Sackson (1969) described the game in his first edition of *A Gamut of Games*. The objective of a connection game is to group the pieces in such a way that they connect two opposite edges (a static goal) or form a fully-connected group (a dynamic goal). The precise definition of what constitutes a connection depends on the game in question. Whatever the case, the notion of connection became one of the great themes in the world of abstract gaming. Many prominent game inventors made a contribution to this theme. Other examples of connection games are TwixT (Bush, 2000) and Hex (Anshelevich, 2002). In contrast to the typical connection games, LOA is more chess-like because (1) pieces are moved over the board instead of put on the board, and (2) pieces can be captured.

[1]The author is grateful to the Editor of the *ICGA Journal* for granting permission to reuse part of the article in this thesis.

2.2 The Rules

LOA is played on an 8×8 board by two sides, Black and White. Each side has twelve (checker) pieces at its disposal. In this thesis we are using the rules which are used at the Computer Olympiads and at the MSO World Championships. Below they are formulated in 9 rules. In some books, magazines or tournaments, the rules 2, 7, 8, and 9 are different from what is specified here.

1. The black pieces are placed in two rows along the top and bottom of the board, while the white pieces are placed in two files at the left and right edge of the board (see Figure 2.1a).
2. The players alternately move a piece, starting with Black.
3. A move takes place in a straight line, exactly as many squares as there are pieces of either colour anywhere along the line of movement (see Figure 2.1b).
4. A player may jump over its own pieces.
5. A player may not jump over the opponent's pieces, but can capture them by landing on them.
6. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit. Connected pieces are on squares that are adjacent, either orthogonally or diagonally (e.g., see Figure 2.1c). A single piece is a connected unit.
7. In the case of simultaneous connection, the game is drawn.
8. If a player cannot move, this player has to pass.
9. If a position with the same player to move occurs for the third time, the game is drawn.

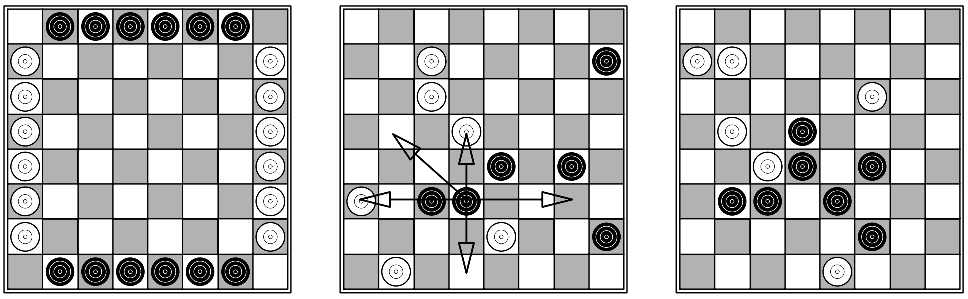

Figure 2.1: (a) The initial position. (b) An example of possible moves. (c) A terminal position.

In the thesis we use the standard chess notation for LOA. The possible moves of the black piece on **d3** in Figure 2.1b are indicated by arrows. The piece cannot move to **f1** because its path is blocked by an opposing piece. The move to **h7** is not allowed because the square is occupied by a black piece.

2.3 Characteristics

Analysis of 2585 self-play matches showed an average branching factor of 29 and an average game length of 44 ply. The game-tree complexity is estimated to be $O(10^{64})$ (Winands, Uiterwijk, and Van den Herik, 2001a) and the state-space complexity $O(10^{23})$. The game-tree complexity and state-space complexity are comparable with those of Othello (Allis *et al.*, 1991a). Considering the current state-of-the-art computer techniques, it seems that LOA is not solvable by brute-force methods.

A characteristic property of LOA is that it is a converging game (Allis, 1994), since the initial position consists of 24 pieces, and during the game the number of pieces (usually) decreases. However, since most terminal positions have still more than 10 pieces remaining on the board (Winands, 2000), endgame databases are (probably) not effectively applicable in LOA. As a case in point, we remark that an endgame database of ten pieces would require approximately 10 terabytes. Therefore, LOA seems an appropriate test domain for endgame solvers, such as PN-search algorithms.

2.4 Basic Principles

As stated before, LOA is a chess-like connection game based on a few general basic principles. Unlike Chess and Checkers, where material is the dominant principle (Shannon, 1950; Schaeffer, 1997), there is no principle which dominates the others in LOA. Strong LOA players have identified several basic principles which they recognise and use as fundamentals for a strategy to play LOA. Currently, there exists a debate on whether some of the basics principles are really fundamentals for good play (Billings and Björnsson, 2003). Disregarding this debate, below we will discuss seven basic principles, which are mentioned frequently as important principles (see Handscomb, 2000a, 2000b, 2000c; Chaunier and Handscomb, 2001). The seven basic principles are: threats (2.4.1), solid formations (2.4.2), mobility (2.4.3), blocking (2.4.4), centralisation (2.4.5), material advantage (2.4.6), and initiative (2.4.7).

2.4.1 Threats

If a player can connect in one move all its pieces in a single connected unit, the position contains a threat. A threat is a basic principle, because it is an effective way to devastate the opponent's position. The opponent typically has to break up its own formation to undermine the threat (Billings and Björnsson, 2003). For instance, in Figure 2.2 we see that White threatens to win by the move **e8-e6**. Black can only prevent this by **g4xe4**, weakening its own formation.

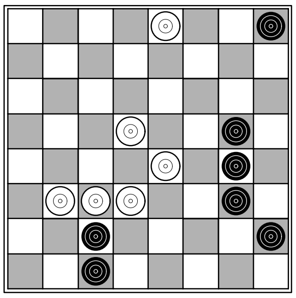

Figure 2.2: A position with a threat.

2.4.2 Solid Formations

A solid formation denotes a group of pieces that is connected in more than one direction in such a way that the group cannot be split into separated groups in a single capture move by the opponent. Solid formations are a basic principle because it is hard for the opponent to disconnect the pieces (Handscomb, 2000b). In Figure 2.3 Black has created a solid formation in the centre of the board, which is hard to destroy. However, creating such a formation usually takes many moves, during which the opponent may create a threat. It is good practice to investigate first whether it is possible to connect a piece to a group of connected pieces, and only thereafter to see whether a solid formation can be made.

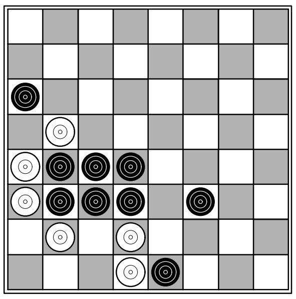

Figure 2.3: A solid formation.

2.4.3 Mobility

It is important to have a position with many options. Increased mobility makes it easier to connect your own pieces or obstruct the connection of the opponent pieces (Handscomb, 2000c). Therefore mobility is a basic principle. We remark that certain move types are more important than others. In the implementation of mobility in MIA we come back to this issue (see Subsection 3.1.5).

2.4.4 Blocking

Because a piece is not allowed to jump over the opponent's pieces, it can happen that the piece is blocked, i.e., cannot move. The fourth basic principle is blocking, since blocking a piece far away from the other pieces is an effective way of preventing the opponent to win. For instance, in Figure 2.4 White must first spend many moves to free the blocked piece, i.e., capturing pieces around the blocked piece on **a8**, before it can successfully resume the connection goal. During that time, Black may set up a winning strategy. Even partial blocking can be quite effective, especially if it forces a player to find a way around the opponent's pieces. Handscomb (2000a) describes that it is a common tactic to create a wall on the b- or g-file, or the 2^{nd} or 7^{th} rank in the opening phase of the game, which partially blocks pieces of the opposing side.

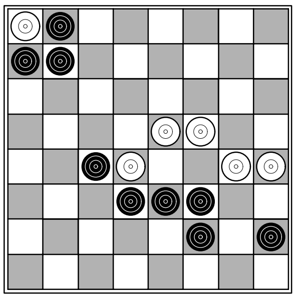

Figure 2.4: Blocking a piece.

2.4.5 Centralisation

Centralisation means that pieces dominating the centre are regarded to be more important than other pieces. Centralisation can be done in two ways: (1) putting the pieces in the centre (*actual centralisation*) or (2) controlling the centre by tactical countermeasures. Obviously centralisation is a basic principle, because pieces have to move through the centre to connect with each other. However, Handscomb (2000b) argues that the benefits of actual centralisation in LOA are sometimes exaggerated. Because of the nature of the game, pieces huddled together in the middle of the board are incapable of capturing each other. Even with only two pieces in a given line of action, a distance of two squares is needed for a capture, and if there are three pieces a considerable distance between the pieces is required. In Figure 2.5 the white pieces are scattered around the edges of the board. Nevertheless, the white piece on **h4** can destroy Black's formation in the middle by capturing the black piece on **d4**. So, in this case, the white pieces on the edges of the board are acting as cruise missiles.

Of course, this is not possible when all the pieces are in the centre. The example shows that controlling the centre might be better than occupying it. Hence, actual centralisation has a limited importance, but successfully controlling the centre is an important component in the evaluation function. Through an effective control of

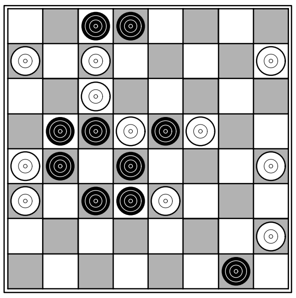

Figure 2.5: LOA email tournament, 1999, Roessner vs. Handscomb, after **10. e1-c3**.

the centre, the two starting groups of the opponent remain separated, whereas the own groups can be brought together.

2.4.6 Material Advantage

Contrary to Chess and Checkers it is not clear whether the principle of having extra material is an advantage, a disadvantage, or neutral. At first sight capturing the opponent's pieces is not a good idea, because the opponent then needs to connect fewer pieces. However, it is not hard to see that capture moves are beneficial in some positions. First, capturing a piece such that the opponent's formation is destroyed is mostly a good move. Second, it also happens that a player is able to connect its own formation only by a capture move. Third, a capture move may be the only way to free a blocked piece. Fourth, some authors argue that capturing pieces for the sake of a material advantage is sometimes appropriate (Handscomb, 2000c), because it contributes to a higher mobility. A player with a material advantage has more opportunities to prevent the opponent's threats and create its own threats. Whether material really is an advantage depends on the position. Whatever the case, for LOA it is a basic principle. Finally, we remark that material is probably strongly interrelated to other principles.

2.4.7 Initiative

In their article *The Advantage of the Initiative* Uiterwijk and van den Herik (2000) define the initiative as having the right to move first. They show that the initiative plays an important part when focussing on the solving of small games, such as Domineering and k-in-a-row games. In the large games Chess and Go the initiative is an important factor too (Uiterwijk and Van den Herik, 2000). Since in LOA with its state-space complexity of $O(10^{23})$ having the initiative constitutes also an advantage, we consider it as one of the basic principles. Following Uiterwijk and van den Herik's idea, we remark that LOA 3×3 is win for the second player, but that 4×4 and 5×5 are wins for the first player. A close analysis of these three sizes reveals that the 3×3 board is an exception due to its size. For the 4×4 and 5×5 boards the initiative is a decisive factor. No conclusion can be made for the $n \times n$ boards, but we conjecture that having the initiative is sufficient to be a basic principle. Moreover, the endgame in LOA often results in a race to full connection. So having the first move seems to be an advantage in LOA. It is quite rare in LOA that the obligation to move leads to a decisive deterioration of the position (i.e., *zugzwang*).

2.5 The Role of LOA in the AI Domain

Around 1975 LOA received its first credits as an AI research topic. For instance, then the first LOA program was written at the Stanford AI laboratory by an unknown author. In the 1980s and 1990s "hobby" programmers wrote several LOA programs. However, all were easily beaten by humans (Dyer, 2000). At the end of the nineties LOA became a clear objective or even target of AI researchers. Considering the role of LOA in the AI domain we can distinguish two different categories.

The first category consists of researchers using LOA as a test domain for their algorithms. Eppstein (1997) mentioned his dynamic planar-graph techniques to evaluate the connectivity of LOA positions. Kocsis (2001a; 2001b) applied successfully his learning time-allocation algorithms and his new move-ordering method in LOA, called the Neural MoveMap heuristic. Moreover, Björnsson (2002) confirmed the good results of his multi-cut method for LOA. Up to then it was only tested for Chess. Donkers (2003) used LOA to test the admissibility in opponent-model search. Sakuta *et al.* (2003) investigated the application of the killer-tree heuristic and the λ-search method to the endgame of LOA. These techniques were initially developed for Shogi. Hashimoto *et al.* (2003) chose LOA as a test domain for his automatic realisation-probability search method.

The second category consists of researchers trying to build strong LOA programs by using new ideas. For instance, the programs MIA (Maastricht In Action), BING, YL, and MONA belong to this category. MONA was the first program to win the Annual E-mail Tournament (the unofficial world championship for LOA) in 2001, with a perfect 14-0 record, including wins over most of the best LOA players in the world (see Billings and Björnsson, 2003 for more details). Since 2000 LOA is played at the Computer Olympiad. This is a multi-games event in which all of the participants are computer programs. For details on the LOA tournaments we refer to Appendix A. A demo version of MIA can be played online at the website: http://www.cs.unimaas.nl/m.winands/loa/. The program has been written in Java and can easily be ported to all platforms supporting Java.

2.6 MIA's Search Engine

In this thesis MIA's search engine is used as test vehicle for the experiments. The standard framework of $\alpha\beta$ search (Knuth and Moore, 1975) with all kinds of enhancements (Marsland, 1986) offers a good start for building a strong LOA-playing program. Thus, MIA started its career with an $\alpha\beta$ depth-first iterative-deepening search. Below we briefly describe MIA's original basic design, which serves as a starting point for our research. Several techniques were implemented to enhance the search. We mention (1) Principal Variation Search (PVS), (2) transposition tables, (3) forward pruning, (4) move ordering, and (5) quiescence search.

First, the program uses PVS to narrow the $\alpha\beta$ window as much as possible (Marsland and Campbell, 1982). This means that the β value equals $\alpha + 1$. Such an algorithm is in general more efficient than the original $\alpha\beta$. The basic idea behind the method is that it is cheaper to prove a subtree inferior, than to determine its exact value. It has been shown that this method does well for bushy trees such as occur in Chess. Because the branching factor of LOA (29) is in the same range as that of Chess (38), it works fine in LOA too. Another popular algorithm for searching game trees is NegaScout (Reinefeld, 1983). The two algorithms (PVS and NegaScout) are essentially equivalent to each other; they expand the same search tree (Björnsson, 2002).

Second, a *two-deep* transposition table (Breuker *et al.*, 1996) is applied to prune a subtree or to narrow the $\alpha\beta$ window. The well-known Zobrist-hashing method (Zobrist, 1970) is used for storing the entries in the table. At all interior nodes which are more than 2 ply away from the leaves, the program generates all the moves to perform the Enhanced Transposition Cutoffs (ETC) scheme (Schaeffer and Plaat, 1996).

Third, two forward-pruning techniques are applied. A null move (Donninger, 1993) is performed before any other move and it is searched to a lower depth (reduced by R) than other moves. A variable scheme, called adaptive null move (Heinz, 1999), is used to set R. If the remaining depth is more than 6, R is set to 3. When the number of pieces of the side to move is lower than 5 the remaining depth has to be more than 8 to set R to 3. In all other cases R is set to 2. If the null move does not cause a β cut-off, multi-cut (Björnsson and Marsland, 1999) is performed.

Fourth, for move ordering, (1) the move stored in the transposition table, if applicable, is always tried first. Then (2) two killer moves (Akl and Newborn, 1977) are tried. These are the last two moves, which were best or at least caused a cut-off at the given depth. Thereafter follow: (3) capture moves going to the inner area (the central 4×4 board) and (4) capture moves going to the middle area (the 6×6 rim); finally, (5) all the other moves are ordered decreasingly according to the history heuristic (Schaeffer, 1983).

Fifth, in the leaf nodes of the tree a quiescence search is performed, since the evaluation function should only be applied to positions that are quiescent. This quiescence search looks at capture moves that form or destroy connections (Winands *et al.*, 2001a) and at capture moves going to the central 4×4 board.

We remark that the various proof-number search algorithms to be discussed in Chapters 4 and 5 are not part of the original search engine. Moreover, in Chapter 6 the original search engine will be enhanced with multi-cut at ALL nodes and in Chapter 7 with the relative history heuristic.

2.7 LOA Test Sets

In contrast to Chess there are no widely accepted test sets in LOA. This fact has given us the additional challenge to construct our own test sets. The sets should at least fulfil three conditions (see below). All test sets have the following two conditions in common: (1) the positions should have occurred in real games, and (2) the players who created the positions should be strong. Therefore, we selected the positions from games played between strong human players, man-machine games, games played at the Computer Olympiads, and other computer-computer contests. Moreover, we constructed two types of sets: (a) sets of positions to test our proof-number search algorithms, and (b) sets of positions to test the $\alpha\beta$-search enhancements.

The first type consists of endgame positions. They have to fulfil the additional condition that (3a) the positions should not be too trivial to solve. Therefore, positions are selected which are 10 plies or more before the end of the game. These sets are used in Chapters 4 and 5. Sakuta *et al.* (2003) used successfully one of our sets of endgame positions to test the killer-tree heuristic and the λ-search method.

The second type consists of positions which are (3b) uniformly selected from real games. Because the $\alpha\beta$-search enhancements are applied during the game, test positions from the opening and middle game have to be included. These sets are used in Chapters 6 and 7.

All test sets can be found at http://www.cs.unimaas.nl/m.winands/loa.

Chapter 3

An Evaluation Function for Lines of Action

This chapter is an updated and abridged version of the following two publications:

1. Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001a). The Quad Heuristic in Lines of Action. *ICGA Journal*, Vol. 24, No. 1, pp. 3–15.
2. Winands, M.H.M., Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2003a). An Evaluation Function for Lines of Action. *Advances in Computer Games 10: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 249–260. Kluwer Academic Publishers, Boston, MA, USA.[1]

The chapter answers the first research question by investigating which features are important for a LOA evaluation function. These features are partially based on the basic principles described in Chapter 2. The standard framework of the $\alpha\beta$ search with its enhancements offers a good start for building a strong LOA program (see Section 2.6). But, the real challenge in LOA is building a decent evaluation function, which incorporates the strategic intricacies of the game. The difficulty lies in the fact that knowledge about LOA evaluation functions is not well developed, although some material on this topic has been published recently (Billings and Björnsson, 2003). In this chapter we discuss the latest evaluation function used in the program MIA.

The chapter is organised as follows. In Section 3.1 the relevant features of the evaluation function are enumerated and explained. The profitability of the evaluation function is tested against other evaluators in Section 3.2. An evaluation of the features and a review of their interdependencies are given in Section 3.3. Finally, in Section 3.4 we present our chapter conclusion and topics for future research.

[1]Thanks are due to the Editor of the *ICGA Journal* and the Editors of *Advances in Computer Games 10* for giving permission of reusing relevant parts of the articles in this thesis.

3.1 Features of an Evaluation Function

Below MIA's evaluation function is explained in detail. The evaluator consists of the following nine features: *concentration*, *centralisation*, *centre-of-mass position*, *quads*, *mobility*, *walls*, *connectedness*, *uniformity*, and *player to move*. The nine features follow directly from the seven basic principles given in Chapter 2. The choice of features that fully cover the description of a position is most relevant. It is better to have all features correct and all the initial weights wrong than to have the initial weights correct and miss one of the (important) features (Bushinsky, 2004). The description of the features follows below; relevant examples and clarifications are given, adequate references to further details are supplied (Subsections 3.1.1 to 3.1.9). It is followed by some information about the use of caching (Subsection 3.1.10).

3.1.1 Concentration

The concentration feature is based on the basic principles of threats and solid formations. It measures how close pieces are to each other. By doing so, we reward positions with pieces in each other's neighbourhood. It is hoped that the pieces eventually will be connected in a solid formation or will create threats to win.

The concentration of the pieces is calculated by a centre-of-mass approach (see also the third feature). In MIA it is done in four steps. First, the centre-of-mass of the pieces on the board is computed for each side (in MIA this is done incrementally to save time). Second, we compute for each piece its distance to the centre-of-mass. The distance is measured as the minimal number of squares from the piece to the centre-of-mass. These distances are summed together, called the sum-of-distances. Third, the sum-of-minimal-distances is looked up in a table. The sum-of-minimal-distances is dependent on the number of pieces on the board (see the example below) and it is defined as the sum of the minimal distances of the pieces from the centre-of-mass. This number is necessary since otherwise boards with a few pieces would be preferred. For instance, if we have ten pieces, there will be always eight pieces at a distance of at least 1 from the centre-of-mass, and one piece at a distance of at least 2. In this case the sum-of-minimal-distances is 10. Thus, the sum-of-minimal-distances is subtracted from the sum-of-distances, the result being called the surplus-of-distances. Fourth, we calculate the concentration, defined as the inverse of the average surplus-of-distances. The pseudo code of the concentration feature is given in Figure 3.1.

The disadvantage of this feature is that it aims to connect as many pieces as possible in a local group, hardly worrying about some remote pieces (orphans). It is sometimes hard to connect these orphans. For instance, in Figure 3.2 the black pieces are grouped around their centre-of-mass at **e2**, but the black piece on **b8** is rather far away from this group.

```
.......................................................
//com = centre-of-mass
//Compute total distance of all the pieces towards c.o.m.
 for(each p Pieces){
 //Compute distance
    difrow = abs(com.row - p.row);
    difcol = abs(com.col - p.col);
    sum_of_distances += (difrow > difcol) ? difrow : difcol;
 }
sum_of_min_distances = lookupTable(Pieces);
surplus_of_distances = sum_of_distances - sum_of_min_distances;
concentration = 1 / surplus_of_distances;
.......................................................
```

Figure 3.1: Pseudo code for the concentration feature.

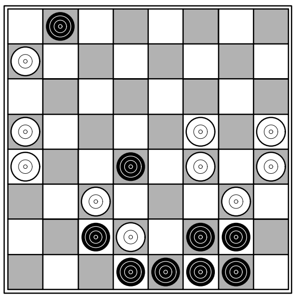

Figure 3.2: Position with an outlier on **b8**.

3.1.2 Centralisation

The centralisation feature is based on the basic principle of the same name. According to the centralisation feature pieces controlling the centre are more important than others. Centralisation is important because pieces have to move through the centre to connect with each other.

Analogous to piece-square tables in Chess, each piece obtains a value dependent on its board square in MIA. Pieces at squares closer to the centre are given higher values than the ones farther away. Pieces at the edge are given a negative value. This is done because such pieces are easy to block by a wall (but see Subsection 3.1.6). Pieces at the corner are punished even more severely. To prevent the program from over-aggressively capturing pieces, the average is computed instead of the sum of piece values. The piece-square table for the black pieces as used in MIA IV is given in Figure 3.3. The numbers are based on strategic principles and tuned by experiments.

We remark that centralisation can also be obtained indirectly by punishing moves not going to the centre. This is done in the mobility feature (see Subsection 3.1.5).

```
...................................................
pieceSquareTable[] = {-80, -25, -20, -20, -20, -20, -25, -80,
                      -25,  10,  10,  10,  10,  10,  10, -25,
                      -20,  10,  25,  25,  25,  25,  10, -20,
                      -20,  10,  25,  50,  50,  25,  10, -20,
                      -20,  10,  25,  50,  50,  25,  10, -20,
                      -20,  10,  25,  25,  25,  25,  10, -20,
                      -25,  10,  10,  10,  10,  10,  10, -25,
                      -80, -25, -20, -20, -20, -20, -25, -80};
...................................................
```

Figure 3.3: Piece-square table.

3.1.3 Centre-of-Mass Position

The centre-of-mass position feature is indirectly based on the basic principle of solid formations. It evualates the global position of all the pieces. This means that it looks at the position of the centre-of-mass on the board. The initial idea was to prevent formations from being built on the edges, where they are rather easily destroyed or blocked.

The value of this feature is dependent on the board square of the centre-of-mass. We use a simple table lookup for computation in MIA. Interestingly, after applying Temporal-Difference (TD) learning to enhance the weights, the weight for the centralised centre-of-mass feature changed its sign (Winands *et al.*, 2002), which means that opposite to expectations it is good to have the centre-of-mass closer to the edge instead of in the centre.

If the centre-of-mass is in the centre, it is possible that pieces are scattered over the board (e.g., the white pieces in Figure 3.4). If the centre-of-mass is at the edge, pieces have to be in the neighbourhood of each other, otherwise they would lie outside the board. Therefore, this feature contributes to the concentration and indirectly to the connectedness (see Subsection 3.1.7). Another plausible explanation of why it is worse to have the main piece formation in the centre is that it can be more

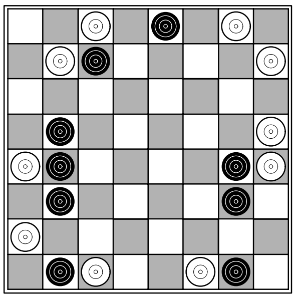
Figure 3.4: Scattered pieces.

easily attacked at that place, whereas groups residing closer to the edge can only be attacked from one side.

3.1.4 Quads

The quads feature is based on the basic principles of solid formations and material advantage. It looks at the solidness of the formation in particular. The feature favours pieces, which are connected in more than one direction, because it is harder for the opponent to disconnect them. The use of quads for a LOA evaluation function was first proposed and implemented by Dave Dyer in 1996 in his program LOAJAVA and empirically evaluated by Winands *et al.* (2001a). The heuristic is based on the use of quads, an Optical Character Recognition method. A quad is defined as a 2×2 array of squares (Gray, 1971). In LOA there are 81 quads for each side, including also quads covering only a part of the board along the edges. Taking into account rotational equivalence, there are six different quad types, depicted in Figure 3.5.

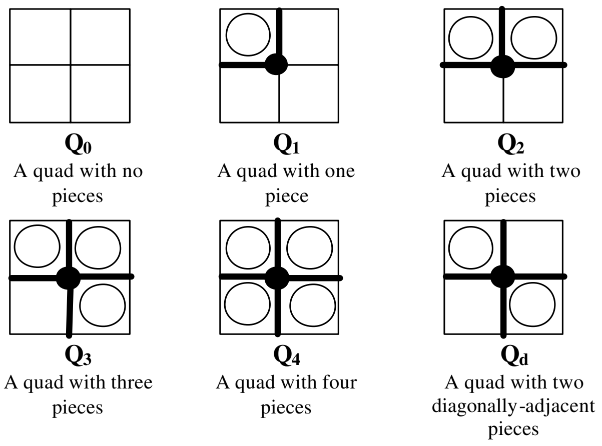

Figure 3.5: Six different quad types.

In this feature we only consider quads of three (Q_3) or four pieces (Q_4) of the same colour, since it is impossible to destroy these formations by a single capture. However, the danger exists that many of those quads are created outside the neighbourhood of the centre-of-mass. So, in MIA we reward only Q_3's and Q_4's, which are at a distance of at most two squares of the centre-of-mass. For instance, Black has two Q_4's in Figure 3.6. In passing we note that this feature implicitly favours a material advantage.

The effect of implicitly favouring a component due to the introduction of another feature is first described by Schaeffer (1984) for chess. Obviously, it is a challenge to analyse the interrelationship in LOA too, since it turns out to be an issue for almost all components. A possible disadvantage of this feature is that if the position becomes too solid, its flexibility may decrease drastically. The mobility feature (see Subsection 3.1.5) may adjust this disadvantage.

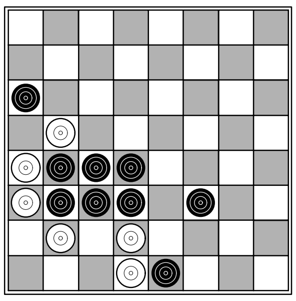

Figure 3.6: Position with two black Q_4's.

3.1.5 Mobility

The mobility feature is based on the basic principle of the same name. It looks at the potential of the moves in a position. The idea is that it is easier to connect your own pieces or obstruct the connection of opponent pieces if you have more and better moves. The feature was first implemented in MONA and YL.

When evaluating a position in MIA, the possible moves of both sides are generated (irrespective of who is to move). The moves are not rewarded equally. Experiments have shown that certain move types are to be preferred above others (see also Hashimoto *et al.*, 2003). Therefore, in MIA the following bonus/malus system is applied: the value of a capture move is doubled, the value of a move going to an edge or a move along an edge is halved. If a move belongs to multiple categories, the bonus/malus system is used multiple times. For example, let us assume that a regular move gets value 1, then a capture move gets value 2, a capture move going to an edge gets value 1, a capture move in an edge line going to a corner gets value 0.5. The computational requirements of this feature are not high. For each line configuration (represented as a bit vector) the mobility can be precomputed and stored in a table. During the search, the index scheme can be updated incrementally and in the evaluation function only a few table lookups have to be done.

An advantage of this feature that it is fast to evaluate. A disadvantage of this implementation is that it is too static. For example, all capture moves are given a bonus, even the ones which capture the last unconnected opponent's piece. Moreover, all edge moves are given a penalty, even if they connect to the main group. A more global look of the position would be needed to distinguish these kind of exceptions.

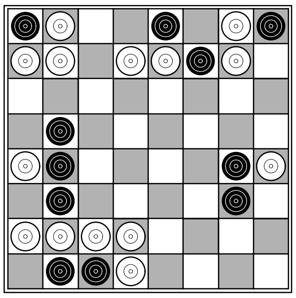

Figure 3.7: Position with walls.

3.1.6 Walls

The wall feature is based on the basic principle of blocking. Because a piece is not allowed to jump over the opponent's pieces, it can happen that the piece is blocked, i.e., cannot move. Blocking a piece far away from the other pieces is an effective way of preventing the opponent to win. Even partial blocking, called a wall (Handscomb, 2000a), can be quite effective, since it forces a player to find a way around the wall. Detecting whether a piece is (partially) blocked can be expensive as we have to know what the moves of the piece are and what its goal is.

In MIA we look only at walls that prevent the opponent's edge pieces from moving toward the centre. These walls are quite common and effective. The patterns can be precomputed and stored in a table. Using a bit-board representation they can be easily looked-up. We remark that we take special care of walls which block corner pieces.

For example, in Figure 3.7 the piece on **a4** is blocked in three ways going to the centre, whereas the piece on **h4** is only blocked in two centre directions. In the evaluator, we distinguish between walls which block two or three centre directions. The piece on **h8** is blocked only in two directions, but we evaluate this position as if it was blocked in 3 centre directions. The totally isolated piece on **a8** is evaluated as if there were two walls which both block the piece in three directions. The pieces on **b1** and **c1** are completely blocked, but we take only the two 3-centre-directions blocks into account. Thus, we only look at certain blocking patterns for edge pieces. It is a subject of future research to incorporate more of these patterns.

3.1.7 Connectedness

The connectedness feature is based on the basic principles of threats and solid formations. It measures the pairwise connections between the pieces. We reward positions with high connectedness; it is hoped that they eventually will be connected in one unit or will create threats to win.

In MIA we compute the average number of connections of a piece. In some evaluation functions the total number of connections is taken into account (e.g., YL), but this could implicitly be a material advantage. Any kind of material feature in LOA evaluation functions can be dangerous because the program might wildly capture pieces. This feature does not take into account whether a connection is important. To distinguish among connections, a global look at the board would be needed, which is time consuming. The number of connections for each side in each line configuration can be precomputed as is done with the mobility feature.

Of course the connectedness feature is highly correlated with the concentration feature and the quads feature. Though each has its own merits, these three features should be carefully tuned consequently.

3.1.8 Uniformity

The uniformity feature is based on the basic principle of solid formations. It is used to achieve a uniform distribution of the pieces (Chaunier and Handscomb, 2001) to counterbalance the negative effects of the centre-of-mass approach. It prevents that one or more pieces become too remote from the main group.

In MIA this is done in a way which is primitive but effective. The smallest rectangular area covering the distributed pieces is computed. The smaller the area is, the higher the reward is. An analogous implementation was first realised in the program YL (Billings and Björnsson, 2003).

3.1.9 Player to Move

The player-to-move feature is based on the basic principle of the initiative. It rewards the moving side. Having the initiative is mostly an advantage in LOA (Winands, 2000) like in many other games (Uiterwijk and Van den Herik, 2000).

Since MIA is using variable-depth search (because of the adaptive null move, the multi-cut, and quiescence search) not all leaf nodes are evaluated at the same depth. Therefore, leaf nodes in the search tree may have a different player to move, which is compensated in the evaluation function. This is done by giving a small bonus to the side to move.

3.1.10 Caching certain Features

It is possible in our evaluation function to cache computations of certain features, which can be used in other positions. For example, let us assume that we investigate the move **b8-c8** in Figure 3.2 and evaluate the resulting position. If we next investigate **b8-b7** we notice that certain properties of White's position remain the same (e.g., the number of pieces, centre-of-mass, the number of connections), whereas others can change (e.g., moves, blockades). It is easy to see that we do not have to compute the concentration, centralisation, centre-of-mass position, quads, connectedness, and uniformity for White again. Evaluation of these six features, which are independent of the position of the other side, are stored in an evaluation cache table. In the current evaluation function this gives a speed-up of at least 60 percent in the number of nodes investigated per second.

The pseudo code of the evaluation function using caching is given in Figure 3.8.

```
...........................................................
//Compute evaluation score
//Are the independent features cached for Black?
independent_features_Black = lookUpCacheTable(Hash_Black);

if(independent_features_Black == NO_VALUE){
    //Compute Black's independent features
    independent_features_Black = compConcentration(comBlack) +
                                 compCentralisation() +
                                 lookUpCOMPosition(comBlack) +
                                 compQuads() +
                                 compUniformity() +
                                 lookUpConnection(index_Black);
    //Store Black's independent features value in the cache table
    storeCacheTable(Hash_Black, independent_features_Black);
 }
//Are the independent features cached for White?
independent_features_White = lookUpCacheTable(Hash_White);

if(independent_features_White == NO_VALUE){
    //Compute White's independent features
    independent_features_White = compConcentration(comWhite)+
                                 compCentralisation()+
                                 lookUpCOMPosition(comWhite) +
                                 compQuads() +
                                 compUniformity() +
                                 lookUpConnection(index_White);
    //Store White's independent features value in the cache table
    storeCacheTable(Hash_White, independent_features_White);
 }
//Compute evaluation score
 eval = independent_features_Black - independent_features_White +
        lookupMobility(index) +
        lookupWalls(index) +
        player_to_move +
        randomValue(); //Random factor
...........................................................
```

Figure 3.8: Pseudo code for evaluation function.

3.2 Experiments

In order to quantify the improvements of the evaluation function, we played a round-robin tournament in which evaluators from earlier tournament versions of the program participated. All evaluators used the original search engine, described in Section 2.6. The evaluators are explained in Subsection 3.2.1. The results are described in Subsection 3.2.2.

3.2.1 Benchmark Evaluators

The benchmark evaluation functions[2] are described below.

Evaluator: MIA I The core of this evaluation function is the centre-of-mass approach. The quads feature is also implemented. Pieces at the edge are given a negative bonus (*edge-penalty* feature). In this version, a *centralised centre-of-mass position* was slightly more preferred (Winands *et al.*, 2001a). The idea was to prevent formations from being built on the edges, where they are more easily destroyed or blocked. The weights of the features were carefully hand-tuned. In retrospect this evaluator was primitive, although it won a game against both MONA and YL at the Fifth Computer Olympiad (Björnsson, 2000).

Evaluator: MIA II The major change of this evaluation function compared to the previous one is the introduction of a *primitive mobility* feature. There is no discrimination in rewarding different move types. In this evaluator, pieces at a corner edge are punished more severely. Using this evaluator, the tournament program shared the first place with YL in the regular tournament at the CMG Sixth Computer Olympiad. The play-off match was won by YL (Björnsson and Winands, 2001).

Evaluator: MIA III This evaluation function is enhanced with the wall feature. An *absolute centralisation* feature replaced the edge-penalty feature. The difference with the centralisation feature, described in Subsection 3.1.2, is that it computes the sum of piece values. A bonus is given for the player to move. The major improvement was retuning all the weights by using TD-learning (Winands *et al.*, 2002). There were three major changes in the weights. First, the initial weight of the dominating centre-of-mass was decreased to one tenth of its original value, indicating that we had wrongly interpreted the precise value of this feature. Second, the weight for the centralised centre-of-mass feature changed its sign, which means that opposite to expectations it is good to have the centre-of-mass closer to the edge instead of in the centre. Third, the weight of the absolute centralisation feature increased the most, indicating that we had underestimated the importance of this feature. Using this evaluator the tournament program finished second at the Seventh Computer Olympiad (scoring 1.5 points out of 4 against the much improved winner YL) (Björnsson and Winands, 2002). An exhibition match was played against MONA during the *Third International Conference on Computers and Games 2002 (CG'02)*, which ended in a 2-2 tie (Billings and Björnsson, 2002).

Evaluator: MIA IV This evaluation function incorporates all features as described in Section 3.1. The centre-of-mass position, wall, and player-to-move features used the same weights as the ones in MIA III. All the weights of the other features were basically found by using TD-learning. Some of them were adjusted by hand afterwards. Using this evaluator the tournament program won the Eighth Computer Olympiad (Winands, 2003) and the Ninth Computer Olympiad (see Appendix A).

An overview of the separate features as used in the four evaluators is given in Table 3.1. Four of them are not used anymore in the latest evaluator MIA IV. The edge penalty and absolute centralisation are integrated in the centralisation feature. Cen-

[2]Here, we would like to recall the footnote of Section 1.4 on the program MIA and the evaluators MIA I, MIA II, MIA III, and MIA IV.

tralised centre-of-mass was replaced after applying TD-learning. Primitive mobility was replaced after experiments had revealed that a mobility feature distinguishing move types was significantly better. Here, we note that the weights and details of the features may differ between different evaluators. We will not elaborate on the specific configuration of the weights. Using TD-learning we found several weight configurations, which performed more or less equally well. Finally, we reiterate that it is easier to correct a weight than to discover a missing feature when improving an evaluation function. Therefore, we keep our focus on the nine features discussed in Section 3.1 in the remainder of the chapter.

Table 3.1: Overview of the features.

	MIA I	MIA II	MIA III	MIA IV
Concentration	X	X	X	X
Edge Penalty	X	X		
Absolute Centralisation			X	
Centralisation				X
Centralised Centre-of-mass Position	X	X		
Centre-of-mass Position			X	X
Quads	X	X	X	X
Primitive Mobility		X	X	
Mobility				X
Walls			X	X
Connectedness				X
Uniformity				X
Player to Move			X	X

3.2.2 Results

The evaluators, previously described, competed with each other in a round-robin tournament. Ten different start positions, given in Figure 3.9, were used to increase the variety of play. The positions appear often after two ply in games at a strong level. For each start position, 100 games (50 with Black and 50 with White) were played. Thus, in total 1000 games were performed for each pair of evaluators.

To prevent that programs played the same games over and over again, a sufficiently large random factor (i.e., $[-100, +100]$ points[3]) was included in each evaluation function.

Fixed-depth searches were used as time control instead of time. At first sight it may look as if we are favouring the more advanced evaluators (i.e., they are time intensive because of the extra knowledge). However, there are two factors that counterbalance this potential favouring. First, the difference in speed is quite moderate. The program runs only 15 per cent slower with the MIA IV evaluator

[3]100 points equals more or less the heuristic advantage of the first player in the initial position.

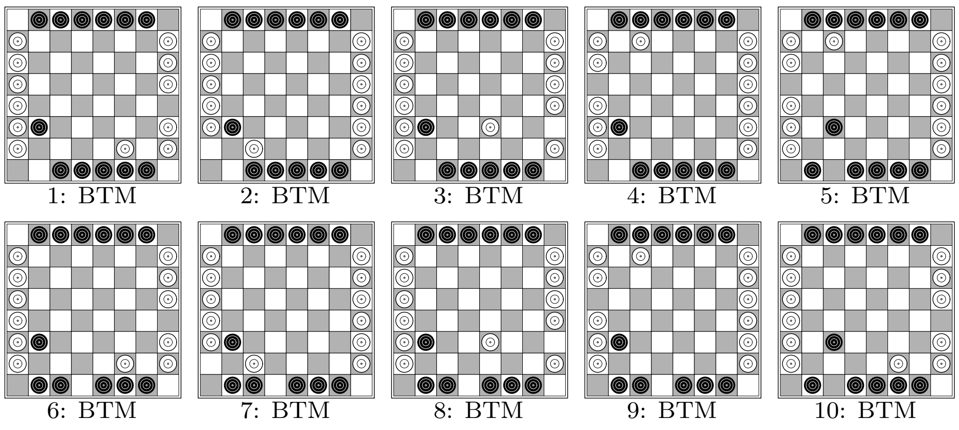

Figure 3.9: Start positions for the benchmark-evaluator experiments.

than with the MIA I evaluator. All the evaluators have to compute the average distance to the centre-of-mass and the quads, these are time-consuming elements. Most other additions are relatively cheap. Second, when an evaluator is a good predictor of the position, a best move found at a shallow search is more likely to remain good and it may therefore cause cut-offs at deeper searches. For example, when the MIA I evaluator is used in the original search engine it searches 75 per cent more nodes compared to the MIA IV evaluator. The advantage of fixing the depth is that we can measure the influence of increasing the depth.

In Table 3.2 the results of the tournaments are given for searches to depth 4, 6, 8, and 10, respectively. In each tournament a total of 6000 games is played. It is easy to see that an increase of version number is accompanied by an increase in playing strength. On the one hand, we see that search depth has an influence on the play of the weak MIA I. It performs worse against the other evaluators when both programs are searching more deeply. A reason might be that a deep search is not able to compensate the lack of knowledge of MIA I, while the search depth exploits more of the potential of MIA II, III and IV. On the other hand, MIA II, III and IV perform approximately the same against each other at each depth. Although MIA II's only major improvement is a primitive mobility feature, it outperforms MIA I, but it even plays much better against MIA III and IV than MIA I does. The increase of strength from MIA II to MIA III is not that large compared to the increase from MIA I to MIA II. MIA II has a winning percentage of 90 per cent against MIA I at depth 10, whereas MIA III has only a winning percentage of 65 per cent against MIA II. Even worse, at deep searches MIA II performs better against MIA I than MIA III does. But, MIA III performs much better against the sophisticated MIA IV than MIA II does. A possible explanation is that the addition of the wall feature has a negative effect against primitive evaluators and does only work against sophisticated evaluators. MIA IV defeats the previous evaluators of MIA by a fair distance. Even the strong MIA III is not able to score more than 20 to 25 per cent of the points against MIA IV.

Table 3.2: Tournament results at various depths.

Tournament results at depth 4.

Evaluator	MIA I	MIA II	MIA III	MIA IV
MIA I	-	259	199	71.5
MIA II	741	-	373	163.5
MIA III	801	627	-	248.5
MIA IV	928.5	836.5	751.5	-

Tournament results at depth 6.

Evaluator	MIA I	MIA II	MIA III	MIA IV
MIA I	-	188	168.5	51
MIA II	812	-	356	174
MIA III	831.5	644	-	223.5
MIA IV	949	826	776.5	-

Tournament results at depth 8.

Evaluator	MIA I	MIA II	MIA III	MIA IV
MIA I	-	137	159.5	41.5
MIA II	863	-	360	129
MIA III	840.5	640	-	205
MIA IV	958.5	871	795	-

Tournament results at depth 10.

Evaluator	MIA I	MIA II	MIA III	MIA IV
MIA I	-	97.5	137.5	44.5
MIA II	902.5	-	359.5	121.5
MIA III	862.5	640.5	-	234.5
MIA IV	955.5	878.5	765.5	-

3.3 Evaluation of Results

In this section we give an evaluation of the importance of the features used in the evaluation function of MIA IV. Thereafter we will provide a schematic overview of the feature space for the nine features and review the interdependencies of the features. Our findings are based on experiments performed in the phase of fine-tuning the evaluation-function features for the Computer Olympiads, and on our experience with several versions of MIA when playing games. We deal with them in the order we perceive now as the best order, i.e., enumerated from the most important feature down to the feature with the weakest impact.

Obviously, the concentration feature is the dominant feature. It contributes to the essence of the goal: connect all pieces. It was present in all evaluation functions of MIA, right from the beginning, and as evidenced by our experiments gave by far the largest improvement in playing strength.

The importance of the mobility of the pieces was not taken into account in MIA I. However, it appeared to be crucial to be included. In first instance we did so as a primitive component (disregarding move types), but in the later evaluation functions its effect appeared to be even more important when the bonuses for the moves were distinguished according to their type. Apart from the concentration feature, mobility dominates all other features.

After mobility, the quads feature is the most important one. The quads feature has proved to be a very effective way to stimulate reaching solid positions, from which the connection goal can be reached. We claim to have been the first to discover the importance of the feature for LOA (Winands, 2000).

Thereafter follows on the fourth place, centralisation. It is a simple but important addition to the concentration feature, encouraging the control of the centre of the board.

The next three most important features are difficult to arrange in the right order. They are uniformity, connectedness, and walls. According to our experience they are more or less equally important, but clearly dominate the remaining two features. All three can be seen as corrections or additions to the two dominating features, i.e., concentration and mobility.

The centre-of-mass position feature comes at the eight position. It stimulates that building of the main block happens closer to the edge, where it is less vulnerable to opponent attacks. The effects are small, but sometimes significant.

The least important feature is the bonus for the player to move. Its effect is small, but non-negligible.

To summarise the importance of the features by the effects on the playing strength of each feature, the decreasing order is as follows:

> *concentration* > *mobility* > *quads* > *centralisation* > {*uniformity*, *connectedness*, *walls*} > *centre-of-mass position* > *player to move*

In Figure 3.10 we sketch the feature space in the form of a qualitative probabilistic network (Wellman, 1990) for the nine features of MIA IV. By arrows we denote the interdependencies. A '+' label means that a higher value for the source feature will correspond with large probability to a higher value of the destination feature, i.e., a positive dependency relation. Likewise a '–' denotes a negative dependency relation. We will briefly elucidate Figure 3.10 by describing the twelve interdependencies from left to right.

It is easy to see that the centralisation feature and centre-of-mass position feature have a negative dependency. The centralisation aims to put the pieces in the centre of the board, whereas the centre-of-mass position feature aims to build a main group outside the centre. However, both the centralisation feature and centre-of-mass position contribute to the concentration. If the pieces are in the centre, they will

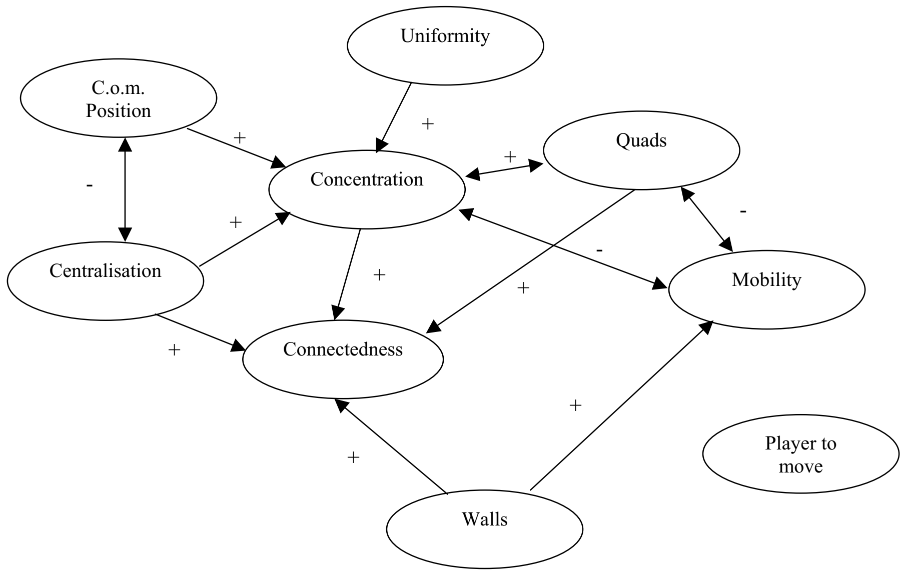

Figure 3.10: The feature space.

be concentrated. Moreover, if the centre-of-mass is at the edge, the pieces will be concentrated too.

The uniformity feature contributes to the concentration feature. For instance, moving an outlier one square closer to the centre-of-mass increases not only the uniformity value but also the concentration value. Concentration does not necessarily contribute to uniformity. There can still be outliers in a position with high concentration.

The concentration and quads feature have a positive dependency on each other. Both the concentration and quads feature aim at grouping pieces together, where concentration has the largest impact.

If the concentration is high, the pieces will be grouped in one main group, and more pieces will be connected with each other. However, if the number of connections is high, it does not mean they are concentrated into one main group.

Quads and walls contribute to the connectedness feature. They are both implicitly favouring connections. But the connectedness feature does not necessarily contribute to them. It is possible to build a position with high connectedness but with a low quads-value and a low walls-feature value. If quads are built far away from the centre-of-mass, they will not be rewarded by the quads feature. If a wall does not block a piece, it will not receive a reward too.

Mobility and quads have a negative dependency. If the position becomes too solid, its flexibility and therefore mobility may decrease drastically. Mobility can prevent building strong unbreakable positions. In the same way mobility and concentration have a negative dependency.

Walls are a way to decrease the opponent's mobility and therefore relatively increase the player's own mobility. But, a high mobility does not mean that the opponent's pieces are blocked by walls.

The player-to-move is a correcting factor which does neither influence nor is influenced by other features.

There exist much interconnectedness and overlap between the features, which influence their performance. Therefore, all the features have to be simultaneously fine-tuned (or heuristically optimised) in a careful way.

3.4 Chapter Conclusion and Future Research

In this chapter we tested the strength of the various evaluators of MIA which all performed respectably at the Computer Olympiads. Many features in the evaluation function do not consume much time. By using precomputed tables and caching, most of them are quite straightforward to evaluate.

The most important feature is concentration, followed by the mobility feature. All features are essential and contribute to the playing strength. As evidenced by observation we mention that MIA IV defeated all older evaluators by large margins. There exist much interconnectedness and overlap between the features, which influence their performance. Therefore, all the features have to be simultaneously fine-tuned (or heuristically optimised) in a careful way. We conclude that the combination of the nine features mentioned in Section 3.1 has resulted in an evaluation function that significantly increased the playing strength of our LOA program compared to programs with less-sophisticated evaluation functions. With the present evaluator we gained first places at the 8^{th} and 9^{th} Computer Olympiad.

There are still many possibilities to improve the evaluator. We mention for instance, more patterns of blocked pieces, better distinction of move types in the mobility component, and additional knowledge whether a connection is important. Moreover, there is room to fine-tune certain weights and parameters in the evaluation function. Finally, a topic of future research is switching off one feature, retuning the weights and investigating whether the playing strength increases or decreases (cf. Schaeffer, 1986).

In Chapters 2 and 3 we have addressed the *terminal knowledge* part of the informed search. In the next chapters we will concentrate on the various aspects of the search engine of MIA, including the *directing knowledge* used to steer the search process.

Chapter 4

Proof-Number Search Algorithms

This chapter is an updated and abridged version of the following three publications:

1. Winands, M.H.M. and Uiterwijk, J.W.H.M. (2001). PN, PN^2 and PN* in Lines of Action. *The CMG Sixth Computer Olympiad Computer-Games Workshop Proceedings* (ed. J.W.H.M. Uiterwijk), Technical Reports in Computer Science CS 01-04, Universiteit Maastricht, Maastricht, The Netherlands.
2. Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001b). Combining Proof-Number Search with Alpha-Beta Search. *Proceedings of the Thirteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)* (eds. B. Kröse, M. de Rijke, G. Schreiber, and M. van Someren), pp. 299–306, Universiteit van Amsterdam, Amsterdam, The Netherlands.
3. Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003c). PDS-PN: A New Proof-Number Search Algorithm: Application to Lines of Action. *Computers and Games, Lecture Notes in Computer Science 2883* (eds. M. Müller, J. Schaeffer, and Y. Björnsson), pp. 170–185, Springer-Verlag, Berlin, Germany.[1]

The second research question deals with proof-number search algorithms. In this chapter we will investigate several PN-search algorithms. They can be applied in two different ways: offline and online. First, we will concentrate on the *offline* application of the PN-search algorithms. The number of positions they can solve (i.e., the post-mortem analysis quality) is tested on a set of endgame positions. Moreover, we will investigate to what extent the algorithms are restricted by their working memory *or* by the search speed. Our findings will be used when answering

[1]The author is grateful to Springer-Verlag for the permission of reusing relevant parts of the article in this thesis.

the second research question in the next chapter. Then, we will briefly investigate the *online* application of PN search. In particular, the real-time application of PN search during a game is examined.

The chapter is organised as follows. In Section 4.1 we discuss the need for special algorithms to solve endgame positions. Section 4.2 describes PN, PN^2, and depth-first variants of PN search. In Section 4.3 we examine the offline solution power and the solution time, in relation to that of $\alpha\beta$. Then, we focus on the performance of online PN search in Section 4.4. Finally, in Section 4.5 we present our conclusion and propose future research.

4.1 Endgame Solvers

Most modern game-playing computer programs successfully use $\alpha\beta$ search (Knuth and Moore, 1975) with enhancements for online game-playing (Campbell, Hoane, and Hsu, 2002). However, the enhanced $\alpha\beta$ search is sometimes not sufficient to play well in the endgame. In some games, such as Chess, this problem is solved by the use of endgame databases (Nalimov, Haworth, and Heinz, 2000). Due to memory constraints this is only feasible for endgames with a relatively small state-space complexity, although nowadays the size may be considerable. An alternative approach is the use of a specialised binary (win or non-win) search method, such as proof-number (PN) search (Allis *et al.*, 1994). The latter method is inspired by the conspiracy-number algorithm (McAllester, 1988; Schaeffer, 1990).

In many domains PN search outperforms $\alpha\beta$ search in proving the game-theoretic value of endgame positions. The PN-search idea is a heuristic, which prefers expanding shallow subtrees over wide ones. PN search or a variant thereof has been successfully applied to the endgame of Awari (Allis *et al.*, 1994), Chess (Breuker, Allis, and Van den Herik, 1994a), Checkers (Schaeffer and Lake, 1996), Shogi (Seo, Iida, and Uiterwijk, 2001), and Go (Kishimoto and Müller, 2003a). Since PN search is a best-first search, it has to store the whole search tree in memory. When the memory is full, the search has to end prematurely.

To overcome this problem PN^2 was proposed in Allis (1994), as an algorithm to reduce memory requirements in PN search. It is elaborated upon in Breuker (1998). Its implementation and testing for chess positions is extensively described in Breuker, Uiterwijk, and Van den Herik (2001b). PN^2 performs two levels of PN search, one at the root and one at the leaves of the first level. As in the B* algorithm (Berliner, 1979), a search process is started at the leaves to obtain a more accurate evaluation. Although it uses far less memory than PN search, it is still a best-first search algorithm with the disadvantage that the search can end prematurely because of memory exhaustion.

Recently, the idea behind the MTD(f) algorithm (Plaat *et al.*, 1996) is successfully applied in PN variants: try to construct a depth-first algorithm that behaves as its corresponding best-first search algorithm. In 1995, Seo formulated a depth-first iterative-deepening version of PN search, later called PN* (Seo *et al.*, 2001). The advantage of this variant is that there is no need to store the whole tree in memory. The disadvantage is that PN* is slower than PN (Sakuta and Iida, 2001).

Other depth-first variants are PDS (Nagai, 1998) and df-pn (Nagai and Imai, 1999). Although their generation of nodes is even slower than PN*'s, they are building smaller search trees. Hence, they are in general more efficient than PN*.

4.2 Five Proof-Number Search Algorithms

In this section we give a short description of PN search (Subsection 4.2.1), PN^2 search (Subsection 4.2.2) and three depth-first variants of PN search (Subsection 4.2.3).

4.2.1 Proof-Number Search

Proof-number (PN) search is a best-first search algorithm especially suited for finding the game-theoretical value in game trees (Allis, 1994). Its aim is to prove the true value of the root of a tree. A tree can have three values: *true*, *false*, or *unknown*. In the case of a forced win, the tree is *proved* and its value is true. In the case of a forced loss or draw, the tree is *disproved* and its value is false. Otherwise the value of the tree is unknown. In contrast to other best-first algorithms PN search does not need a domain-dependent heuristic evaluation function to determine the most-promising node to be expanded next (Allis *et al.*, 1994). In PN search this node is usually called the *most-proving* node. PN search selects the most-proving node using two criteria: (1) the shape of the search tree (the branching factor of every internal node) and (2) the values of the leaves. These two criteria enable PN search to treat game trees with a non-uniform branching factor efficiently.

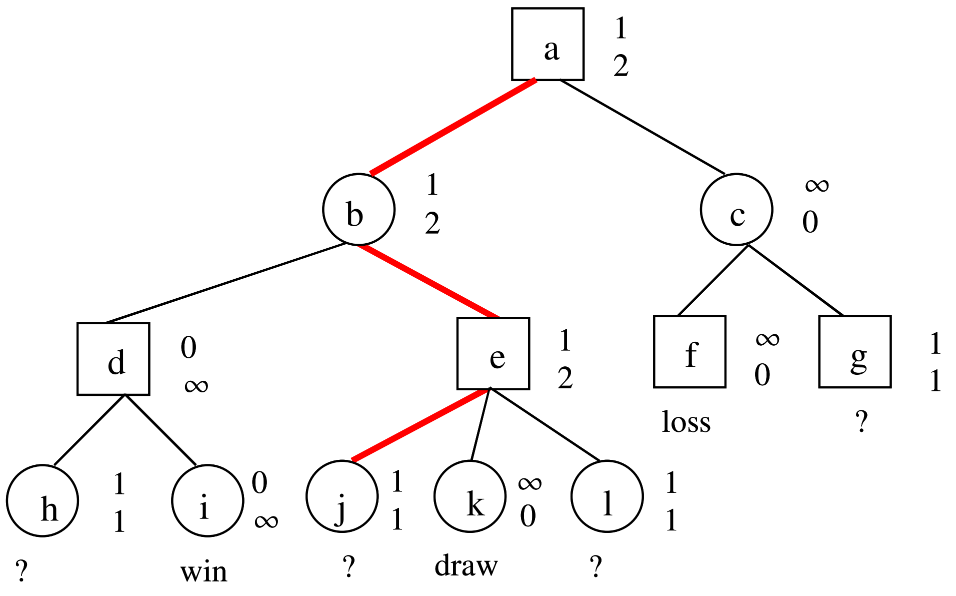

Figure 4.1: An AND/OR tree with proof and disproof numbers.

Below we explain PN search on the basis of the AND/OR tree depicted in Figure 4.1, in which a square denotes an OR node, and a circle denotes an AND node. The numbers to the right of a node denote the proof number (upper) and disproof number (lower). A *proof number* represents the minimum number of leaf nodes which have to be proved in order to prove the node. Analogously, a *disproof number* represents the minimum number of leaf nodes which have to be disproved in order to disprove the node. Because the goal of the tree is to prove a forced win, winning nodes are regarded as proved. So, they have proof number 0 and disproof number ∞ (e.g., node i). Lost or drawn nodes are regarded as disproved (e.g., nodes f and k). They have proof number ∞ and disproof number 0. Unknown leaf nodes have a proof and disproof number of unity (e.g., nodes g, h, j and l). The proof number of an internal AND node is equal to the sum of its children's proof numbers, since to prove an AND node all the children have to be proved. The disproof number of an AND node is equal to the minimum of its children's disproof numbers, since to disprove an AND node it suffices to disprove one child. The proof number of an internal OR node is equal to the minimum of its children's proof numbers, since to prove an OR node it suffices to prove one child. The disproof number of an internal OR node is equal to the sum of its children's disproof numbers, since to disprove an OR node all the children have to be disproved. The procedure of selecting the most-proving node to expand is as follows. We start at the root. Then, at each OR node the child with the lowest proof number is selected as successor, and at each AND node the child with the lowest disproof number is selected as successor. Finally, when a leaf node is reached, it is expanded and its children are evaluated. This is called *immediate evaluation*. The selection of the most-proving node (j) in Figure 4.1 is given by the bold path.

The number of node traversals to select the most-proving node can have a negative impact on the execution time. Therefore, Allis (1994) proposed the following small enhancement. The updating process can be terminated when the proof and disproof number of a node do not change. From this node we can start the next most-proving node selection. For an adequate description of implementation details we refer to Allis *et al.* (1994) and Appendix B.1, where the essentials for implementation are given.

In the naive implementation, proof and disproof numbers are each initialised to unity in the unknown leaves. In other implementations, the proof number and disproof number are set to 1 and n for an OR node (and the reverse for an AND node), where n is the number of legal moves. In LOA this would mean that we take the *mobility* of the moving player in the position into account. As we have seen in the previous chapter this is an important feature. The effect of this enhancement is tested in Subsection 4.3.1.

Here we reiterate that a disadvantage of PN search is that the whole search tree has to be stored in memory. When the memory is full, the search process has to be terminated prematurely. A partial solution is to delete proved or disproved subtrees (Allis, 1994). In the next subsections we discuss two main variants of PN search that handle the memory problem more adequately.

4.2.2 PN^2 Search

For an adequate description we repeat a few sentences of our own. PN^2 is first described in Allis (1994), as an algorithm to reduce memory requirements in PN search. It is elaborated upon in Breuker (1998). Its implementation and testing for chess positions is extensively described in Breuker *et al.* (2001b). PN^2 consists of two levels of PN search. The first level consists of a PN search (pn_1), which calls a PN search at the second level (pn_2) for an evaluation of the most-proving node of the pn$_1$-search tree. This pn$_2$ search is bound by a maximum number of nodes to be stored in memory. The number is a fraction of the size of the pn$_1$-search tree. The fraction *f(x)* is given by the logistic-growth function (Berkey, 1988), x being the size of the first-level search:

$$f(x) = \frac{1}{1 + e^{\frac{a-x}{b}}} \tag{4.1}$$

with parameters a and b, both strictly positive. The number of nodes y in a pn$_2$-search tree is restricted to the minimum of this fraction function and the number of nodes which can still be stored. The formula to compute y is:

$$y = min(x \times f(x), N - x) \tag{4.2}$$

with N the maximum number of nodes to be stored in memory.

The pn$_2$ search is stopped when the number of nodes stored in memory exceeds y or the subtree is (dis)proved. After completion of the pn$_2$ search, the children of the root of the pn$_2$-search tree are preserved, but subtrees are removed from memory. The children of the most-proving node (the root of the pn$_2$-search tree) are not immediately evaluated by a second-level search, only when they are selected as most-proving node. This is called *delayed evaluation*. We remark that for pn$_2$-search trees immediate evaluation is used. The essentials of our implementation are given in Appendix B.2.

As we have seen in Subsection 4.2.1, proved or disproved subtrees can be deleted. If we do not delete proved or disproved subtrees in the pn$_2$ search the number of nodes searched is the same as y, otherwise we can continue the search longer. The effect of *deleting (dis)proved pn$_2$ subtrees* is tested in Subsection 4.3.1.

4.2.3 Three Depth-First Proof-Number Search Algorithms

Recently, three depth-first PN variants have been proposed, which solved the memory problem of PN-search algorithms.

In 1995, Seo formulated the first depth-first iterative-deepening version of PN search, later called PN* (Seo *et al.*, 2001). PN* uses a method called *multiple-iterative deepening*. Instead of iterating only at the root node such as in the ordinary iterative deepening, it iterates also at AND nodes. To each AND node a threshold is given. The subtree rooted at that node is continued to be searched as long as the proof number is below the assigned threshold. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a transposition table.

The disadvantage of PN* is that it has difficulties to disprove a (sub)tree, which harms its solving performance (Sakuta and Iida, 2001). Nagai (1998, 1999) proposed a second depth-first search algorithm, called Proof-number and Disproof-number Search (PDS), which is a straight extension of PN*. Instead of using only proof numbers such as in PN*, PDS uses disproof numbers too.[2] Moreover PDS uses *multiple-iterative deepening* in *every* node. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a *TwoBig* transposition table (Breuker *et al.*, 1996). PDS uses two thresholds in searching, one for the proof numbers and one for the disproof numbers. We note that PDS suffers from the graph-history interaction (GHI) problem (cf. Breuker *et al.*, 2001a). In the present implementation this problem is ignored (Nagai, 1999). In Chapter 5 we will describe PDS in detail. For essentials of the implementation we refer to Appendix B.3.

Recently, Nagai (2002) has introduced a third depth-first PN algorithm, called df-pn (depth-first proof-number search). It is mainly a variant of PDS. The algorithm df-pn does not perform iterative deepening at the root node. Instead it sets the thresholds of both proof number and disproof number at the root node to a large value. As the search goes more deeply, the threshold values are distributed among the descendant nodes. Contrary to PDS, it has been proved that df-pn always selects the most-proving node. It turns out that df-pn suffers more from the GHI problem than PDS. It has a fundamental problem when applied to a domain with repetitions (Kishimoto and Müller, 2003a). Solutions to the GHI problem have recently been proposed (Nagai, 2002; Kishimoto and Müller, 2003b). Although df-pn sometimes solves positions faster than PDS, it solves in practice fewer positions. Experiments have shown that PDS is superior to df-pn (Sakuta, 2001). Therefore, we have chosen not to concentrate on the df-pn algorithm.

4.3 Offline PN Search

In this section we test the offline performance of three PN-search variants. We are making a comparison between PN, PN^2, PDS and $\alpha\beta$. For the $\alpha\beta$ depth-first iterative-deepening search, nodes at depth i are counted only during the first iteration that the level is reached. This is how analogous comparisons are done in Allis (1994). For PN, PN^2 and PDS search, all nodes evaluated for the termination condition during the search are counted. For PDS this node count is equal to the number of expanded nodes (function calls of the recursive PDS algorithm); for PN and PN^2, this node count is equal to the number of nodes generated. The maximum number of nodes searched is 50,000,000. The limit corresponds roughly to tournament conditions (i.e., 180 seconds per move). The maximum number of nodes stored in memory is 1,000,000. The parameters (a,b) of the growth function used in PN^2 are set at (1800K, 240K) according to the suggestions in Breuker *et al.* (2001b). First, we test the mobility enhancement in PN and PN^2; and the effect of deleting (dis)proved subtrees at the pn$_2$ search of the PN^2 in Subsection 4.3.1. Then we compare PN, PN^2, PDS and $\alpha\beta$ search with each other in Subsection 4.3.2.

[2]We recall that PN and PN^2 use disproof numbers too.

4.3.1 Two Enhancements of PN and PN^2

We will first test the effect of enhancing PN and PN^2 with mobility in one experiment and then of deleting (dis)proved pn$_2$ subtrees in another experiment.

In the first experiment, we tested PN search and PN^2 with the mobility enhancements on a test set of 116 positions.[3] We used the enhancement of deleting of (dis)proved pn$_2$ subtrees for PN^2. The results are shown in Table 4.1. In the second column we see that PN search solved 85 positions using mobility; without mobility it solved 53 positions. PN^2 search using mobility solved 109 positions and without it solved only 91 positions. Next, in the third column we see that on a set of 53 positions that both PN algorithms are able to solve, PN search using mobility is roughly 5 times faster in nodes than PN search without mobility. Finally, in the fourth column we see that on a set of 91 positions that both PN^2 algorithms are able to solve, PN^2 search using mobility is more than 6 times faster in nodes than PN^2 search without using mobility. In general we may conclude that mobility speeds up the PN and PN^2 with a factor 5 to 6. The extra time spent on this extension is only 20 per cent. Owing to mobility PN search can solve many more positions, because the memory constraint is violated less frequently.

Table 4.1: Mobility in PN and PN^2.

Algorithm	# of pos. solved	Total nodes (53 pos.)	Total nodes (91 pos.)
PN	53	24,357,832	-
PN + Mob.	85	5,053,630	-
PN^2	91	-	345,986,639
PN^2+ Mob.	109	-	56,809,635

In the second experiment, we tested the effect of deleting (dis)proved subtrees at the pn$_2$ search of the PN^2. The results are shown in Table 4.2. Both variants (not deleting pn$_2$ subtrees and deleting pn$_2$ subtrees) used mobility in the experiment. On a set of 108 positions that both versions were able to solve, we can see that deleting (dis)proved subtrees improves the search with 10 per cent. It also solves one additional position.

Table 4.2: Deleting (dis)proved subtrees at the second-level search PN^2.

Algorithm	# of pos. solved	Total nodes (108 pos.)
PN^2 not deleting pn$_2$ subtrees	108	463,076,682
PN^2 deleting pn$_2$ subtrees	109	416,168,419

In the remainder of this thesis we will use these two enhancements (i.e., mobility and deleting (dis)proved pn$_2$ subtrees), for PN and PN^2.

[3]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/tswin116.zip.

4.3.2 Comparison

In this subsection we compare PN, PN^2, PDS and $\alpha\beta$ search with each other. The goal is to investigate the effectiveness of the PN-search variants by experiments. We will look how many endgame positions they can solve and how much effort (in nodes and CPU time) they take. PN, PN^2, PDS and $\alpha\beta$ are tested on a set of 488 forced-win LOA positions.[4] Two comparisons are made, they are described below.

First Comparison

In the second column of Table 4.3 we see that 470 positions were solved by the PN^2 search, 473 positions by PDS, only 356 positions by PN, and 383 positions by $\alpha\beta$. In the third and fourth column the number of nodes and the time consumed are given for the subset of 314 positions, which all four algorithms were able to solve. If we have a look at the third column, we see that PN search builds the smallest search trees and $\alpha\beta$ by far the largest. PN^2 and PDS build larger trees than PN but can solve significantly more positions. This suggests that both algorithms are better suited for harder problems. PN^2 investigates 1.2 times more nodes than PDS, but PN^2 is (more than) six times faster than PDS in CPU time for this subset.

Table 4.3: Comparing the search algorithms on 488 test positions.

Algorithm	# of positions solved (out of 488)	314 positions	
		Total nodes	Total time (ms.)
$\alpha\beta$	383	1,711,578,143	22,172,320
PN	356	89,863,783	830,367
PN^2	470	139,254,823	1,117,707
PDS	473	118,316,534	6,937,581

From the experiments we may draw the following three conclusions.

1. PN-search algorithms clearly outperform $\alpha\beta$ in solving endgame positions in LOA.
2. The memory problems make the plain PN search a weaker solver for the harder problems.
3. PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$.

Second Comparison

For a better insight how much faster PN^2 is than PDS in CPU time, we did a second comparison. In Table 4.4 we compare PN^2 and PDS on the subset of 463 test positions, which both algorithms were able to solve. Now, PN^2 searches 2.6 times more nodes than PDS. The reason for the decrease of performance is that for hard

[4]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/tscg2002a.zip.

problems the pn_2-search tree becomes as large as the pn_1-search tree. Therefore, the pn_2-search tree is causing more overhead. However, if we have a look at the CPU times we see that PN^2 is still three times faster than PDS. The reason is that PDS has a relatively large time overhead because of the delayed evaluation (see the next chapter). Consequently, the number of nodes generated is higher than the number of nodes expanded. In our experiments, we observed that PDS generated nodes 7 to 8 times slower than PN. Such a figure for the overhead is in agreement with experiments performed in Othello and Tsume-Shogi (Sakuta and Iida, 2001). We remark that Nagai's (1999) Othello results showed that PDS was better than PN search, (i.e., it solved the positions faster than PN). Nagai assigned to both the proof number and the disproof number of unknown nodes a 1 in his PN search and therefore did not use the mobility enhancement. In contrast, we incorporated the mobility in the initialisation of the proof numbers and disproof numbers in our PN search. We believe that comparing PDS with a PN-search algorithm without using mobility component is not fair. Since PDS does not store unexpanded nodes which have a proof number 1 and disproof number 1, it can be said that PDS already initialises the proof number and disproof number by using the number of its children (Nagai, 1999). The mobility enhancement is already implicitly incorporated in the PDS search.

Table 4.4: Comparing PDS and PN^2 on 463 test positions.

Algorithm	Total nodes	Total time (ms.)
PN^2	1,462,026,073	11,387,661
PDS	562,436,874	34,379,131

From this second experiment we may conclude that PDS is considerably slower than PN^2 in CPU time. Therefore, PN^2 seems to be a better endgame solver under tournament conditions. Counterbalancing this success, we note that PN^2 is still restricted by its working memory and is not fit for solving really hard problems.

4.4 Online PN Search

In the previous section we have seen that PN search is able to solve more instances of a set of endgame positions than $\alpha\beta$ search and that it does so faster than $\alpha\beta$ search. So, we may state that PN search has been applied successfully *offline* in our test domain. In this section we introduce a method of combining PN search and $\alpha\beta$ search *online* instead of *offline*. Moreover, the benefit of using a transposition table to reuse some knowledge, achieved during the PN search, in the $\alpha\beta$ search is examined. We remark that our method of combining perfect and heuristic evaluations is much simpler to apply than Beal's (1984) nested Minimax, using heuristic values together with perfect upper and lower bounds. The organisation of this section is as follows. The combination of PN and $\alpha\beta$, called PN-$\alpha\beta$, is described in Subsection 4.4.1. In Subsection 4.4.2 we describe the experimental design. The results of using PN-$\alpha\beta$ are presented and discussed in Subsection 4.4.3.

4.4.1 PN-$\alpha\beta$

The idea of combining PN search and $\alpha\beta$ search originates from Van der Meulen who used PN search in combination with $\alpha\beta$ search (Allis, 1994) in the Awari program LITHIDION (Allis, Van der Meulen, and Van den Herik, 1991b). In the opponent's time LITHIDION performed PN searches on its potential moves looking for wins. When the opponent selected a losing move, LITHIDION used the outcome of the PN search to select the winning move. The weakness of the method is that it is impossible to perform deep PN searches for *all* possible opponent moves in the opponent's time when the number of moves is large.

In this subsection, we introduce another way of combining the two search methods consecutively. We denote this combination by PN-$\alpha\beta$. PN is done in the player's own time and the information gained in the PN search is used in the $\alpha\beta$ search. PN-$\alpha\beta$ works as follows. In endgame positions PN search is applied for some fixed fraction of the allotted time for a move. Here, an endgame position is defined as a position occurring in a game after a fixed number of moves. The PN search terminates prematurely when either the position is proved or disproved, or the system runs out of memory. When the position can be proved, a winning move is played. It is possible that this move is not optimal (i.e., it will not lead to a shortest win).

During the PN search a position is stored if and only if it is a proven position. Nodes are stored in a proved-node transposition table (called PN-TT) by using the well-known Zobrist-hashing method (Zobrist, 1970). Disproved nodes are not stored because it is not known whether the game-theoretic value of the node is a loss or a draw. When two positions have the same hash index, the last examined position is preferred over early ones (replacement scheme *New*, see Breuker *et al.*, 1994b). The usual schemes *Big* or *Deep* are not used, because the number of nodes of a subtree or the depth of a subtree are not appropriate measures for replacement in case of PN search. If it is known that draws are impossible in a game (such as Hex), then it is possible to store also the disproved nodes in a separate table and to use them in an analogous way as proved nodes in $\alpha\beta$.

If the PN-search algorithm is not able to find a winning move, the $\alpha\beta$-search algorithm is applied for the remaining time. Whenever a position is examined in the $\alpha\beta$ search, the position is looked up in the PN-TT for a possible cut-off. Between the moves the transposition table is not cleared because the stored positions may still be useful during the game. The number of moves played defining an endgame position and the fraction of time allotted to PN search are controllable parameters to be fine-tuned.

4.4.2 Experimental Design

One of the big challenges in PN search is to find out when to apply it during a real game. On the one hand, if PN search is used too soon in a game, it will have a negative effect on the outcome of the game because time has been wasted. On the other hand, if PN search is applied too late, it will have no positive effect at all. After some experiments we found that the best strategy to enable the PN-$\alpha\beta$ in MIA was to exploit it in positions after 17 moves (34 plies) by using PN search a quarter of the time allotted for a move. Since the time was limited to 30 seconds per move, it means that PN search is trying to solve the position in 7.5 seconds when to move. In that amount of time the memory will not completely be filled. Regarding the experiments we only took into account games which lasted longer than 34 plies. Drawn games were also not taken into account. Five series of experiments have been run. The first experiment estimated the proportion of wins by Black and White when both sides used plain $\alpha\beta$ search in a series of 300 games. This proportion was taken as a reference for the other experiments. The second and third experiment measured the advantage of PN-$\alpha\beta$ over plain $\alpha\beta$ for each colour. The fourth and fifth experiment measured the advantage of PN-$\alpha\beta$ without using PN-TT over plain $\alpha\beta$ for each side.

4.4.3 Experimental Results

Table 4.5 provides the results for the five series of experiments. The third and fourth column give the absolute and relative game scores, respectively. The table shows that PN-$\alpha\beta$ search has an advantage over normal $\alpha\beta$ search, but only when information gathered in the PN search is reused by the transposition table. If we take the reference ratio of Black and White win percentages into account, we see that enabling PN-$\alpha\beta$ yields an increase in performance of some 6 per cent, irrespective of colour.

Table 4.5: Experimental results.

Black	White	Absolute	Relative
$\alpha\beta$	$\alpha\beta$	159-131	55-45
PN-$\alpha\beta$ with PN-TT	$\alpha\beta$	122-78	61-39
$\alpha\beta$	PN-$\alpha\beta$ with PN-TT	96-104	48-52
PN-$\alpha\beta$ without PN-TT	$\alpha\beta$	150-149	50-50
$\alpha\beta$	PN-$\alpha\beta$ without PN-TT	104-81	57-43

In these experiments using PN search even has a negative effect when the information is not reused. A potential reason is the following. If we have performed an unsuccessful PN search, $\alpha\beta$ cannot reach the usual search depth (approximately one ply) because of the time already used for the PN search. The PN-TT is making up for this in two ways. First, when a position occurs in the PN-TT it should not be explored or evaluated again. Despite of the time used for doing the PN search, it can even happen that $\alpha\beta$ can search more deeply than a normal search due to the PN-TT. Second, the heuristic error of the evaluation function is assumed to be less, which can result in other (better) move decisions.

Although the method is quite successful, we still have not solved the problem when to use PN search. Using PN search after a fixed number of moves is an artificial solution. Clearly, for each different opponent, the parameters have to be determined separately. Nevertheless, even then PN search will sometimes still be activated too late (or too early). In Table 4.6 we have selected from Table 4.5 only the games with over 45 plies. Although the set is too small for meaningful conclusions, it seems that applying PN search for too long has a negative effect. In other words, sometimes it is better to inactivate PN-$\alpha\beta$ in the very end of a game. In conclusion, it remains

Table 4.6: Experimental results for games with over 45 plies.

Black	White	absolute	relative
$\alpha\beta$	$\alpha\beta$	22-18	55-45
PN-$\alpha\beta$ with PN-TT	$\alpha\beta$	21-21	50-50
$\alpha\beta$	PN-$\alpha\beta$ with PN-TT	21-18	54-46
PN-$\alpha\beta$ without PN-TT	$\alpha\beta$	38-36	51-49
$\alpha\beta$	PN-$\alpha\beta$ without PN-TT	31-17	65-35

a challenge to find a *dynamic* strategy, which determines on a per-move basis when to apply PN search in a game. We believe that a good heuristic may improve the results of Table 4.5 considerably.

An important issue is that the framework of MIA is appropriate for the beneficial use of a PN-TT. According to Breuker (1998) PN search explores some positions more deeply than an $\alpha\beta$ search would do. The information gained in those deep searches is not useful in a narrow $\alpha\beta$ search, but MIA uses a search extension in the form of a quiescence search (see Chapter 2), which can lead to deeper paths than usual in depth-limited $\alpha\beta$ search. In this sense MIA is somewhat biased in favour of the method of using information from PN search.

From the above, we may conclude that our method of combining PN search and $\alpha\beta$ outperforms plain $\alpha\beta$ search in the tournament program MIA. We note that the kind of games played by MIA potentially results in PN-search friendly positions, i.e., positions with many forced moves. Therefore, it has to be tested whether this approach is also profitable in other LOA programs. It might happen that PN-$\alpha\beta$ is compensating for the possible weak play of MIA in the endgame. Another conclusion might be that the proved-nodes transposition table makes up for the loss of time when the position is not proved.

4.5 Chapter Conclusion and Future Research

Below we offer five observations. First, we have observed that mobility and deleting (dis)proved pn_2 subtrees speed up PN and PN^2 and increase their ability of solving endgame positions. Second, we have seen that the various PN-search algorithms outperform $\alpha\beta$ in solving endgame positions in LOA. Third, the memory problems make the plain PN search a weaker solver for the harder problems. Fourth, PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$. Fifth, in the tournament program MIA our method of combining PN search and $\alpha\beta$ outperforms plain $\alpha\beta$ search.

We may conclude that PN and its variants offer a valuable tool for enhancing programs in endgames. We remark that PN^2 is still restricted by working memory, and that PDS is three times slower than PN^2 (Table 4.4) because of the delayed evaluation. We will solve the memory and speed problem in the next chapter.

One problem clearly remains, viz. that there is no dynamic strategy available that determines when to use PN-$\alpha\beta$ search instead of $\alpha\beta$. This will be subject of future research.

Chapter 5

An Effective Two-Level Proof-Number Search Algorithm

This chapter is an updated and abridged version of the following two articles:

1. Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003c). PDS-PN: A New Proof-Number Search Algorithm: Application to Lines of Action. *Computers and Games, Lecture Notes in Computer Science 2883* (eds. M. Müller, J. Schaeffer, and Y. Björnsson), pp. 170–185, Springer-Verlag, Berlin, Germany.
2. Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2004b). An Effective Two-Level Proof-Number Search Algorithm. *Theoretical Computer Science*, Vol. 313, No. 3, pp. 511–525.[1]

In Chapter 4 we have seen that (1) the advantage of PN^2 over PDS is that it is faster and that (2) the advantage of PDS over PN^2 is that its tree is constructed as a depth-first tree, which is not restricted by the available working memory. This chapter answers the second research question by presenting a new proof-number search algorithm, called PDS-PN . It is a two-level search (like PN^2), which performs at the first level a depth-first Proof-number and Disproof-number Search (PDS), and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS.

The chapter is organised as follows. In Section 5.1 we explain the working of PDS-PN by elaborating on PDS and the idea of two-level search algorithms. Then, in Section 5.2, the results of experiments with PDS-PN on a set of endgame positions are given. Finally, in Section 5.3 we present our conclusion and propose future research.

[1]The author is grateful to Springer-Verlag and Elsevier for giving permission of reusing relevant parts of the articles in this thesis.

5.1 PDS-PN

In this section we give a description of PDS-PN search, which is a two-level search using PDS at the first level and PN at the second level. In Subsection 5.1.1 we motivate why we developed the method. In Subsection 5.1.2 we describe the first-level PDS, and in Subsection 5.1.3 we provide background information on the second-level technique. Finally, in Subsection 5.1.4 the relevant parts of the pseudo code are given.

5.1.1 Motivation

We were motivated to develop the PDS-PN algorithm by the clear advantage that PDS is traversing a depth-first tree instead of a best-first tree. Hence, PDS is not restricted by the available working memory. As against this, PN has the advantage of being fast compared to PDS (see Chapter 4).

The PDS-PN algorithm is designed to combine the two advantages. At the first level, the search is a depth-first search, which implies that PDS-PN is not restricted by memory. At the second level the focus is on fast PN. It is a complex balance, but we expect that PDS-PN will be faster than PDS, and PDS-PN will not be hampered by memory restrictions. Since the expectation on the effectiveness of PDS-PN is difficult to prove we have to rely on experiments (see Section 5.2). In the next two subsections we start describing PDS-PN.

5.1.2 First Level: Proof-Number and Disproof-Number Search

PDS-PN is a two-level search like PN^2. At the first level a PDS search is performed, denoted pn$_1$. For the expansion of a pn$_1$ leaf node, not stored in the transposition table, a PN search is started, denoted pn$_2$.

Proof-number and Disproof-number Search (PDS) (Nagai, 1998) is a straightforward extension of PN*. Instead of using only proof numbers such as in PN*, PDS uses disproof numbers too. PDS exploits a method called *multiple-iterative deepening.* Instead of iterating only in the root such as in ordinary iterative deepening, PDS iterates in *all* interior nodes. The advantage of using the multiple-iterative-deepening method is that in most cases it accomplishes to select the most-proving node (see below), not only in the root, but also in the interior nodes of the search tree. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a *TwoBig* transposition table (Breuker *et al.*, 1996).

PDS uses two thresholds for a node, one as a limit for proof numbers and one for disproof numbers. Once the thresholds are assigned to a node, the subtree rooted in that node is stopped to be searched if both the proof number and disproof number are larger than or equal to the thresholds *or* if the node is proved or disproved. The thresholds are set in the following way. At the start of every iteration, the proof-number threshold *pnt* and disproof-number threshold *dnt* of a node are equal to the node's proof number *pn* and disproof number *dn*. If it seems more likely that the node can be proved than disproved (called *proof-like*), the proof-number threshold is increased. If it seems more likely that the node can be disproved than proved (called *disproof-like*), the disproof-number threshold is increased. In passing we note that it is easier to prove a tree in an OR node, and to disprove a tree in an AND node. Below we repeat Nagai's (1998) heuristic to determine proof-like and disproof-like.

In an interior OR node n with parent p (direct ancestor) the solution of n is proof-like, if the following condition holds:

$$pnt_p > pn_p \;\; AND \;\; (pn_n \leq dn_n \;\; OR \;\; dnt_p \leq dn_p) \tag{5.1}$$

otherwise, the solution of n is disproof-like.

In an interior AND node n with parent p (direct ancestor) the solution of n is disproof-like, if the following condition holds:

$$dnt_p > dn_p \;\; AND \;\; (dn_n \leq pn_n \;\; OR \;\; pnt_p \leq pn_p) \tag{5.2}$$

otherwise, the solution of n is proof-like.

When PDS does not prove or disprove the root given the thresholds, it increases the proof-number threshold if its proof number is smaller than or equal to its disproof number, otherwise it increases the disproof-number threshold. Finally, we remark that only expanded nodes are evaluated. This is called *delayed evaluation* (cf. Allis, 1994). The expanded nodes are stored in a transposition table. The proof and disproof number of a node are set to unity when not found in the transposition table. Since PDS does not store unexpanded nodes which have a proof number 1 and disproof number 1, it can be said that PDS initialises the proof and disproof number by using the number of children. The mobility enhancement of PN and PN^2 (see Subsection 4.2.1) is already implicitly incorporated in the PDS search.

PDS is a depth-first search algorithm but behaves like a best-first search algorithm. In most cases PDS selects the same node for expansion as PN search. By using transposition tables PDS suffers from the graph-history-interaction problem (cf. Breuker *et al.*, 2001a). Especially the GHI evaluation problem can occur in LOA too. For instance, draws can be agreed upon due to the three-fold-repetition rule. Thus, dependent on its history a node can be a draw or can have a different value. However, in the current PDS algorithm we ignore this problem, since we believe that it is less relevant for the game of LOA than for Chess.

A Detailed Example

A detailed step-by-step example of the working of PDS is given in Figure 5.1. A square denotes an OR node, and a circle denotes an AND node. The numbers at the upper side of a node denote the proof-number threshold (left) and disproof-number threshold (right). The numbers at the lower side of a node denote the proof number (left) and disproof number (right).

In the first iteration (top of Figure 5.1), threshold values of the root A are set to unity. A is expanded, and nodes B and C are generated. The proof number of A becomes 1 and the disproof number becomes 2. Because both numbers are larger than or equal to the threshold values the search stops.

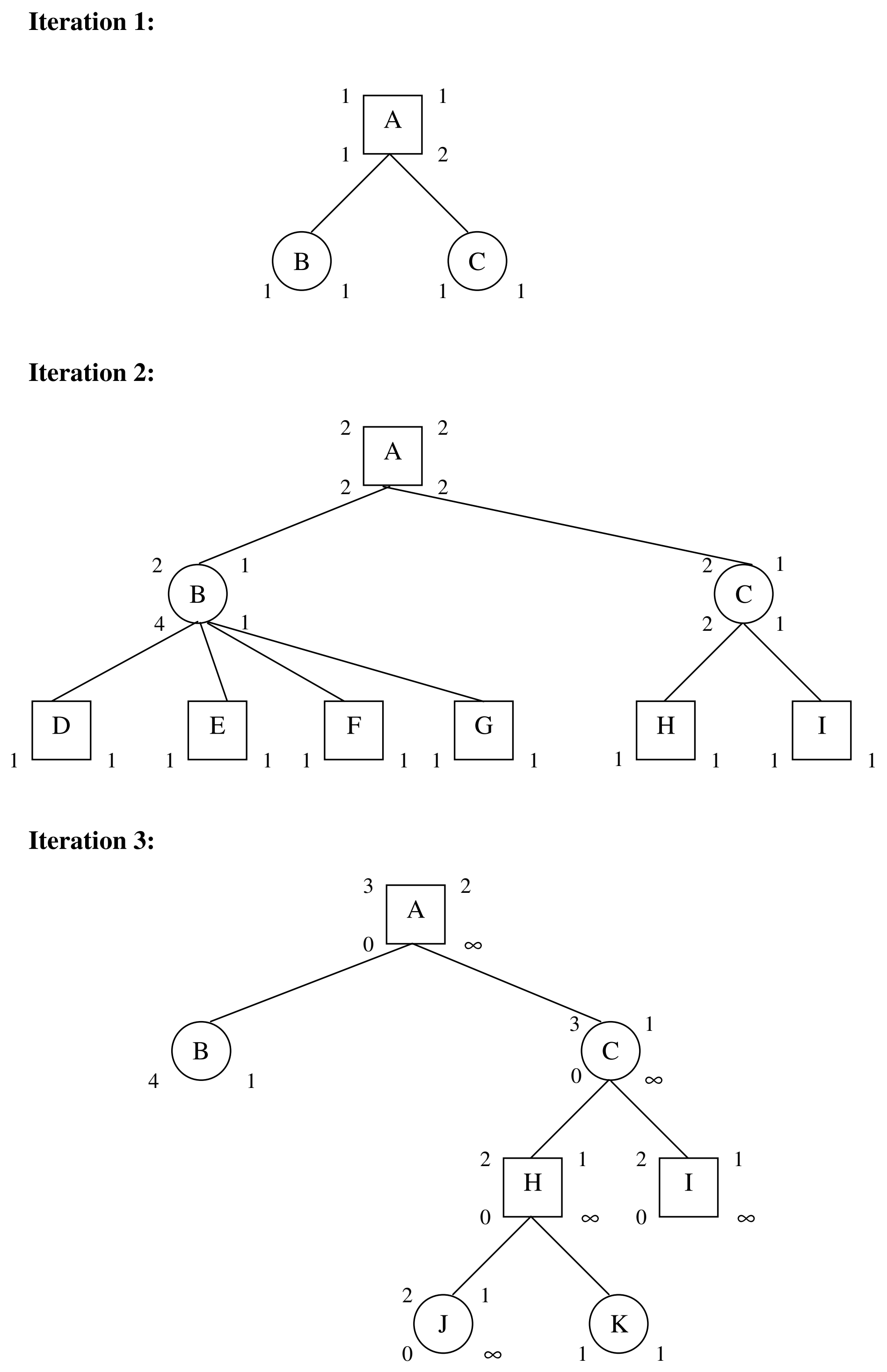

Figure 5.1: An illustration of PDS.

In the second iteration (middle of Figure 5.1), the proof-number threshold is incremented to 2, because the proof number of A (i.e., 1) is the smaller one of both A's proof number and disproof number (i.e., 2). We again expand A and re-generate B and C. The proof number of A is below its proof-number threshold and we continue searching. Now we have to select the child with minimum proof number. Because B and C have the same proof number, the left-most node B is selected. Initially, we set the proof-number and disproof-number threshold of B to its proof and disproof number (both 1). Because B is an AND node we have to look whether the solution of B is disproof-like by checking condition 5.2. The disproof-number threshold of A is not larger than its disproof number (both are 2), therefore the solution of B is not disproof-like but proof-like. Thus, the proof-number threshold of B has to be incremented to 2. Next, node B is expanded and the nodes D, E, F and G are generated. The search in node B is stopped because its proof number (i.e., 4) and disproof number (i.e., 1) are larger than or equal to the thresholds (i.e., 2 and 1, respectively). Node B is stored in the transposition table with proof number 4 and disproof number 1. Then the search backtracks to A. There we have to check whether we still can continue searching A. Since the proof number of A is smaller than its threshold, we continue and subsequently we select C, because this node has now the minimum proof number. The thresholds are set in the same way as in node B. Node C has two children H and I. The search at node C is stopped because its proof number (i.e., 2) and disproof number (i.e., 1) are not below the thresholds. C is stored in the transposition table with proof number 2 and disproof number 1. The search backtracks to A and is stopped because its proof number (i.e., 2) and disproof number (i.e., 2) are larger than or equal to the thresholds. We remark that at this moment B and C are stored because they were expanded.

In the third iteration (bottom of Figure 5.1) the proof-number threshold of A is incremented to 3. Nodes B and C are again generated, but this time we can find their proof and disproof numbers in the transposition table. The node with smallest proof number is selected (C with proof number 2). Initially, we set the proof-number threshold and disproof-number threshold of C to its proof and disproof number (i.e., 2 and 1, respectively). Because C is an AND node we have to look whether the solution is disproof-like by checking condition 5.2. The disproof-number threshold of A is not larger than its disproof number (both are 2), therefore the solution is not disproof-like but proof-like. Thus, the proof-number threshold of C has to be incremented to 3. C has now proof-number threshold 3 and disproof-number threshold 1. Nodes H and I are generated again by expanding C. This time the proof number of C (i.e., 2) is below the proof-number threshold (i.e., 3) and the search continues. The node with minimum disproof number is selected (i.e., H). Initially, we set the proof-number threshold and disproof-number threshold of H to its proof and disproof number (i.e., both 1). Because H is an OR node we have to look whether the solution is proof-like by checking condition 5.1. The proof-number threshold of C (i.e., 3) is larger than its proof number (i.e., 2), therefore the solution is proof-like. Hence, the search expands node H with proof-number threshold 2 and disproof-number threshold 1. Nodes J and K are generated. Because the proof number of H (i.e., 1) is below its threshold (i.e., 2), the node with minimum proof number is selected. Because J is an AND node we have to look whether the solution of J is disproof-like by checking condition 5.2. The disproof-number threshold of H (i.e., 1) is not larger than its disproof number (i.e., 2), therefore the solution of J is not disproof-like but proof-like. J is expanded with proof-number threshold 2 and disproof number threshold 1. Since node J is a terminal win position its proof number is set to 0 and its disproof number set to ∞. The search backtracks to H. At node H the proof number becomes 0 and the disproof number ∞, which means the node is proved. The search backtracks to node C. The search continues because the proof number of C (i.e., 1) is not larger than or equal to the proof-number threshold (i.e., 3). We select now node I because it has the minimum disproof number. The thresholds of node I are set to 2 and 1, as was done in H. The node I is a terminal win position; therefore its proof number is set to 0 and its disproof number to ∞. At this moment the proof number of C is 0 and the disproof number ∞, which means that the node is proved. The search backtracks to A. The proof number of A becomes 0, which means that the node is proved. The search stops at node A and the tree is proved.

5.1.3 Second Level: PN Search

For an adequate description we reiterate a few sentences from Subsection 4.2.2. At the leaves of the first-level search tree, the second-level search is called, similar as in PN^2 search. The PN search of the second-level, denoted pn$_2$ search, is bounded by the number of nodes that may be stored in memory. The number is a fraction of the size of the pn$_1$-search tree, for which we take the current number of nodes stored in the transposition table of the PDS search. Preferably, this fraction should start small, and grow larger as the size of the first-level search tree increases. A standard model for this growth is the logistic-growth model (Berkey, 1988). The fraction *f(x)* is therefore given by the logistic-growth function, x being the size of the first-level search:

$$f(x) = \frac{1}{1 + e^{\frac{a-x}{b}}} \tag{5.3}$$

with parameters a and b, both strictly positive. The parameter a determines the transition point of the function: as soon as the size of the first-level search tree reaches a, the second-level search equals half the size of the first-level search. Parameter b determines the S-shape of the function: the larger b, the more stretched the S-shape is. The number of nodes y in a pn$_2$-search tree is restricted by the minimum of this fraction function and the number of nodes which can still be stored. The formula to compute y is:

$$y = min(x \times f(x), N - x) \tag{5.4}$$

with N the maximum number of nodes to be stored in memory.

The pn$_2$ search is stopped when the number of nodes stored in memory exceeds y or the subtree is (dis)proved. After completion of the pn$_2$-search tree, only the root of the pn$_2$-search tree is stored in the transposition table of the PDS search.

We remark that for pn_2-search trees *immediate evaluation* (cf. Allis, 1994) is used. This two-level search is schematically sketched in Figure 5.2.

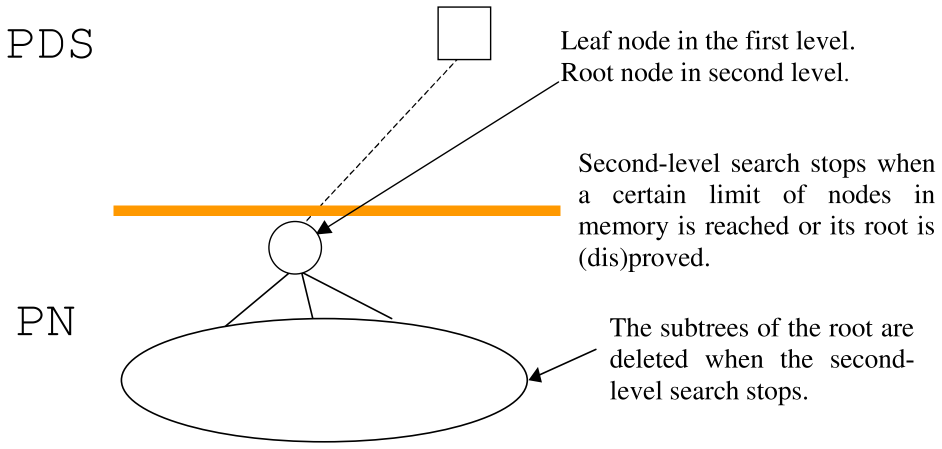
PDS
Leaf node in the first level.
Root node in second level.
Second-level search stops when a certain limit of nodes in memory is reached or its root is (dis)proved.
PN
The subtrees of the root are deleted when the second-level search stops.

Figure 5.2: Schematical sketch of PDS-PN.

In the second-level search proved or disproved subtrees are deleted. If we do not delete proved or disproved subtrees in the pn_2 search, the number of nodes searched becomes the same as y. When we include deletions the second-level search can continue on average considerably longer.

5.1.4 Pseudo Code for PDS-PN

In this subsection we provide the pseudo code for PDS-PN. For ease of comparison we use similar pseudo code as used by Nagai (1998) for the PDS algorithm. The proof number in an OR node and the disproof number in an AND node are equivalent. Analogously, the disproof number in an OR node and the proof number in an AND node are equivalent. As they are dual to each other, an algorithm similar to negamax in the context of minimax searching can be constructed. This algorithm is called NegaPDSPN. In the following, procedure `MID(n)` performs multiple iterative deepening. The function `proofSum(n)` computes the sum of the proof numbers of all the children. The function `disproofMin(n)` computes the minimum of the disproof numbers of all the children. The procedures `putInTT()` and `lookUpTT()` store and retrieve information to and from the transposition table. `isTerminal(n)` checks whether a node is a win, a loss or a draw. The procedure `generateChildren(n)` generates the children of the node. By default, the proof number and disproof number of a node are set to unity. The procedure `findChildrenInTT(n)` checks whether the children are already stored in the transposition table. If a hit occurs for a child, its proof number and disproof number are set to the values found in the transposition table. The procedure `PN()` is just the plain PN search. The algorithm is described in Allis (1994) and Breuker (1998). The function `computeMaxNodes()` computes the number of nodes which may be stored for the PN search, according to equation 5.4.

```
//Iterative deepening at root r
procedure NegaPDSPN(r){

  r.proof = 1;
  r.disproof = 1;

  while(true){
    MID(r);
    //Terminate when the root is proved or disproved
    if(r.proof == 0 || r.disproof == 0)
      break;

    if(r.proof <= r.disproof)
      r.proof++;
    else
      r.disproof++;
  }
}

//Explore node n
procedure MID(n){

  //Look up in the transposition table
  lookUpTT(n, &proof, &disproof);
  if(proof == 0 || disproof == 0
  || (proof >= n.proof && disproof >= n.disproof)){
    n.proof = proof; n.disproof = disproof;
    return;
  }

  //Terminal node
  if(isTerminal(n)){
    if((n.value == true && n.type == AND_NODE)
    ||(n.value == false && n.type == OR_NODE)){
      n.proof = INFINITY; n.disproof = 0;
    }
    else{
      n.proof = 0; n.disproof = INFINITY;
    }
    putInTT(n);
    return;
  }

  generateChildren(n);
  //Avoid cycles
  putInTT(n);

  //Multiple-iterative deepening
  while(true){
    //Check whether the children are already stored in the TT.
    //If a hit occurs for a child, give its proof number and
    //disproof number the values found in the TT.
    findChildrenInTT(n);

    //Terminate searching when both proof and disproof number
    //exceed their thresholds
    if(proofSum(n) == 0 || disproofMin(n) == 0 || (n.proof <=
    disproofMin(n) && n.disproof <= proofSum(n))){
      n.proof = disproofMin(n);
      n.disproof = proofSum(n);
      putInTT(n);
      return;
    }

    proof = max(proof, disproofMin(n));
    n_child = selectChild(n, proof);

    if(n.disproof > proofSum(n) && (proof_child <= disproof_child
      || n.proof <= disproofMin(n)))
      n_child.proof++;
    else
      n_child.disproof++;

    //This is the PDS-PN part
    ////////////////////////////////
    if(!lookUpTT(n_child)){
      //Call PN search with a second argument the maximum number
      //of nodes in memory
      PN(n_child, computeMaxNodes());
      putInTT(n_child);
    }
    else
    ////////////////////////////////
      MID(n_child);
  }
}

//Select among children
selectChild(n, proof){

  min_proof = INFINITY;
  min_disproof  = INFINITY;
  for(each child n_child){
    disproof_child = n_child.disproof;
    if(disproof_child != 0)
      disproof_child = max(disproof_child, proof);

    //Select the child with the lowest disproof_child (if there are
    //plural children among them select the child with the lowest
    //n_child.proof)
    if(disproof_child < min_disproof || (disproof_child = min_disproof
    && n_child.proof < min_proof)){
      n_best = n_child;
      min_proof = n_child.proof;
      min_disproof = disproof_child;
    }
  }
  return n_best;
}
```

Figure 5.3: Pseudo code for PDS-PN.

5.2 Experiments

In this section we compare $\alpha\beta$, PN^2, PDS, and PDS-PN search with each other. The goal is to prove the effectiveness of PDS-PN by experiments. We will investigate how many endgame positions it can solve and the effort (in nodes and CPU time) it takes compared with $\alpha\beta$, PN^2, PDS. For PDS and PDS-PN we use a *TwoBig* transposition table. In Subsection 5.2.1 we test PDS-PN with different parameters a and b for the growth function. In Subsection 5.2.2 we compare PDS-PN with $\alpha\beta$, PN^2 and PDS on a set of 488 LOA positions in three different ways. In Subsection 5.2.3 we compare PDS-PN with PN^2 on a set of hard LOA problems. Finally, we evaluate the algorithms PDS-PN and PN^2 in solving problems under restricted memory conditions in Subsection 5.2.4.

5.2.1 Parameter Tuning

In the following series of experiments we measured the solving ability with different parameters a and b. Parameter a takes values of 150K, 450K, 750K, 1050K, and 1350K, and for each value of a parameter b takes values of 60K, 120K, 180K, 240K, 300K, and 360K. The results are given in Table 5.1. For each a holds that the number of solved positions grows with increasing b, when the parameter b is still small. If b is sufficiently large, increasing it will not enlarge the number of solved positions. In the process of parameter tuning we found that PDS-PN solves the most positions with (450K, 300K) (see the bold line in Table 5.1). However, the difference with parameters configurations (150K, 180K), (150K, 240K), (150K, 300K), (150K, 360K), (450K, 360K), and (1350K, 300K) is not significant. On the basis of these results we decided that it is not necessary to perform experiments with a larger a.

Table 5.1: Number of solved positions (by PDS-PN) for different values of a and b.

a	b	# of solved pos.
150,000	60,000	460
150,000	120,000	458
150,000	180,000	466
150,000	240,000	466
150,000	300,000	465
150,000	360,000	466
450,000	60,000	445
450,000	120,000	463
450,000	180,000	460
450,000	240,000	461
450,000	**300,000**	**467**
450,000	360,000	464
750,000	60,000	432
750,000	120,000	449
750,000	180,000	461
750,000	240,000	463
750,000	300,000	460
750,000	360,000	461
1,050,000	60,000	421
1,050,000	120,000	448
1,050,000	180,000	451
1,050,000	240,000	459
1,050,000	300,000	459
1,050,000	360,000	460
1,350,000	60,000	421
1,350,000	120,000	433
1,350,000	180,000	447
1,350,000	240,000	454
1,350,000	300,000	465
1,350,000	360,000	459

5.2.2 Three Comparisons of the Algorithms

In the experiments with PN^2, PDS, and PDS-PN all nodes evaluated during the search are counted; for the $\alpha\beta$ depth-first iterative-deepening searches nodes at depth i are counted only during iteration i. We adopted this method from Allis (1994). It makes a general comparison possible. The maximum number of nodes searched is 50,000,000. The limit corresponds roughly to tournament conditions. The maximum number of nodes stored in memory is 1,000,000. The parameters (a,b) of the growth function used in PN^2 are set at (1800K, 240K) according to the suggestions in Breuker *et al.* (2001b). The parameter configuration (450K, 300K) found in the previous subsection will be used for PDS-PN. The smaller value of a corresponds with the smaller pn_1 trees resulting from the use of PDS-PN instead of PN^2. The fact that PDS is much slower than PN is an important factor too.

First Comparison

$\alpha\beta$, PN^2, PDS, and PDS-PN are tested on the same set of 488 forced-win LOA positions as described in Subsection 4.3.2. The results are given in Table 5.2. In the first column the four algorithms are mentioned. In the second column we see that 382 positions are solved.[2] by $\alpha\beta$, 470 positions by PN^2, 473 positions by PDS, and 467 positions by PDS-PN. The set of 488 positions contains no position that only could be solved by $\alpha\beta$ search. In the third and fourth column the number of nodes and the time consumed are given for the subset of 371 positions, which all four algorithms are able to solve. A look at the third column shows that PDS search builds the smallest search trees and $\alpha\beta$ by far the largest. Like PN^2 and PDS, PDS-PN solves significantly more positions than $\alpha\beta$. This suggests that PDS-PN is a better endgame solver than $\alpha\beta$. As we have seen before, PN^2 and PDS-PN investigate more nodes than PDS, but both are still faster in CPU time than PDS for this subset. Due to the limit of 50,000,000 nodes and the somewhat lower search efficiency, PDS-PN solves three positions fewer than PN^2 and six fewer than PDS.

Table 5.2: Comparing the search algorithms on 488 test positions with a limit of 50,000,000 nodes.

Algorithm	# of positions solved (out of 488)	371 positions	
		Total # of nodes	Total time (ms.)
$\alpha\beta$	382	2,645,022,391	33,878,642
PN^2	470	505,109,692	3,642,511
PDS	473	239,896,147	16,960,325
PDS-PN	467	924,924,336	5,860,908

Second Comparison

To investigate whether the memory restrictions are an actual obstacle we increase the limit of nodes searched to 500,000,000 nodes. In this second comparison PN^2 solves now 479 positions and PDS-PN becomes the best solver with a performance of 483 positions. The detailed results are given in Table 5.3.

Table 5.3: Comparing PN^2 and PDS-PN on 488 test positions with a limit of 500,000,000 nodes.

Algorithm	# of positions solved (out of 488)	479 positions	
		Total # of nodes	Total time (ms.)
PN^2	479	2,261,482,395	13,295,688
PDS-PN	483	4,362,282,235	23,398,899

The performance of PDS-PN in Table 5.3 is more effective than that of PN^2, viz. 483 to 479. However, we should thoughtfully take into account the condition for the total number of nodes searched and the time spent. Therefore, we continue

[2]We remark that a slightly less inefficient version of our $\alpha\beta$ implementation could solve 383 positions (see Chapter 4).

our research in the direction of nodes searched and time spent with the 50,000,000 nodes limit. A reason for this decision is that the experimental time constraints are necessary for the PDS experiments.

Third Comparison

For a better insight into the relation between PN^2, PDS, and PDS-PN we have done a third comparison. In Table 5.4 we provide the results of PN^2, PDS, and PDS-PN on a new subset of 457 positions of the principal test set, viz. all positions the three algorithms could solve under the 50,000,000 nodes limit condition. Now, PN^2 searches 2.6 times more nodes than PDS. The reason for the difference of performance is that for hard problems the pn$_2$-search tree becomes as large as the pn$_1$-search tree. Therefore, the pn$_2$-search tree is causing more overhead. However, if we look at the CPU time we see that PN^2 is almost four times faster than PDS. PDS has a relatively large time overhead because it performs multiple-iterative deepening at all nodes. PDS-PN searches 3.7 times more nodes than PDS but is still three times faster than PDS in CPU time. This is because PDS-PN is focussing more on the fast PN at the second level than on PDS at the first level. PDS-PN searches more nodes than PDS since the pn$_2$-search tree is repeatedly rebuilt and removed. The overhead is even bigger than PN^2's overhead because the children of the root of the pn$_2$-search tree are not stored (i.e., this is done to focus more on the fast PN search). It explains why PDS-PN searches 1.4 times more nodes than PN^2. Hence, our provisional conclusions are that on this set of 457 positions and under the 50,000,000 nodes condition: (1) PN^2 outperforms PDS-PN, and (2) PDS-PN is a faster solver than PDS and therefore more effective than PDS.

Table 5.4: Comparing PN^2, PDS and PDS-PN on 457 test positions (all solved) with a limit of 50,000,000 nodes.

Algorithm	Total # of nodes	Total time (ms.)
PN^2	1,275,155,583	9,357,663
PDS	498,540,408	36,802,350
PDS-PN	1,845,371,831	11,952,086

5.2.3 Comparing the Algorithms for Hard Problems

Since the impact of the 50,000,000 nodes condition somewhat obscured our provisional conclusions above and since we felt that PDS-PN had its own merits in comparison with PN^2 we performed a new experiment with LOA problems. In this experiment PN^2 and PDS-PN are tested on a different set of 286 LOA positions, which were on average harder than the ones in the previous test set.[3] The conditions are the same as in the previous experiments except that the maximum number of

[3]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/tscg2002b.zip.

nodes searched is set at 500,000,000. The PDS algorithm is not included because it takes too much time given the current node limit. In Table 5.5 we see that PN^2 solves 265 positions and PDS-PN 276. We remark that PN^2 solves 10 positions, that PDS-PN does not solve, but that PDS-PN solves 21 positions that PN^2 does not solve. The ratio in nodes and time between PN^2 and PDS-PN for the positions solved by both (255) is roughly similar to the previous experiments. The reason why PN^2 solves fewer positions than PDS-PN is its being restricted in working memory. We are in a delicate position since new experiments with much more working memory are now on the list to be performed. However, we assume that the nature of PN^2 with respect to using so much memory cannot be overcome. Hence we may conclude that within an acceptable time frame PDS-PN is a more effective endgame solver than PN^2 for hard problems.

Table 5.5: Comparing PN^2 and PDS-PN on 286 hard test positions with a limit of 500,000,000 nodes.

Algorithm	# of positions solved (out of 286)	255 positions	
		Total # of nodes	Total time (ms.)
PN^2	265	10,061,461,685	57,343,198
PDS-PN	276	16,685,733,992	84,303,478

5.2.4 Comparing the Algorithms under Reduced Memory

From the experiments in the previous subsection it is clear that PN^2 will not be able to solve very hard problems since it will run out of working memory. To support this statement experimentally even further, we tested the solving ability of PN^2 and PDS with restricted working memory. In these experiments we started with a memory capacity sufficient to store 1,000,000 nodes, subsequently we divided the memory capacity by two at each next step. The parameters a and b were also divided by two. The relation between memory and number of solved positions for both algorithms is given in Figure 5.4. We see that the solving performance rapidly decreases for PN^2. The performance of PDS-PN remains stable for a long time. Only when PDS-PN is restricted to fewer than 10,000 nodes, it begins to solve fewer positions. This experiment suggests that PDS-PN is to be preferred above PN^2 for the very hard problems, because it is not suffering from memory constraints.

5.3 Chapter Conclusion and Future Research

Below we offer three observations, a conclusion, and a suggestion for future research.

Our first observation is that PDS-PN is able to solve significantly more LOA endgame problems than $\alpha\beta$ search with enhancements. Our second observation is that the PDS-PN algorithm is almost as fast as PN^2 when the parameters for its growth function are chosen properly. It turns out that for each a it holds that the

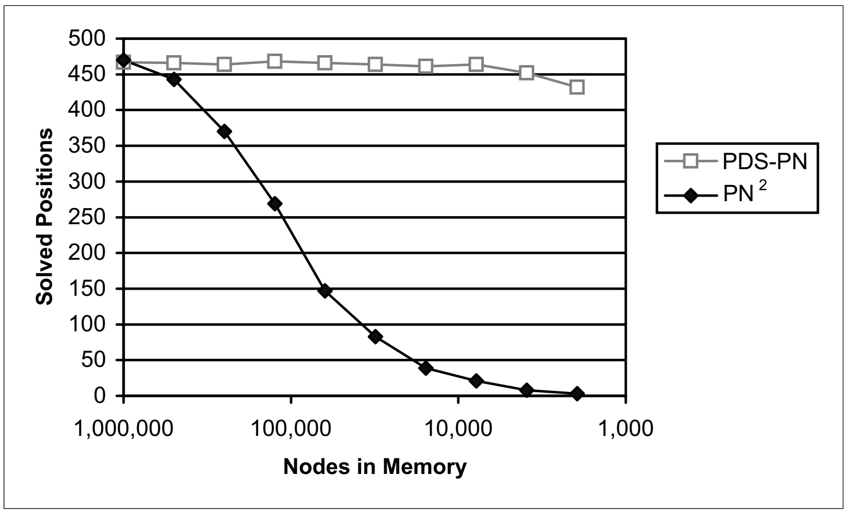

Figure 5.4: Results with restricted memory.

number of solved positions grows with increasing *b*, when the parameter *b* is still small. If *b* is sufficiently large, increasing it will not enlarge the number of solved positions. Our third observation states that (1) PDS-PN solves more hard positions than PN^2 within an acceptable time frame and (2) PDS-PN is more effective than PN or even PN^2 because it does not run out of memory for hard problems. Moreover, PDS-PN performs quite well under harsh memory conditions. This is especially appropriate for hard problems and for environments with very limited memory such as hand-held computer platforms.

Hence, we may conclude that PDS-PN is a more effective endgame solver for a set of hard problems than PDS and PN^2.

Finally, we believe that an adequate challenge is testing PDS-PN in other domains with difficult endgames. An example of a game notoriously known for its difficult endgames is the game of Tsume-Shogi (a variant of Shogi). Several hard problems including solutions over a few hundred ply are solved by PN* (Seo *et al.*, 2001) and PDS (Sakuta and Iida, 2001; Nagai, 2002). It would be interesting to test PDS-PN on these problems.

Chapter 6

Enhanced Forward Pruning

This chapter is an updated and abridged version of the following two publications:

1. Winands, M.H.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Werf, E.C.D. van der (2003b). Enhanced Forward Pruning. *Proceedings of the 7th Joint Conference on Information Sciences (JCIS 2003)* (eds. P. Wang *et al.*), pp. 485–488.
2. Winands, M.H.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Werf, E.C.D. van der (2004a). Enhanced Forward Pruning. *Information Sciences*. Accepted for publication.[1]

This chapter answers the third research question by improving forward-pruning methods in the Principal-Variation-Search (PVS) framework (Marsland, 1983). PVS is essentially equivalent to the popular NegaScout (Reinefeld, 1983) in the sense that it expands the same search tree (Björnsson, 2002). Forward-pruning methods, such as multi-cut and null move, are improved at so-called ALL nodes. We modified the Principal Variation Search by four small but essential additions. The new PVS algorithm guarantees that forward pruning is safe at ALL nodes.

This chapter is organised as follows. In Section 6.1 we give a short overview about variable-depth search. Section 6.2 explains the different node types in the $\alpha\beta$ search. Next, the way forward pruning works in PVS is explained in Section 6.3. Subsequently, multi-cut is described in Section 6.4. The method is tested at expected ALL nodes and compared to an aggressive version of the null move in Section 6.5. Finally, Section 6.6 gives the chapter conclusion and future research.

6.1 Variable-Depth Search

For a long time, brute-force $\alpha\beta$ search was the standard procedure in games such as Chess and Checkers. Over the years many improvements have been proposed.

[1]The author is grateful to Elsevier for the permission of reusing relevant parts of the article in this thesis.

An obvious improvement is the introduction of a variable-depth search, i.e., exploring promising moves more deeply (extending the search) and non-promising moves less deeply (pruning the search). The use of selective extensions provided better results right from the beginning. An example of a knowledge-independent technique which explores some continuations more deeply is singular extensions (Anantharaman, Campbell, and Hsu, 1988). Enhancing the search with forward pruning led to mixed results until the 1990s, when the null move (Beal, 1989; Goetsch and Campbell, 1990; Donninger, 1993) was introduced in Chess. It proved to be successful in Chess and quite decent in some other games. However, its implicit assumption that passing is often bad made it less applicable for games where zugzwang occurred a lot (e.g., Checkers). As an alternative Buro introduced ProbCut (1995) and its enhanced version Multi-ProbCut (2000). It has been proved to be successful in Othello, but has not yet been shown convincingly useful in other games (e.g., Chess or Checkers). Nevertheless, these forward-pruning techniques have in common that, again, no explicit domain knowledge is required to make the search decision.

Recently, a new forward-pruning mechanism, called multi-cut (Björnsson and Marsland, 1999), has been proposed. According to its inventors it is more domain independent than the previous two. It has been adopted by some of the world's strongest commercial Chess programs (Björnsson, 2002). In particular, multi-cut is successfully applied to the game of LOA (Björnsson, 2002). Traditionally, multi-cut forward pruning was not applied at *expected* ALL nodes, since it was assumed that the technique was only successful at CUT nodes. However, in this chapter we will investigate whether it is beneficial to use forward-pruning methods at expected ALL nodes in the PVS framework.

6.2 Three Node Types

Knuth and Moore (1975) identified three types of nodes in the $\alpha\beta$ minimax tree: type-one, type-two, and type-three nodes. In this paper we use the terminology introduced by Marsland and Popowich (1985). They identify the nodes as PV, CUT, and ALL nodes, respectively. The root of the tree is a PV node. At a PV node all the children have to be investigated. The best move found at a PV node leads to a successor PV node, while all the other investigated children are CUT nodes. At a CUT node the child causing a β cut-off is an ALL node. In a perfectly ordered tree only one child of a CUT node has to be explored. At an ALL node all the children have to be explored. The successors of an ALL node are CUT nodes.

Before searching a node we do not really know what the type of the node will be. Thus, before exploring nodes we refer to them as *expected* PV nodes, *expected* CUT nodes, and *expected* ALL nodes. If none of the moves causes a cut-off at an expected CUT node, the node becomes an ALL node. If one of the children at an expected ALL node turns out not to be a CUT node, the expected ALL node becomes a CUT node. When all expected CUT nodes on a path from the root to a leaf node have become ALL nodes, a new principal variation has emerged (all the nodes on the path have become in fact PV nodes). In Figure 6.1 the different types of nodes are depicted in a tree. True PV, CUT, and ALL nodes are denoted by *P*, *C*, and *A*.

Expected CUT and expected ALL nodes, which turn to have a different node type, are denoted by $\underline{C}$ and $\underline{A}$. Shaded nodes are not belonging to the minimal tree.

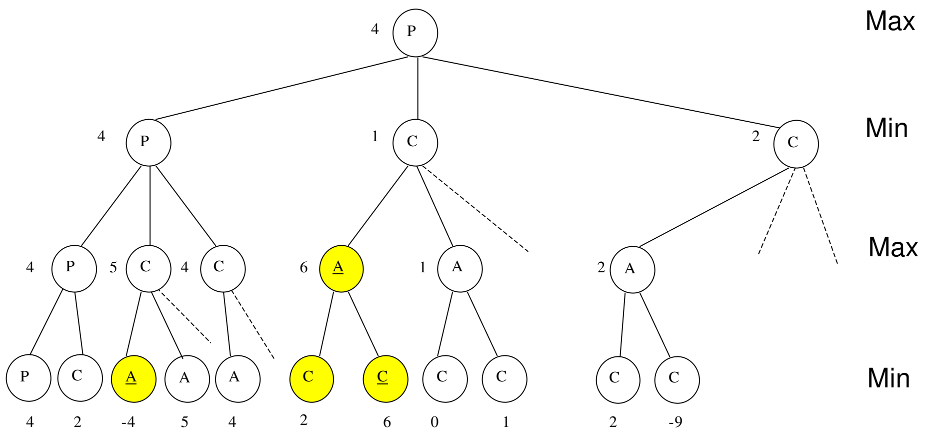

Figure 6.1: An example of an $\alpha\beta$ tree with different types of nodes.

The Principal-Variation-Search (PVS) framework (Marsland, 1983) is able to determine the expected and true type of a node. This algorithm is in general more efficient than the original $\alpha\beta$. The algorithm considers the first node explored at the root (and at subsequent PV nodes) to be a PV node. The value of that node is therefore treated as best value, and all the siblings are searched using a closed $\alpha\beta$-window (i.e., $\beta = \alpha + 1$) to prove that they are inferior. This part of the search is called zero-width-window search (Marsland and Björnsson, 2001) or null-window search (NWS) (Björnsson *et al.*, 1997). In the NWS, nodes are assumed not to be on the principal variation and therefore are expected to be alternately CUT and ALL nodes. If the NWS returns a score less than or equal to α, then that particular sibling has been proved inferior. Sometimes, the NWS returns a better score and a re-search has to be done. In that case the $\alpha\beta$-window is opened and the child node under consideration is regarded as a PV node.

6.3 Forward Pruning in the Null-Window Search

Recently, it has become practice to use forward-pruning methods only in the NWS part of the PVS framework (Björnsson *et al.*, 1997; Feldmann, 1997; Marsland and Björnsson, 2001). The idea is that it is too risky to prune forward at an expected PV node, because a possible mistake causes an immediate change of the principal variation. If we erroneously prune forward at some CUT node and the resulting score is backed up all the way to a PV node, we obtain a fail low (the subtree seemingly has been proved inferior). The result of the mistake is that a possible new principal variation is overlooked in this position. Therefore, forward pruning at a

CUT node without further provisions is dangerous. However, by the large savings achieved it has proved worthwhile to implement further provisions making forward pruning at CUT nodes more safe (and thus acceptable). If we erroneously forward prune at some ALL node and the resulting score is backed up all the way to a PV node, we obtain a fail high and a re-search will be performed. The algorithm is then searching for a new principal variation and regards the subsequent nodes as PV nodes. Because no forward pruning is done at PV nodes, it is possible to find out whether there exists a new PV or not. In this case the result of the mistake will be that an extra amount of nodes has been searched. In principle, forward pruning at an expected ALL node is not dangerous but can trigger unnecessary re-searches.

To ensure that a re-search is performed which is able to correct the value, some small changes in the PVS algorithm have to be made. The following four additions are performed.

1. To prevent that a backed-up value of a forward-pruned ALL node causes a β cut-off at the PV node lying above, the forward-pruning mechanism should return a value equal to β in case of a cut-off (which is equal to $\alpha + 1$ at an ALL node).

2. If the window of the PV node was already closed and the NWS should return a value equal to β $(\alpha + 1)$, we still have to do a re-search.

3. If we do a re-search and the returned value of the NWS equals $\alpha + 1$, we should do a re-search with α as lower bound.

4. CUT nodes where a fail-low has occurred with a value equal to α are not stored in the transposition table because their values are uncertain.

Pseudo code for this enhanced PVS algorithm is given in Figure 6.2.

6.4 Multi-Cut at ALL Nodes (MC-A)

Multi-cut pruning is a new forward-pruning method (Björnsson and Marsland, 1999). Before examining an expected CUT node to full depth, the first M child nodes are searched to a depth reduced with a factor R. If at least C child nodes return a value larger than or equal to β, a cut-off occurs. However, if the pruning condition is not satisfied, the search continues as usual, re-exploring the node under consideration to a full depth d. The multi-cut code is given in Figure 6.3.

In general the behaviour of multi-cut is as follows. The higher M and R are and the lower C is, the higher the number of prunings is.

In our opinion, we made two small improvements to the original algorithm by Björnsson and Marsland, 1999 (indicated in Figure 6.3). First, when at a reduced depth a winning value is found, the search is stopped and the winning value is returned. Second, if the multi-cut does not succeed in causing a cut-off, the moves causing a β cut-off at the reduced depth are tried first in the normal search. The remaining question is whether multi-cut is also useful at ALL nodes. The inventors of the multi-cut algorithm anticipated that it would not be successful elsewhere

```
CUT_NODE = 1, ALL_NODE = -1, PV_NODE = 0;

PVS(node, alpha, beta, depth, node_type){
    //Transposition table lookup, omitted
    ....................................
    if(depth == 0)
      return evaluate(pos);
    if(node_type != PV_NODE){
      //Forward-pruning code, omitted
      ....................................
      if(forward_pruning condition holds) return beta; //Addition 1
    }
    next = firstSuccessor(node);
    best = -PVS(next, -beta, -alpha, depth-1, -node_type);
    if(best >= beta) goto Done;
    next = nextSibling(next);
    while(next != null){
      alpha = max(alpha, best);
      //Null-Window Search Part
      value = -PVS(next, -alpha-1, -alpha, depth-1,
                  (node_type == CUT_NODE)?ALL_NODE:CUT_NODE);
      //Re-search
      if((value > alpha && value < beta) ||
        //Addition 2
        (node_type == PV_NODE && value == beta && beta == alpha+1)){
        //Value is not a real lower bound
        if(value == alpha+1) //Addition 3
           value = alpha;
        value = -PVS(next, -beta, -value, depth-1,  node_type);
      }
      if(value > best){
        best = value;
        if(best >= beta) goto Done;
      }
      next = nextSibling(next);
    }
    if(node_type == CUT_NODE && best == alpha) return best; //Addition 4

    Done: //Store in Transposition table, omitted
    ....................................
}
```

Figure 6.2: Pseudo code for enhanced PVS.

```
...........................................................
//Forward-pruning code
 if(node_type == CUT_NODE && depth > 2){
   next = firstSuccessor(node);
   c = 0, m = 0;
   while(next != null && m < M){
     value = -PVS(next, -beta, -alpha, depth-1-R, -node_type);
     if(value >= beta){
       c++;
       //Keep track of the moves causing a cut-off at d-R
       storeCutOffNode(next);
       if(value >= WIN_SCORE) //Addition 1
         return value;
       else if(c >= C)
         return beta;
     }
     m++;
     next = nextSibling(next);
   }
   //Re-order moves
   putCutOffNodesInFront(); //Addition 2
}
...........................................................
```

Figure 6.3: Pseudo code for enhanced MC-C.

(Björnsson and Marsland, 2001b) and therefore did not test it at *expected* ALL nodes. Henceforth, we call multi-cut at expected CUT nodes MC-C and at expected ALL nodes MC-A. In the following section, some experiments are reported on testing whether MC-A is beneficial.

6.5 Experiments

In this section four series of experiments are described with multi-cut at expected ALL nodes. In the first series, MC-A is tested under different parameter settings (Subsection 6.5.1). In the second series of experiments, we investigate the added value of MC-A with different combinations of forward-pruning methods (Subsection 6.5.2). In the third series, the variable null-move bound is tested and compared to MC-A (Subsection 6.5.3). Finally, in the fourth series of experiments the increase in playing strength of MIA is tested (Subsection 6.5.4).

6.5.1 Parameter Tuning in MC-A

In the first series of experiments we tried to find the parameter setting (C, M, R) of MC-A, which gave the largest node reduction. Using a set of 171 LOA positions[2], the program was tested at depth 10 using its normal enhancements described in the previous subsection. The following pairs of C and R were investigated: (1,2), (2,2), (3,2), (4,2), (1,3), (2,3), (3,3), (4,3), and (4,4). For each pair the parameter M took the values 2, 4, 6, 8, 10, 12, 14, and 16 except when $M < C$. In the case of $C = 3$ we also investigated $M = 3$. In Figure 6.4 the total number of nodes searched for each set of parameters is given. The results of the search without MC-A (default) are showed for comparison. In the process of parameter tuning we found that MC-A (2,10,2) is the most efficient. However, the difference with several other parameter configurations is not significant (e.g., (2,6,3)). It is clear that configurations with $C = 1$ have a moderate success. Probably, many re-searches had to be executed because of its aggressive nature (i.e., too much pruning). Configurations with $C = 4$ do not give good results as well, especially not when R and M are chosen too low. The parameter $C = 4$ is too conservative (i.e., not much pruning) and can be made more aggressive by increasing R and M. It appears that $C = 2$ and $C = 3$ are close to each other if M is sufficiently high.

In Figure 6.4 we see that MC-A with an optimal parameter setting gives a reduc-

[2]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/TMP.zip.

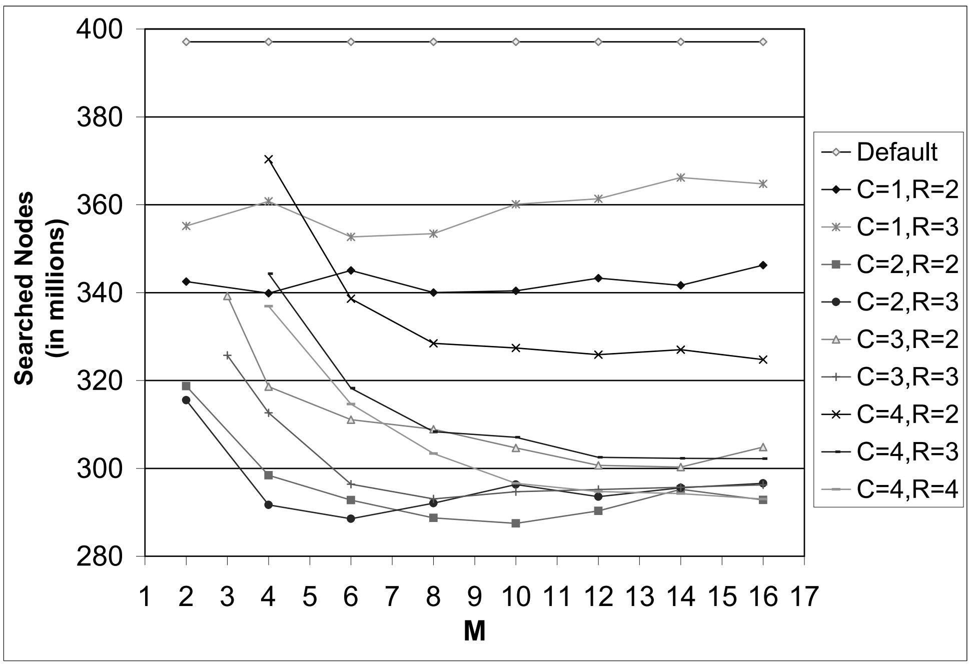

Figure 6.4: Tree sizes for different C, M, and R.

tion in nodes of 28 per cent compared to the default search engine. If we compare the (aggressive) optimal setting of parameters (i.e., (2,10,2)) to the ones in MC-C (i.e., (3,10,2)) we see a difference of 6 per cent. We would like to remark that MC-C parameters of MIA were already chosen more aggressively than the ones in the LOA-playing program YL (i.e., (3,3,2)) (Björnsson, 2002). The last data point gives less reduction according to our experiments (see Figure 6.4).

A possible enhancement is making the values for C, M, and R variable instead of constant for MC-A. An attractive scheme is to set R to 2 or 3 dependent on the search depth and static board properties (this idea is used in adaptive null-move pruning). Unfortunately, preliminary experiments showed no extra reduction of nodes, so we did not pursue this idea any further. Another idea is to forward prune immediately when the reduced search returns a value greater than $\beta + \delta$ with a sufficiently large δ. Using the optimal parameter setting, the optimal δ value (525) gives a small reduction of 1.5 per cent of nodes searched at depth 10 for the given test set. This heuristic is included in the following experiments. Finally, we remark that if we would not include the re-ordering of moves after an unsuccessful MC-A the number of nodes searched increases with 2 per cent at depth 10.

6.5.2 Evaluation of MC-A and Forward-Pruning Methods

In the second series of experiments we tested the added value of MC-A, using the optimal parameter setting of the previous subsection, against the same set of 171 positions at several depths in our original search engine. We also looked at the performance of MC-A in combination with the already included pruning methods (i.e., MC-C and null move). The results are given in Table 6.1.

If MC-A is added in a search engine where no forward pruning is used, we see that there is hardly an improvement in performance. The decrease in the number of nodes searched, is only some 2 per cent. Apparently, when no forward pruning is used at all, multi-cut is useless at expected ALL nodes. This is in accordance with the claim of Björnsson and Marsland (2001b). However, if some forward-pruning method is used we see that MC-A gives significant improvements. In the case of using only null move adding MC-A gives a further reduction increasing to 15 per cent at depth 12. If only MC-C is used as forward-pruning method, introducing MC-A to the search brings a further reduction increasing to 85 per cent at depth 12. This saving seems enormous but we have to consider that in the previous experiment null-move pruning was already performed at ALL nodes. In the case of using only MC-C there is no forward pruning at expected ALL nodes, which therefore results in a large saving when MC-A is added. We remark that the tree sizes of the combination MC-C and MC-A are approximately the same as the combination null move and MC-C. When null move and MC-C are available in the engine (the original situation), MC-A reduces the search tree with another 42 per cent at depth 14.

Since we have tuned MC-A at this particular test set (Subsection 6.5.1), we performed similar experiments on a set consisting of 156 different positions[3] to validate the result. In the first column of Table 6.2 we see the relative performance of MC-A on the original test set. In the second column the relative performance of MC-A is

[3]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/VMP.zip.

Table 6.1: Added value of MC-A.

	No Forward pruning			Only Null move		
d	No MC-A	MC-A	%	No MC-A	MC-A	%
5	5,071,689	4,995,845	98.5	3,504,759	3,404,882	97.2
6	19,896,101	19,286,868	96.9	10,109,533	9,518,082	94.1
7	113,653,808	110,663,056	97.4	36,265,257	34,671,647	95.6
8	416,549,038	406,489,302	97.6	92,749,650	89,483,140	96.5
9	2,427,406,280	2,395,844,102	98.7	314,507,126	303,466,596	96.5
10	9,635,185,102	9,460,591,510	98.2	891,348,022	813,032,326	91.2
11	-	-	-	2,930,142,106	2,599,157,486	88.7
12	-	-	-	8,362,297,395	7,080,475,905	84.7
	Only MC-C			Null move and MC-C		
d	No MC-A	MC-A	%	No MC-A	MC-A	%
5	2,097,908	1,955,564	93.2	2,012,835	1,897,600	94.3
6	7,314,731	5,549,772	75.9	6,083,136	5,122,496	84.2
7	28,656,432	20,221,202	70.6	20,491,711	17,423,109	85.0
8	85,103,638	50,333,688	59.1	50,018,470	42,242,144	84.5
9	297,239,554	149,671,128	50.4	142,182,834	116,784,068	82.1
10	1,286,515,396	393,490,307	30.6	397,092,800	283,350,391	71.4
11	4,860,474,957	1,358,658,246	28.0	1,223,918,717	846,066,886	69.1
12	23,806,355,059	3,536,842,482	14.9	3,328,838,963	2,162,692,924	65.0
13	-	-	-	9,869,101,893	6,289,563,990	63.7
14	-	-	-	30,087,791,323	17,578,589,423	58.4

given for the validation set. If we compare those two columns with each other, we see that there is not much difference between them and similar results are achieved.

Table 6.2: Relative performance of MC-A in combination with null move and MC-C.

Depth	Original Set (171 positions)	Validation Set (156 positions)
5	94.3	94.7
6	84.2	86.8
7	85.0	83.7
8	84.5	82.0
9	82.1	79.8
10	71.4	73.9
11	69.1	72.0
12	65.0	68.5
13	63.7	64.8
14	58.4	61.4

In summary, our experiments have showed that if forward pruning is already used, additional forward pruning at ALL nodes gives a significant improvement. Of course, it is possible that another forward-pruning mechanism might achieve the same results. For example, making the null-move mechanism at ALL nodes more aggressive is a possibility to gain reductions. In our experiments we have made the null move more aggressive by setting the reduction factor at 3 when searching an expected ALL node. In the next subsection we test the variable null-move bound.

6.5.3 Variable Null-Move Bound

Campbell and Goetsch (1990) introduced the idea of variable null-move bound, which was implemented and tested by Björnsson and Marsland (2001a). The idea is that a null-move cut-off can be forced if the returned null-move search value is larger than or equal to $\beta - t$, where t is the minimal value of a tempo. The value t is dependent on the evaluation function. This allows a larger part of the null-move searches to cause cut-offs. The pseudo code is given in Figure 6.5. We remark that the null move is considered as a regular move. Therefore, if a null move originates from a CUT node, its successor is an ALL node, and *vice versa.*

```
.......................................................
//Forward-pruning code
 if(node_type != PV_NODE && depth > 2){
   next = swapSides(node);
   if(node_type == ALL_NODE) bound = beta - t else bound = beta;
   value = -PVS(next, -bound, -bound+1, depth-1-R, -node_type);
   if(value >= bound) return beta;
 }
.......................................................
```

Figure 6.5: Pseudo code for variable null-move bound.

To compare variable null-move bound (VNMB) with MC-A we investigated in the third series of experiments whether VNMB at expected ALL nodes gives a reduction of nodes. The heuristic was tested against the same set of 171 positions with 10-ply searches. The search was not enhanced with MC-A. The parameter t was increased with a step size of 25 to find a value which gives the most reduction. The results are given in Figure 6.6. For comparison the results of the default search and the search enhanced with MC-A are also given.

We see that the best result for t is achieved when t lies between 200 and 350. A reduction of 16 per cent is achieved compared to the original one (of course with the null move and MC-C). But the search enhanced with MC-A is still 15 per cent faster. When t is larger than 500, the search enhanced with variable null-move bound searches more nodes than the original one. The forward pruning becomes too aggressive resulting in too many re-searches.

We additionally tested the combination of MC-A and VNMB with different t (see the graph denoted "Combination" in Figure 6.6). There was no setting of t for which the combination was better than the MC-A enhanced search. When t became

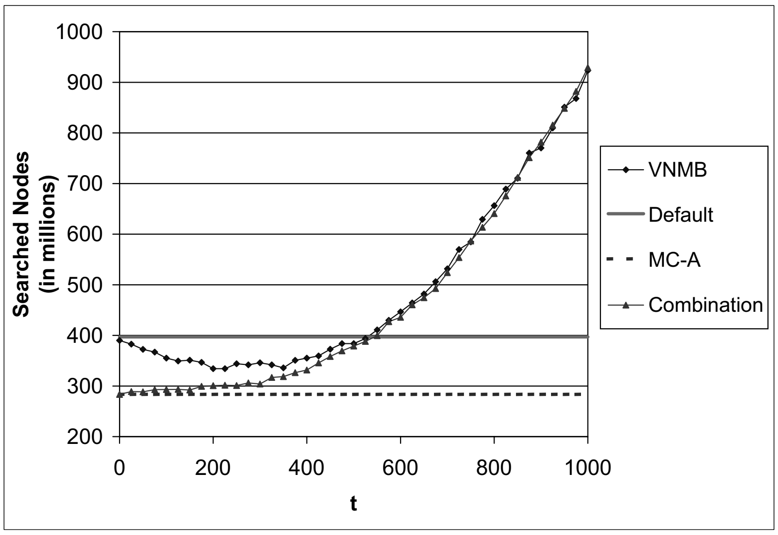

Figure 6.6: Variable null-move bound.

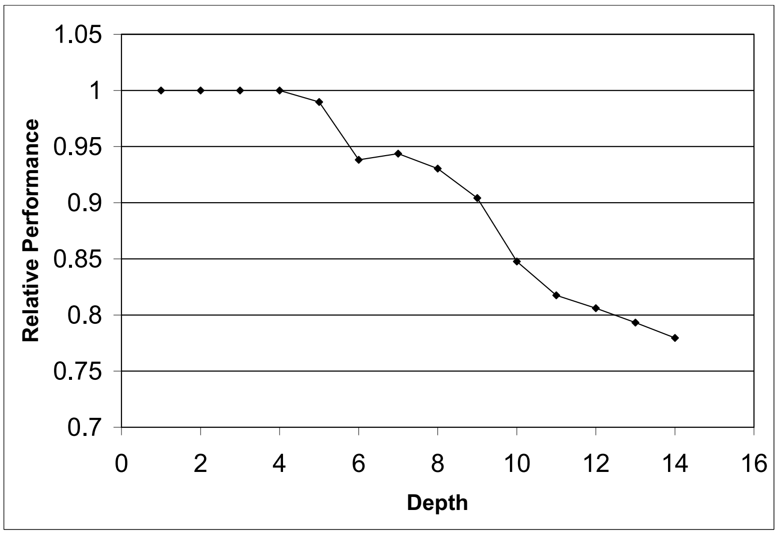

Figure 6.7: MC-A compared to variable null-move bound.

sufficiently large the combination behaved the same as the VNMB. In Figure 6.7 the relative performance of MC-A is compared to VNMB with the optimal parameter $t = 200$ for several search depths. We see that the search enhanced with MC-A searches fewer nodes than the one enhanced with VNMB. This difference increases with depth to some 22 per cent.

These experiments suggest that MC-A is not surpassed by aggressive null-move pruning at expected ALL nodes. We believe that MC-A is a significant enhancement in a PVS search where forward pruning is used. In general these experiments suggest that extra forward pruning at ALL nodes gives a safe reduction of the number of nodes searched.

6.5.4 Performance Enhancement with MC-A

In the fourth series of experiments we tested the possible increase of playing strength by using MC-A with the optimal parameter setting (of Subsection 6.5.1). Two versions of MIA were matched against each other, one using MC-A and the other without MC-A. These versions used all the enhancements described in Subsection 2.6. Both programs played 1000 games, switching sides halfway. The thinking time was limited to 60 seconds per move, simulating tournament conditions. To prevent that programs played the games over and over again, a random factor was included in the evaluation function (see Subsection 3.2.2). The results are given in Table 6.3.

Table 6.3: 1000-game match results.

	Score	Winning ratio
MC-A vs. Default	549-451	1.21

We observe that the modified version outplayed the original version with a winning ratio of 1.21 (i.e., scoring 21 per cent more winning points than the opponent). This result reveals that MC-A improves the playing strength of MIA significantly.

6.6 Chapter Conclusion and Future Research

This chapter showed that forward pruning at expected ALL nodes is safe and beneficial. Multi-cut at expected ALL nodes gives a safe reduction of approximately 40 per cent of the number of nodes searched in combination with null move and the regular multi-cut MC-C. Experiments suggested that parameters more aggressively chosen than MC-C lead to an additional improvement. We observed that MC-A still searches 22 per cent fewer nodes than variable null-move bound at expected ALL nodes. Moreover, MC-A was able to increase significantly the playing strength of the program MIA. From these observations we may conclude that MC-A is a valuable enhancement of PVS.

As future research, experiments are envisaged in other games to test the performance of MC-A. Whether MC-A surpasses an aggressive version of null move in other games also has to be tested. Finally, the combination of MC-A and variable null-move bound has to be tuned with different settings of C, M, R and t.

Chapter 7

The Relative History Heuristic

This chapter is an updated and abridged version of the following publication:

> Winands, M.H.M., Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2004c). The Relative History Heuristic. *Proceedings of the Fourth International Conference on Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu). Accepted for publication.[1]

Most modern game-playing computer programs successfully use the $\alpha\beta$ algorithm to reduce the search tree. The efficiency of $\alpha\beta$ search is dependent on the enhancements used (Campbell *et al.*, 2002). Move ordering is one of the main techniques to reduce the size of the search tree.

The history heuristic is a popular choice for ordering moves dynamically, in particular when other techniques are not applicable. In the past the butterfly heuristic (Hartmann, 1988) was proposed to replace the history heuristic, but it did not succeed. This chapter answers the fourth research question by proposing a new dynamic move-ordering variant, called the relative history heuristic, to replace the history heuristic.

The chapter is organised as follows. In Section 7.1 we review the history heuristic and the butterfly heuristic. Next, the relative history heuristic is introduced in Section 7.2. Subsequently, the results of the experiments are given in Section 7.3. Finally, in Section 7.4 we present our conclusion and propose topics for future research.

[1]Thanks are due to the Editors of *CG 2004* for giving permission of reusing relevant parts of the article in this thesis.

7.1 The History and the Butterfly Heuristic

The history heuristic is a simple, inexpensive way to reorder moves dynamically at interior nodes of search trees. It was invented by Schaeffer (1983) and has been adopted in several game-playing programs. Unlike the killer heuristic, which only maintains a history of the one or two best killer moves at each ply, the history heuristic maintains a history for every legal move seen in the game tree. Since there is only a limited number of legal moves, it is possible to maintain a score for each move in two (black and white) tables. At every interior node in the search tree the history-table entry for the best move found is incremented by a value (e.g., 2^d, where d is the depth of the subtree searched under the node). The best move is in this case defined as the move which either causes a β cut-off, or which causes the best score. When a new interior node is examined, moves are re-ordered by descending order of their scores. The scores in the tables can be maintained during the whole game. Each time a new search is started the scores are decremented by a factor (e.g., divided by 2). They are only reset to zero or to some default values at the beginning of a complete new game. Details on the effectiveness or the strategy to maintain history scores during the whole game are dependent on the domain or game program.

The history heuristic does not cost much memory. The history tables are defined as two tables with 4096 entries (64 from_squares $\times$ 64 to_squares), where each entry is 4 or 8 bytes large. These tables can be easily indexed by a 12-bit key representing the origin and destination. In the history table we have also defined moves which are illegal.

Hartmann (1988) called attention to two disadvantages of the history heuristic. (A) Quite some space for the history table is wasted, because space for illegal moves is reserved too. For instance, in the game of Chess 44 per cent of the possible moves are legal. In LOA, this number is even lower because knight moves are not allowed. This gives that 1456 of the 4096 moves are legal, meaning that only 36 per cent of the entries in the table are used. Although this waste of memory is not a problem for games with a small dimensionality of moves, it can be a problem for games with a larger dimensionality of moves (for instance Amazons). (B) Moreover, Hartmann pointed out that some moves are played less frequently than others. There are two reasons for this. (B1) The moves are less frequently considered as good moves. (B2) The moves occur less frequently as legal moves in a game. The disadvantage of the history heuristic is that it is biased towards moves that occur more often in a game than others. However, the history heuristic has as implicit assumption that all the legal moves occur roughly with the same frequency in the game (tree). So, in games where this condition approximately holds an absolute measure seems appropriate. But in other games where some moves occur more frequently than other moves, we should resort to other criteria. For instance, assume we have a move which is quite successful when applicable (e.g., it then causes a cut-off) but it does not occur so often as a legal move in the game tree. This move will not obtain a high history score and is therefore ranked quite low in the move ordering. Therefore a different valuation of such a move may be considered.

To counter some elements of the two disadvantages Hartmann (1988) proposed

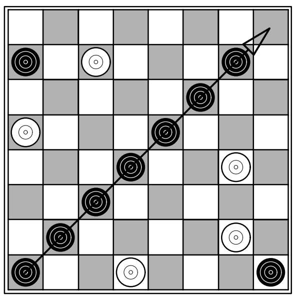
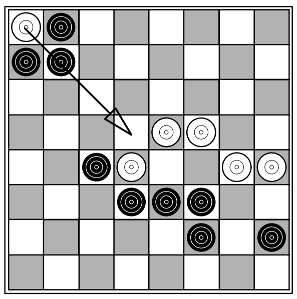

Figure 7.1: (a) Rare move. (b) Blocked move.

an alternative for the history heuristic, the butterfly heuristic. This heuristic takes the move frequencies in search trees into account. Two tables are needed (one for Black and one for White), called *butterfly boards*. They are defined in the same way as in the history heuristic (i.e., 64 from_squares × 64 to_squares). Any move that is not cut is recorded. Each time a move is executed in the search tree, its corresponding entry in the butterfly board (for each side) is also incremented by a value. Moves are now reordered by their butterfly scores. The butterfly heuristic was denied implementation by its inventor, since he expected that it would be far less effective than the history heuristic.

7.2 The Relative History Heuristic

We believe that we can considerably improve the performance of the history heuristic in some games by making it *relative* instead of absolute. The score used to order the moves (*movescore*) is given by the following formula:

$$movescore = \frac{hhscore}{bfscore} \tag{7.1}$$

where *hhscore* is the score found in the history table and *bfscore* is the score found in the butterfly board. We call this move ordering the relative history heuristic. We remark that we only *update* the entries of moves seen in the regular search, not in the quiescence search, because some (maybe better) moves are not investigated in the quiescence search. We *apply* the relative history heuristic everywhere in the tree (including in the quiescence search) for move ordering.

In some sense this heuristic is related to the realisation-probability search method (Tsuruoka, Yokoyama, and Chikayama, 2002). In that scheme the move frequencies gathered *offline* are used to limit or extend the search.

LOA is a nice test bed for the relative history heuristic. Dependent on the position some moves occur more often than others in the search tree. For example, moves going seven squares far are possible if and only if there are seven pieces of the same colour side by side on that line. In Figure 7.1a it is possible to move **a1-h8**, but it is very rare that in a real game a position occurs where seven pieces of the same colour occupy a diagonal. In contrast, consider a move like **a8-d5** that occurs regularly in a game, but in the position depicted in Figure 7.1b it will not often be applied in the corresponding search tree, since most of the time Black will not consider to move its piece on **b7**.

7.3 Experiments

In this section we show the results of various experiments with the relative history heuristic. This is done on a test set of 171 LOA positions.[2] We performed six series of experiments. In the first and second series, we tested the standard history heuristic and the relative history heuristic with different increments, respectively (Subsection 7.3.1). In the third, fourth, and fifth series, we compared the performance of the relative history heuristic with the standard history heuristic under different configurations (Subsection 7.3.2). Finally, in the sixth series of experiments, we tested the performance of the relative history heuristic in another domain, namely for 24 test positions for 6×6 Go (Subsection 7.3.3).

7.3.1 Increment Settings

In the following two series of experiments we tried to find the optimal increment setting for the history table and the butterfly board, which gave the largest node reduction. Using our set of 171 LOA positions, the program was tested for depth 14 using its normal enhancements as described in Subsection 2.6.

Table 7.1: Performance of the history heuristic with different increments on a test set of 171 positions.

History Increment	Total Nodes		
	depth 5	depth 9	depth 14
0	2,480,001	188,717,928	30,997,625,767
1	1,901,956	113,163,113	14,478,291,866
d	1,896,429	111,283,177	14,064,388,392
d^2	1,900,055	111,673,124	13,915,673,199
2^d	1,878,114	111,471,652	13,925,222,389

In the first series of experiments we tested the increments of the history table in a configuration where we used the standard history heuristic. Initially the history heuristic was developed for programs that were searching to a depth much less than 14. Considering the nature of a search tree, it might be that the best increment to be used depends on the search depth. Therefore we performed experiments for the original depths 5 and 9 used as test depths by Schaeffer (1983, 1989). The following

[2]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/TMP.zip.

increments were used: 1, d, d^2, and 2^d, where d is the explored depth of the move causing the cut-off. The increment 2^d is the standard increment of the history table. The result of the search without history heuristic (increment 0) is given for comparison. Table 7.1 shows that there is not much difference between the size of the search tree using increments d, d^2 and 2^d. For depth 14, the history heuristic gives in all the cases a reduction of approximately 55 per cent of the number of nodes searched. Unlike data for Chess, reported in Schaeffer (1989), we see a steady growth of the reduction with increasing search depth in LOA. Surprisingly, the increment of 1 is generating for the various depths a search tree that is only slightly larger than other increments. Apparently, the depth of the move explored is not so important in the history heuristic. Hence we may conclude that, unlike some data so far available for chess (Schaeffer, 1986), the choice of the increment is of little value in the test environment we studied.

Table 7.2: Performance of the relative history heuristic with different increments on a test set of 171 positions.

History Increment	Butterfly Increment	Total Nodes
1	1	12,261,241,807
1	d	12,544,923,748
1	d^2	12,936,458,114
1	2^d	12,741,580,747
d	1	12,654,462,037
d	d	12,433,892,630
d	d^2	12,914,014,566
d	2^d	13,075,535,903
d^2	1	12,501,473,310
d^2	d	12,238,059,952
d^2	d^2	12,509,830,417
d^2	2^d	12,234,575,562
2^d	1	12,354,762,028
2^d	d	13,081,065,785
2^d	d^2	12,928,253,841
2^d	2^d	12,954,976,931

In the second series of experiments we looked at various increment parameter settings of the history table and the butterfly board (h,b) in our engine using the relative history heuristic and using a search depth of 14 ply. In Table 7.2 the total number of nodes searched for each combination of parameters is given. In the process of parameter tuning we found that (d^2,2^d) is the most efficient. However, the difference with several other parameter configurations is not significant (e.g., (1,1), (d^2,d) or (2^d,1)). The difference between the best and worst parameter setting is 6 per cent in nodes searched. Hence, we may conclude that the exact choice of parameters seems to be not very critical.

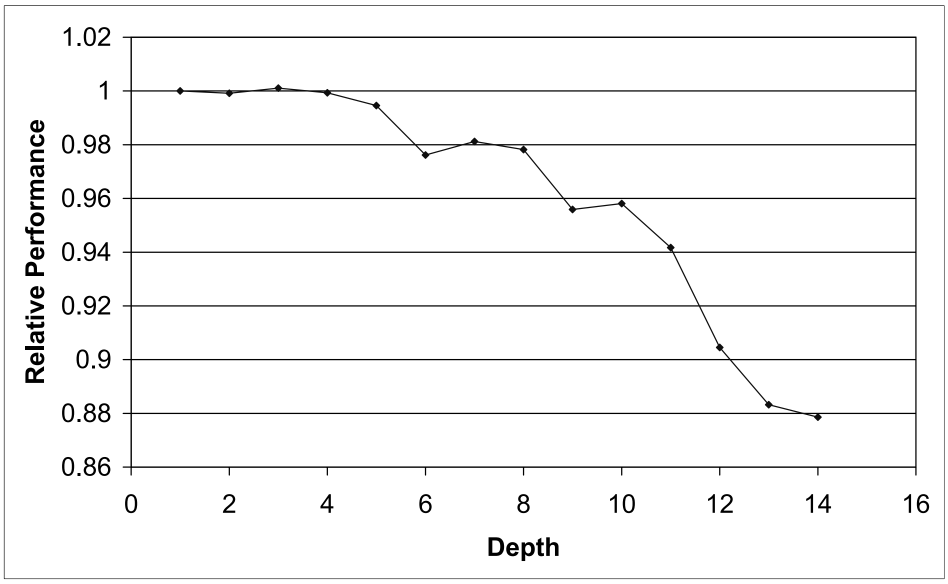

Figure 7.2: Performance of the relative history heuristic.

7.3.2 Performance in LOA

In the third series of experiments we tested the added value of the relative history heuristic, using the optimal parameter setting of the previous subsection, against the same set of 171 LOA positions for several depths in our original search engine under different conditions.

In Figure 7.2 we plot the relative performance of the two heuristics defined as the size of the search tree investigated using the relative history heuristic divided by that of the standard history heuristic, as a function of the search depth. We observe that until depth 8 there is no significant difference between the two move-ordering schemes. From depth 9 onwards the difference increases with the depth to some 12 per cent at depth 14. We see that the search enhanced with the relative history heuristic searches fewer nodes than the one enhanced with the standard setting.

Since we have tuned our heuristic with this particular test set (Subsection 7.3.1), we performed a fourth series of experiments on a set consisting of 156 different positions[3] to validate the result. The positions were searched up to depth 15. In Figure 7.3 we see the relative performance of the two heuristics on the validation set. If we compare Figure 7.2 with Figure 7.3, we see that similar results are achieved.

Since the performance of many search enhancements may to some extent depend on the search engine, we modified the search engine in the fifth series of experiments by switching the multi-cut forward-pruning mechanism off. Because of the diminished forward pruning the sizes of our search trees increased and we were not able to conduct experiments at depth 13 and further. Looking at Figure 7.4 we see that the

[3]The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/VMP.zip.

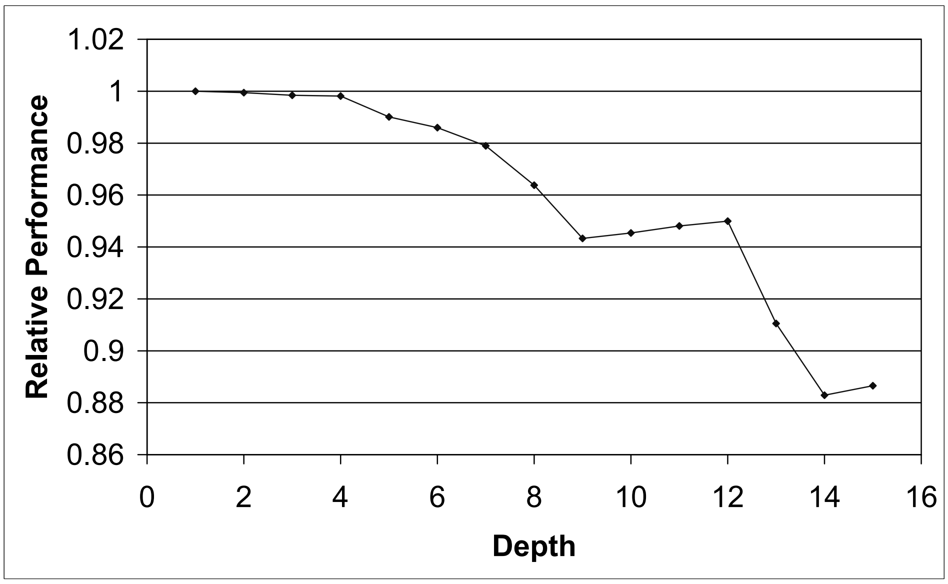

Figure 7.3: Validating the relative history heuristic.

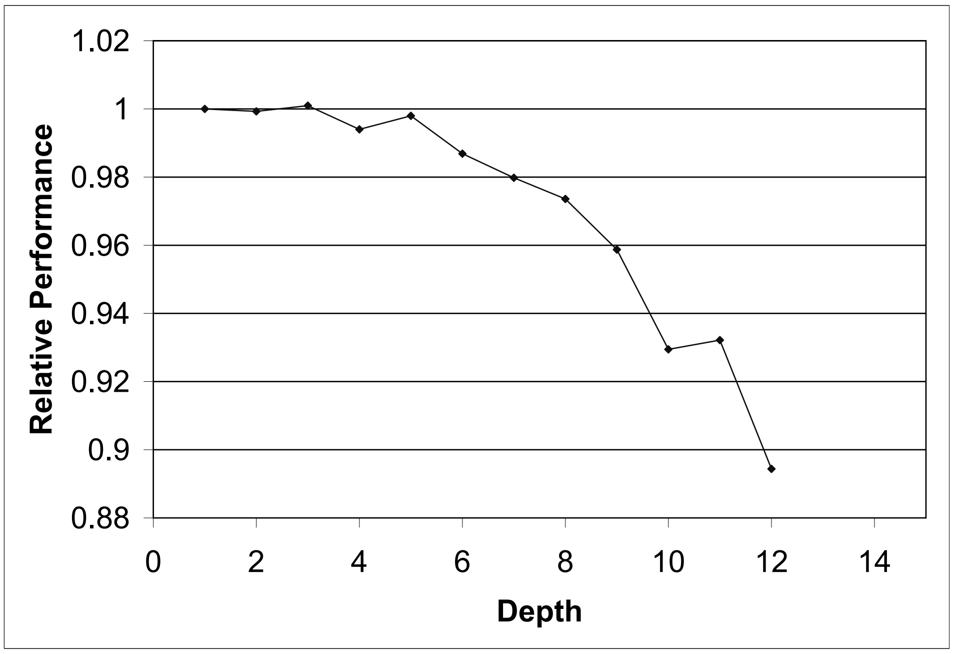

Figure 7.4: Relative history heuristic without using multi-cut.

relative history heuristic decreases the search with 11 per cent at depth 12. Hence, we may conclude that the same pattern as in the previous experiments has started. We expect that this pattern will continue, which is to be confirmed in the future by more powerful machines.

7.3.3 Performance in Go

The relative history heuristic was designed for LOA. To investigate whether the relative history heuristic would be interesting for other domains too, we tested its performance on the small-board games of Go in the sixth series of experiments, for which we used the program MIGOS[4] that recently had solved Go on the 5×5 board (Van der Werf, Van den Herik, and Uiterwijk, 2003).

MIGOS uses an iterative-deepening PVS with a transposition table with 2^{24} double entries (using the *two-deep* replacement scheme), enhanced transposition cut-offs, symmetry lookups in the transposition table, internal unconditional bounds, and an enhanced move ordering in which the history heuristic is an important component. The implementation of the history heuristic employs one shared table for both the black and white moves which exploits the game-dependent property in Go that moves on the same intersection are often good for both sides. After some parameter tuning for the relative history heuristic increments, which we optimised for solving the empty 5×5 board, we found that using d^3 for both the history and the butterfly board gave quite promising results[5].

The current challenge in small-board Go is solving the 6×6 board (5×5 is the largest square board solved by a computer). Therefore we decided to test the performance of the relative history heuristic on a set of 24 problems for the 6×6 board published in *Go World* by James Davies (1979, 1980). Figure 7.5 shows the average relative performance of the relative history heuristic compared to the standard settings without a butterfly table. Since we only used a small number of test positions we also plotted the standard deviations. They tend to increase with the search depth. The reasons for this are (1) the exponential effect of changes in the move ordering, and (2) a reduction in the number of positions because some positions are already solved at smaller depths. The results indicate again that for shallow searches not much should be expected of using the relative history heuristic. However, after about 10 ply the first improvements become noticeable and at about 15 ply the relative history heuristic achieves a reduction of roughly 13 per cent. However, we remark that the test set is too small to draw strong conclusions. So far the results are favourable for the relative history heuristic and they indicate that the relative history heuristic is worth investigating in other domains as well.

7.4 Chapter Conclusion and Future Research

Combining the ideas of the history heuristic and the butterfly heuristic resulted in the relative history heuristic. This heuristic does not suffer from underestimating

[4]The author would like to thank Erik van der Werf for his assistance in this experiment.

[5]We tested this combination on the LOA test set, too. Our experiments showed that this combination belongs to the better ones.

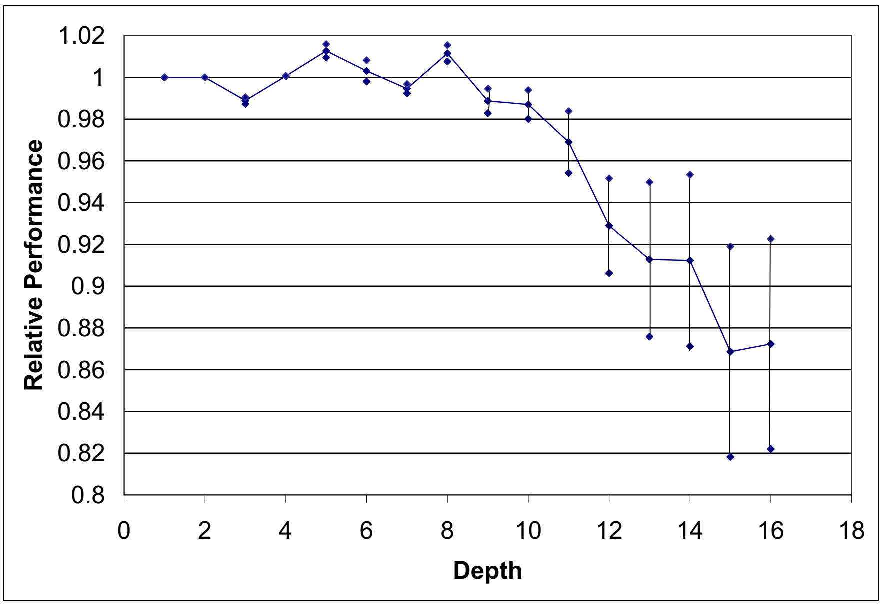

Figure 7.5: Performance of the relative history heuristic in 6×6 Go.

less frequently occurring moves in the search tree as the history heuristic does. We favour moves which are the good moves on average instead of moves which are the best move in absolute terms. Both the history heuristic and the relative history heuristic show a steady growth of the reduction with increasing search depth. Using the relative history heuristic our LOA program MIA searches even between 10 and 15 per cent fewer nodes (see Subsection 7.3.2). The results were confirmed by the Go program MIGOS. Hence, we may conclude that the relative history heuristic is a valuable technique to order the moves in a game tree of considerable depth (more than 12 plies).

It is remarkable that the utility of increments other than 1 does not show much better performance in the (relative) history heuristic for our LOA program MIA. The good performance of the increment of 1 could be the result of some domain-dependent properties.

Finally, it would be interesting for future research to test our heuristic in still more different games, especially in Chess, since the original history heuristic was developed for Chess.

Chapter 8

Conclusions and Future Research

In this thesis we investigated how search can be guided by knowledge in such a way that the search space is traversed efficiently and effectively. For this task we focussed on the question how to combine knowledge with search. This led to the formulation of our problem statement in Section 1.3. There we have posed four research questions that should be answered before we could address the problem statement. In this chapter we will provide our conclusions and provide topics for future research.

In Section 8.1 we will answer the four research questions one by one. We will formulate from these answers a reply to the problem statement in Section 8.2. Finally, in Section 8.3 we will provide promising directions for future research.

8.1 Conclusions on the Research Questions

The four research questions stated in Chapter 1 concern four topics of the knowledge-search trade-off, i.e., concerning (1) the evaluation function, (2) proof-number search, (3) forward pruning, and (4) move ordering. They are dealt with in the following subsections, respectively.

8.1.1 Evaluation Function

We noted that search needs at least terminal knowledge to solve a problem. Moreover, informed search cannot exist without a decent evaluation function for our test domain LOA. A challenge for a LOA program is building an evaluation function, which incorporates the basic principles of the game and increases the profitability. This challenge has led us to the first research question.

> **Research question 1**: *How can we build a strong evaluation function for Lines of Action?*

In our attempt to answer this question, we investigated which features were important for a LOA evaluation function. The features are based on the basic principles described in Chapter 2. It turned out that the following nine features were important: concentration, centralisation, centre-of-mass position, quads, mobility, walls, connectedness, uniformity, and player to move. These features have resulted in the evaluator MIA IV. The profitably of the evaluator was tested in a tournament against other LOA evaluators which performed well at previous Computer Olympiads. Experiments showed that MIA IV defeated them with quite large margins. Many features in the evaluation function do not consume much time. The program runs only 15 per cent slower with the MIA IV evaluator than with the MIA I evaluator. By using precomputed tables and caching, most features are quite straightforward to evaluate. The most important feature is concentration, followed by the mobility feature. All features are essential and contribute to the playing strength. There exist much interconnectedness and overlap between the features, which influence their performance. Therefore, all the features have to be simultaneously fine-tuned (or heuristically optimised) in a careful way. The combination of the nine features mentioned has resulted in an evaluation function that significantly increased the playing strength of our LOA program compared to programs with less-sophisticated evaluation functions.

8.1.2 Proof-Number Search

We saw that that the PN-search algorithms, PN, PN^2 and PDS, clearly outperformed $\alpha\beta$ in solving endgame positions in LOA. However, some memory problems made the plain PN search a weaker solver for the harder problems. PDS and PN^2 were able to solve significantly more problems than PN and $\alpha\beta$. PN^2 was restricted by its working memory, and PDS was considerably slower than PN^2. When reducing the need for memory at the cost of additional searching, we arrived at a crucial memory characteristic of knowledge in an informed-search process. This led to the second research question.

> **Research question 2**: *How can we develop a proof-number search algorithm, which is competitive in speed and not restricted in working memory?*

We presented a new proof-number search algorithm, called PDS-PN. It is a two-level search (like PN^2), which performs at the first level a depth-first Proof-number and Disproof-number Search (PDS), and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS.

We observed that PDS-PN was able to solve significantly more LOA endgame problems than $\alpha\beta$ search with enhancements. Moreover, the PDS-PN algorithm was almost as fast as PN^2 when the parameters (a,b) for its growth function were chosen properly. It turned out that for each a it held that the number of solved positions grew with increasing b, when the parameter b was still small. If b was sufficiently large, increasing it did not enlarge the number of solved positions. We observed that (1) PDS-PN solved more hard positions than PN^2 within an acceptable time frame and (2) PDS-PN was more effective than PN or even PN^2 because it did not run out of memory for hard problems. Moreover, PDS-PN performed quite well under harsh memory conditions. This is especially appropriate for hard problems and for environments with very limited memory such as hand-held computer platforms. Hence, we may conclude that within an acceptable time frame PDS-PN is more effective for really hard endgame positions than $\alpha\beta$ and any other PN-search algorithm.

8.1.3 Forward Pruning

We defined informed search as search, which is also using directing knowledge. Therefore, we investigated whether it is beneficial to improve forward-pruning methods, such as null move and multi-cut, in the Principal-Variation-Search (PVS) framework. This has led to the third research question.

> **Research question 3**: *How can we improve forward-pruning methods in the Principal-Variation-Search framework?*

Forward-pruning methods, such as multi-cut and null move, were tested at so-called ALL nodes. The Principal Variation Search was improved by four small but essential additions. The new PVS algorithm guarantees that forward pruning is safe at ALL nodes. Experiments showed that multi-cut at ALL nodes (MC-A) when combined with other forward-pruning mechanisms offered a significant reduction of the number of nodes searched. Multi-cut at expected ALL nodes gave a safe reduction of approximately 40 per cent of the number of nodes searched in combination with null move and the regular multi-cut MC-C. Experiments suggested that parameters more aggressively chosen than MC-C led to an additional improvement. In comparison, a (more) aggressive version of the null move (variable null-move bound) gave less reduction at expected ALL nodes than our algorithm. We demonstrated that the playing strength of MIA was significantly increased by MC-A. We have shown that our forward pruning method at expected ALL nodes is safe and beneficial. We may conclude that MC-A is a valuable forward-pruning enhancement of PVS.

8.1.4 Move Ordering

We remarked that move ordering is an instance of directing knowledge in an informed-search process. Especially in an $\alpha\beta$ search, move ordering is one of the main techniques to reduce the size of the search tree. One important category is dynamic move ordering, which is dependent on information gained during the search. This has led to the fourth research question.

> **Research question 4**: *How can we use information gained during the search to improve move ordering?*

Combining the ideas of the history heuristic and the butterfly heuristic resulted in the relative history heuristic. This heuristic does not suffer from underestimating less frequently occurring moves in the search tree as the history heuristic does. The relative history heuristic favours moves which are the good moves on average instead of moves which are the best move in absolute terms. The relative history heuristic showed a steady growth of the reduction with increasing search depth. Using the relative history heuristic our LOA program MIA searched even between 10 and 15 per cent fewer nodes. The results were confirmed by the Go program MIGOS (Van der Werf *et al.*, 2003). Hence, we may conclude that the relative history heuristic is a valuable dynamic move-ordering technique in a game tree of considerable depth (more than 12 plies). Finally, we remark that the utility of increments other than 1 does not show a much better performance in the (relative) history heuristic for our LOA program MIA. The good performance of the increment of 1 could be the result of some domain-dependent properties.

8.2 Conclusion on the Problem Statement

Our problem statement was:

> **Problem statement**: *How can we develop informed-search methods in such a way that programs significantly improve their performance in a given domain?*

Taking the answers to the research questions above into account we see that there are several successful ways to improve the performance of informed-search methods. Our improvements of the evaluation function, proof-number search, forward pruning, and move ordering have given significant results in our LOA test domain.

8.3 Recommendations for Future Research

We complete this chapter by listing six recommendations for future research.

1. **Improving the evaluation function.** More patterns of blocked pieces, better distinction of move types in the mobility component, and additional knowledge whether a connection is important are some of the issues which could improve the evaluator. There is still room to fine tune certain weights and parameters in the evaluation function. Finally, we believe that combining the ideas of the strong programs YL and MONA with MIA IV would probably further increase the playing strength significantly.

2. **Automatic feature extraction.** Although the weights are (partially) automatically tuned by TD-learning, the features of our evaluation function are mostly selected by hand (sometimes with the help of statistical data analysis). We believe that ways to perform automatic feature extraction will be a valuable tool for building evaluation functions for unknown games.

3. **Improving PN-$\alpha\beta$.** At the moment there is no dynamic strategy available which determines when to use PN-$\alpha\beta$ search instead of $\alpha\beta$. A possible solution is the use of machine-learning methods that decide when and for how long to use PN search.

4. **Improving the MC-A.** The combination of MC-A and variable null-move bound has to be tuned with different settings of C, M, R and t. Instead of initialising the parameters with fixed values, a variable scheme dependent on the game state is more appropriate.

5. **Reviving the countermove heuristic.** The relative history heuristic was partially based on the forgotten butterfly heuristic. It would be interesting to revisit another dynamic move-ordering technique, the countermove heuristic (Schaeffer, 1986; Uiterwijk, 1992).

6. **Application to other domains.** We believe that an adequate challenge is testing PDS-PN in other domains with difficult endgames. An example of a game notoriously known for its difficult endgames is the game of Tsume-Shogi (a variant of Shogi). Several hard problems including solutions over a few hundred ply are solved by PN* (Seo *et al.*, 2001) and PDS (Sakuta and Iida, 2001; Nagai, 2002). It would be interesting to test PDS-PN on these problems. Next, experiments are envisaged in other games to test the performance of MC-A. Whether MC-A surpasses an aggressive version of null move in other games has to be tested, too. Finally, it would be interesting for future research to test our relative history heuristic in other different games, especially in Chess, since the original history heuristic was developed for Chess.

References

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466–473, ACM Press, New York, NY, USA. [6, 16]

Allis, L.V. (1988). A Knowledge-Based Approach of Connect Four: The Game is Over, White to Move Wins. M.Sc. thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. Report No. IR-163. [2]

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands. [2, 11, 36, 37, 38, 39, 40, 44, 49, 53, 57]

Allis, L.V., Herik, H.J. van den, and Herschberg, I.S. (1991a). Which Games Will Survive? *Heuristic Programming in Artificial Intelligence 2: the Second Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 232–243. Ellis Horwood, Chichester, England. [1, 11]

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1991b). Databases in Awari. *Heuristic Programming in Artificial Intelligence 2: the Second Computer Olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 73–86. Ellis Horwood, Chichester, England. [44]

Allis, L.V., Meulen, M. van der, and Herik, H.J. van den (1994). Proof-Number Search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–123. [5, 36, 37, 38]

Anantharaman, T.S., Campbell, M., and Hsu, F.-h. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *ICCA Journal*, Vol. 11, No. 4, pp. 135–143. Also published (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109. [64]

Anshelevich, V.V. (2002). A Hierarchical Approach to Computer Hex. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 101–120. [2, 9]

Beal, D.F. (1984). Mixing Heuristic and Perfect Evaluations: Nested Minimax. *ICCA Journal*, Vol. 7, No. 1, pp. 10–15. [43]

Beal, D.F. (1989). Experiments with the Null Move. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 65–79. Elsevier Science Publishers, Amsterdam, The Netherlands. [64]

Berkey, D.D. (1988). *Calculus*. Saunders College Publishing, New York, NY, USA. [39, 52]

Berliner, H.J. (1979). The B*-Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, Vol. 12, No. 1, pp. 23–40. [36]

Berliner, H.J. (1984). Search vs. Knowledge: An Analysis from the Domain of Games. *Artificial and Human Intelligence*, pp. 105–117, Elsevier Science Publishers B.V., Amsterdam, The Netherlands. [4]

Berliner, H.J., Goetsch, G., Campbell, M.S., and Ebeling, C. (1990). Measuring the Performance Potential of Chess Programs. *Artificial Intelligence*, Vol. 43, No. 1, pp. 7–20. [3]

Billings, D. and Björnsson, Y. (2002). *Mona and YL's Lines of Action Page.* http://www.cs.ualberta.ca/∼games/LOA. [28]

Billings, D. and Björnsson, Y. (2003). Search and Knowledge in Lines of Action. *Advances in Computer Games 10: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 231–248. Kluwer Academic Publishers, Boston, MA, USA. [5, 11, 15, 19, 26]

Billings, D., Davidson, A., Schaeffer, J., and Szafron, S. (2000). The Challenge of Poker. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 201–240. [2]

Björnsson, Y. (2000). YL Wins Lines of Action Tournament. *ICGA Journal*, Vol. 23, No. 3, pp. 179–180. [28, 101]

Björnsson, Y. (2002). *Selective Depth-First Game-Tree Search.* Ph.D. thesis, University of Alberta, Edmonton, Canada. [15, 16, 63, 64, 70]

Björnsson, Y. and Marsland, T.A. (1999). Multi-Cut Alpha-Beta Pruning. *Computers and Games, Lecture Notes in Computing Science 1558* (eds. H.J. van den Herik and H. Iida), pp. 15–24. Springer-Verlag, Berlin, Germany. [16, 64, 66]

Björnsson, Y. and Marsland, T.A. (2001a). Risk Managament in Game-Tree Pruning. *Information Sciences*, Vol. 122, No. 1, pp. 23–41. [72]

Björnsson, Y. and Marsland, T.A. (2001b). Multi-Cut Alpha-Beta Pruning in Game-Tree Search. *Theoretical Computer Science*, Vol. 252, Nos. 1–2, pp. 177–196. [68, 70]

Björnsson, Y. and Winands, M.H.M. (2001). YL Wins Lines of Action Tournament. *ICGA Journal*, Vol. 24, No. 3, pp. 180–181. [28, 102]

Björnsson, Y. and Winands, M.H.M. (2002). YL Wins Lines of Action Tournament. *ICGA Journal*, Vol. 25, No. 3, pp. 185–186. [28, 103]

Björnsson, Y., Marsland, T.A., Schaeffer, J., and Junghans, A. (1997). Searching with Uncertainty Cut-Offs. *ICCA Journal*, Vol. 20, No. 1, pp. 29–37. [65]

Breuker, D.M. (1998). *Memory versus Search in Games.* Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [3, 36, 39, 46, 53, 111]

Breuker, D.M., Allis, L.V., and Herik, H.J. van den (1994a). How to Mate: Applying Proof-Number Search. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 251–272. University of Limburg, Maastricht, The Netherlands. [36]

Breuker, D.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Allis, L.V. (1994b). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 14, No. 4, pp. 183–193. [44]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1996). Replacement Schemes and Two-Level Tables. *ICCA Journal*, Vol. 19, No. 3, pp. 175–180. [6, 16, 40, 48]

Breuker, D.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Allis, L.V. (2001a). A Solution to the GHI Problem for Best-First Search. *Theoretical Computer Science*, Vol. 252, Nos. 1–2, pp. 121–149. [40, 49]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001b). The PN^2-Search Algorithm. *Advances in Computer Games 9* (eds. H.J. van den Herik and B. Monien), pp. 115–132, IKAT, Universiteit Maastricht, Maastricht, The Netherlands. [36, 39, 40, 57]

Buro, M. (1995). ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal*, Vol. 18, No. 2, pp. 71–76. [64]

Buro, M. (1997). The Othello Match of the Year: Takeshio Murakami vs. Logistello. *ICCA Journal*, Vol. 20, No. 3, pp. 189–193. [2]

Buro, M. (2000). Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. *Games in AI Research* (eds. H.J. van den Herik and H. Iida), pp. 77–96. Universiteit Maastricht, Maastricht, The Netherlands. [64]

Bush, D. (2000). An Introduction to TwixT. *Abstract Games*, Vol. 1, No. 2, pp. 9–12. [9]

Bushinsky, S. (2004). Personal Communication. [20]

Campbell, M., Hoane, A.J. Jr., and Hsu, F.-h. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 57–83. [36, 75]

Chaunier, C. and Handscomb, K. (2001). Lines of Action Strategic Ideas – Part 4. *Abstract Games*, Vol. 2, No. 1, pp. 12–14. [5, 11, 26]

Davies, J. (1979). Small-Board Problems. *Go World*, Vol. 14–16, pp. 55–56. [82]

Davies, J. (1980). Go in Lilliput. *Go World*, Vol. 17, pp. 55–56. [82]

Donkers, H.H.L.M. (2003). *Nosce Hostem: Searching with Opponent Models.* Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [4]

Donkers, H.H.L.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003). Admissibility in Opponent-Model Search. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 119–140. [15]

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143. [16, 64]

Dyer, D. (2000). *Lines of Action Homepage*. http://www.andromeda.com/people/ddyer/loa/loa.html. [15]

Eppstein, D. (1997). Dynamic Connectivity in Digital Images. *Information Processing Letters*, Vol. 62, No. 3, pp. 121–126. [15]

Feldmann, R. (1997). Fail High Reductions. *Advances in Computer Chess 8* (eds. H.J. van den Herik and J.W.H.M Uiterwijk), pp. 111–127. Universiteit Maastricht, Maastricht, The Netherlands. [65]

Fraenkel, A.S. (1996). Combinatorial Games: Selected Bibiliography with a Succint Gourmet Introduction. *Games of No Chance. Combinatorial Games at MSRI 1994* (ed. R.J. Nowakowski), pp. 493–537. Cambridge University Press, Cambridge, England. [2]

Gasser, R.U. (1995). *Harnessing Computational Resources for Efficient Exhaustive Search*. Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, Switzerland. [2]

Goetsch, G. and Campbell, M.S. (1990). Experiments with the Null-Move Heuristic. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 159–168. Springer-Verlag, New York, NY, USA. [64, 72]

Gray, S.B. (1971). Local Properties of Binary Images in Two Dimensions. *IEEE Transactions on Computers*, Vol. 20, No. 5, pp. 551–561. [23]

Guibert, N. and Wesselink, W. (2003). The Revenge Match Samb – Buggy. *ICGA Journal*, Vol. 26, No. 2, pp. 126–131. [2]

Handscomb, K. (2000a). Lines of Action Strategic Ideas – Part 1. *Abstract Games*, Vol. 1, No. 1, pp. 9–11. [5, 11, 13, 25]

Handscomb, K. (2000b). Lines of Action Strategic Ideas – Part 2. *Abstract Games*, Vol. 1, No. 2, pp. 18–19. [5, 11, 12, 13]

Handscomb, K. (2000c). Lines of Action Strategic Ideas – Part 3. *Abstract Games*, Vol. 1, No. 3, pp. 18–19. [5, 11, 12, 14]

Hartmann, D. (1988). Butterfly Boards. *ICCA Journal*, Vol. 11, Nos. 2–3, pp. 64–71. [6, 75, 76]

Hashimoto, T., Nagashima, J., Sakuta, M., Uiterwijk, J.W.H.M., and Iida, H. (2003). Automatic Realization-Probability Search. Internal report, Dept. of Computer Science, University of Shizuoka, Hamamatsu, Japan. [15, 24]

Heinz, E.A. (1999). Adaptive Null-Move Pruning. *ICCA Journal*, Vol. 22, No. 3, pp. 123–132. [16]

Heinz, E.A. (2000). *Scalable Search in Computer Chess.* Vieweg Verlag, Braunschweig, Germany. [3]

Herik, H.J. van den and Iida, H. (eds.) (2000). *Games in AI Research.* Universiteit Maastricht, Maastricht, The Netherlands. [1]

Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games Solved, Now and in the Future. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 277–311. [2]

Hsu, F.-h. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion.* Princeton University Press, Princeton, NJ, USA. [2]

Irving, G., Donkers, H.H.L.M., and Uiterwijk, J.W.H.M. (2000). Solving Kalah. *ICGA Journal*, Vol. 23, No. 3, pp. 139–1–48. [2]

Junghanns, A. and Schaeffer, J. (1997). Search versus Knowledge in Game-Playing Programs Revisited. *IJCAI-97*, pp. 692–697. [3]

Kishimoto, A. and Müller, M. (2003a). A Solution to the GHI Problem for Depth-First Proof-Number Search. *Proceedings of the 7th Joint Conference on Information Sciences (JCIS 2003)* (ed. P. Wang *et. al*), pp. 489–492, JCIS/Association for Intelligent Machinery, Inc, USA. [36, 40]

Kishimoto, A. and Müller, M. (2003b). Df-pn in Go: An Application to the One-Eye Problem. *Advances in Computer Games 10: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 125–141. Kluwer Academic Publishers, Boston, MA, USA. [40]

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [15, 36, 64]

Kocsis, L. (2003). *Learning Search Decisions.* Ph.D. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [6]

Kocsis, L., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001a). Learning Time Allocation using Neural Networks. *Computers and Games, Lecture Notes in Computer Science 2063* (eds. T.A. Marsland and I. Frank), pp. 170–185, Springer-Verlag, Berlin, Germany. [15]

Kocsis, L., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001b). Move Ordering using Neural Networks. *Engineering of Intelligent Systems, Lecture Notes in Artificial Intelligence, Vol. 2070* (eds. L. Montosori, J. Váncza, and M. Ali), pp. 45–50. Springer-Verlag, Berlin, Germany. [6, 15]

Levy, D. (2003a). The State of the Art in Man vs. Machine Chess. *ICGA Journal*, Vol. 26, No. 1, pp. 3–8. [2]

Levy, D. (2003b). Kasparov vs. X3D Fritz. *ICGA Journal*, Vol. 26, No. 4, pp. 289–290. [2]

Marsland, T.A. (1983). Relative Efficiency of Alpha-Beta Implementations. *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI-83)*, pp. 763–766, Karlsruhe, Germany. [63, 65]

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19. [15]

Marsland, T.A. and Björnsson, Y. (2001). Variable-Depth Search. *Advances in Computer Games 9* (eds. H.J. van den Herik and B. Monien), pp. 9–24. Universiteit Maastricht, Maastricht, The Netherlands. [5, 65]

Marsland, T.A. and Campbell, M. (1982). Parallel Search of Strongly Ordered Game Trees. *Computing Surveys*, Vol. 14, No. 4, pp. 533–551. [16]

Marsland, T.A. and Popowich, F. (1985). Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, Vol. 7, No. 4, pp. 442–452. [64]

McAllester, D.A. (1988). Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, Vol. 35, No. 1, pp. 278–310. [36]

Müller, K. (2002). The Clash of the Titans: Kramnik – Fritz Bahrain. *ICGA Journal*, Vol. 25, No. 4, pp. 233–238. [2]

Müller, K. (2003). Man Equals Machine In Chess. *ICGA Journal*, Vol. 26, No. 1, pp. 9–13. [2]

Nagai, A. (1998). A New AND/OR Tree Search Algorithm using Proof Number and Disproof Number. *Proceedings of Complex Games Lab Workshop*, pp. 40–45, ETL, Tsukuba, Japan. [37, 40, 48, 49, 53, 114]

Nagai, A. (1999). A New Depth-First-Search Algorithm for AND/OR Trees. M.Sc. thesis, The University of Tokyo, Tokyo, Japan. [40, 43]

Nagai, A. (2002). *Df-pn Algorithm for Searching AND/OR Trees and its Applications*. Ph.D. thesis, The University of Tokyo, Tokyo, Japan. [40, 61, 89]

Nagai, A. and Imai, H. (1999). Application of Df-pn+ to Othello Endgames. *Proceedings of Game Programming Workshop in Japan '99*, pp. 16–23, Hakone, Japan. [37]

Nalimov, E.V., Haworth, G.M^{c}C., and Heinz, E.A. (2000). Space-Efficient Indexing of Chess Endgame Tables. *ICGA Journal*, Vol. 23, No. 3, pp. 148–162. [36]

Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de (1996). Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, Vol. 87, No. 2, pp. 255–293. [36]

Reinefeld, A. (1983). An Improvement to the Scout Search Tree Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14. [16, 63]

Romein, J.W. and Bal, H.E. (2003). Solving the Game of Awari using Parallel Retrograde Analysis. *IEEE Computer*, Vol. 36, No. 10, pp. 26–33. [2]

Sackson, S. (1969). *A Gamut of Games.* Random House, New York, NY, USA. [9]

Sakuta, M. (2001). *Deterministic Solving of Problems with Uncertainty.* Ph.D. thesis, Shizuoka University, Hamamatsu, Japan. [40]

Sakuta, M. and Iida, H. (2001). The Performance of PN*, PDS and PN Search on 6×6 Othello and Tsume-Shogi. *Advances in Computer Games 9* (eds. H.J. van den Herik and B. Monien), pp. 203–222. Universiteit Maastricht, Maastricht, The Netherlands. [36, 40, 43, 61, 89]

Sakuta, M., Hashimoto, T., Nagashima, J., Uiterwijk, J.W.H.M., and Iida, H. (2003). Application of the Killer-tree Heuristic and the Lamba-Search Method to Lines of Action. *Information Sciences*, Vol. 154, Nos. 3–4, pp. 141–155. [15, 17]

Samuel, A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, pp. 210–229. [1]

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19. [6, 16, 76, 78]

Schaeffer, J. (1984). The Relative Importance of Knowledge. *ICCA Journal*, Vol. 7, No. 3, pp. 138–145. [23]

Schaeffer, J. (1986). *Experiments in Search and Knowledge.* Ph.D. thesis, Department of Computing Science, University of Waterloo, Waterloo, Canada. [3, 34, 79, 89]

Schaeffer, J. (1989). The History Heuristic and the Performance of Alpha-Beta Enhancements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212. [78, 79]

Schaeffer, J. (1990). Conspiracy Numbers. *Artificial Intelligence*, Vol. 43, No. 1, pp. 67–84. [36]

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag, New York, NY, USA. [2, 11]

Schaeffer, J. and Lake, R. (1996). Solving the Game of Checkers. *Games of No Chance* (ed. R.J. Nowakowski), pp. 119–133. Cambridge University Press, Cambridge, England. [36]

Schaeffer, J. and Plaat, A. (1996). New Advances in Alpha-Beta Searching. *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*, pp. 124–130. ACM Press, New York, NY, USA. [16]

Seo, M., Iida, H., and Uiterwijk, J.W.H.M. (2001). The PN*-Search Algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, Vol. 129, Nos. 1–2, pp. 253–277. [36, 39, 61, 89]

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275. [1, 11]

Sheppard, B. (2002). World-Championship-Caliber Scrabble. *Artificial Intelligence*, Vol. 134, pp. 241–275. [2]

Tesauro, G.J. (1994). TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, Vol. 6, pp. 215–219. [2]

Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002). Game-tree Search Algorithm based on Realization Probability. *ICGA Journal*, Vol. 25, No. 3, pp. 132–144. [77]

Turing, A.M. (1953). Digital Computers Applied to Games. *Faster Than Thought* (ed. B.V. Bowden), pp. 286–297, Pitman Publishing, London, England. [1]

Uiterwijk, J.W.H.M. (1992). The Countermove Heuristic. *ICCA Journal*, Vol. 15, No. 1, pp. 8–15. [89]

Uiterwijk, J.W.H.M. and Herik, H.J. van den (2000). The Advantage of the Initiative. *Information Sciences*, Vol. 122, No. 1, pp. 43–58. [14, 26]

Wágner, J. and Virág, I. (2001). Solving Renju. *ICGA Journal*, Vol. 24, No. 1, pp. 30–34. [2]

Wellman, M.P. (1990). Fundamental Concepts of Qualitative Probablistic Networks. *Artificial Intelligence*, Vol. 44, No. 3, pp. 257–303. [32]

Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2003). Solving Go on Small Boards. *ICGA Journal*, Vol. 26, No. 2, pp. 92–107. [82, 88]

Winands, M.H.M. (2000). Analysis and Implementation of Lines of Action. M.Sc. thesis, Universiteit Maastricht, Maastricht, The Netherlands. [11, 26, 32]

Winands, M.H.M. (2003). MIA IV wins Lines of Action Tournament. *ICGA Journal*, Vol. 26, No. 4, pp. 264–265. [28, 105]

Winands, M.H.M. and Uiterwijk, J.W.H.M. (2001). PN, PN^2 and PN* in Lines of Action. *The CMG Sixth Computer Olympiad Computer-Games Workshop Proceedings* (ed. J.W.H.M. Uiterwijk), Technical Reports in Computer Science CS 01-04, Universiteit Maastricht, Maastricht, The Netherlands. [35]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001a). The Quad Heuristic in Lines of Action. *ICGA Journal*, Vol. 24, No. 1, pp. 3–15. [9, 11, 16, 19, 23, 28]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001b). Combining Proof-Number Search with Alpha-Beta Search. *Proceedings of the Thirteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)* (eds. B. Kröse, M. de Rijke, G. Schreiber, and M. van Someren), pp. 299–306, Universiteit van Amsterdam, Amsterdam, The Netherlands. [35]

Winands, M.H.M., Kocsis, L., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2002). Temporal Difference Learning and the Neural MoveMap Heuristic in the Game of Lines of Action. *GAME-ON 2002 3rd International Conference on Intelligent Games and Simulation* (eds. Q. Mehdi, N. Gough, and M. Cavazza), pp. 99–103, SCS Europe Bvba, Ghent, Belgium. [22, 28]

Winands, M.H.M., Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2003a). An Evaluation Function for Lines of Action. *Advances in Computer Games 10: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 249–260. Kluwer Academic Publishers, Boston, MA, USA. [19]

Winands, M.H.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Werf, E.C.D. van der (2003b). Enhanced Forward Pruning. *Proceedings of the 7th Joint Conference on Information Sciences (JCIS 2003)* (ed. P. Wang *et al.*), pp. 485–488, JCIS/Association for Intelligent Machinery, Inc, USA. [63]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2003c). PDS-PN: A New Proof-Number Search Algorithm: Application to Lines of Action. *Computers and Games, Lecture Notes in Computer Science 2883* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), pp. 170–185, Springer-Verlag, Berlin, Germany. [35, 47]

Winands, M.H.M., Herik, H.J. van den, Uiterwijk, J.W.H.M., and Werf, E.C.D. van der (2004a). Enhanced Forward Pruning. *Information Sciences*. Accepted for publication. [63]

Winands, M.H.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2004b). An Effective Two-Level Proof-Number Search Algorithm. *Theoretical Computer Science*, Vol. 313, No. 3, pp. 511–525. [47]

Winands, M.H.M., Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2004c). The Relative History Heuristic. *Proceedings of the Fourth International Conference on Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu). Accepted for publication. [75]

Zobrist, A.L. (1970). A New Hashing Method for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2, pp. 69–73. [16, 44]

Appendix A

LOA at the Computer Olympiads

The Computer Olympiad is a multi-games event in which all participants are computer programs. The Olympiad is a brainchild of David Levy, who organised this tournament in London in 1989 for the first time. The list of the nine Computer Olympiads so far is: London (1989), London (1990), Maastricht (1991), London (1992), London (2000), Maastricht (2001), Maastricht (2002), Graz (2003), and Ramat-Gan (2004). The purpose of the Olympiad is to find the strongest program for each game. Some teams arrive with the clear goal of winning the tournament, other teams come to participate only, and a third type of participants enters the Olympiad to test new ideas under tournament conditions. The Olympiad is a truly international event. Participants come from all over the world: USA, Canada, Japan, Taiwan, China, Israel, and the European Union. The event is held under the auspices of the ICGA (International Computer Games Association). Since 2000 there has been a LOA computer tournament. In this tournament each program must complete its moves for one game in 30 minutes. Programs play four games against each other.

The Fifth Computer Olympiad

The fifth Computer Olympiad was held in London in 2000. Three programs competed in the LOA tournament: YL, MONA, and MIA I (see Björnsson, 2000 for a detailed report). The program YL written by Yngvi Björnsson (Iceland) was the winner of the University of Alberta LOA computer championship (2000), finishing with a perfect score of 22-0, ahead of MONA (second place) and 10 other LOA programs. One of its strengths was that it searched extremely fast and could therefore analyse positions more deeply than other programs (and humans). Its evaluation function was carefully tuned with the help of a Temporal-Difference learning method. The program MONA was written by Darse Billings (Canada). It is well-known for its established impressive track record when playing email games. The program has won each and every email game it has played so far, including games against some of the world's elite players. The program used extensive game-specific knowledge and understood concepts that human players frequently use, such as a main group, bad outlier pieces, threats, and good vs. bad blockades. MIA I is described in this thesis. The final standings of the LOA tournament are shown in Table A.1. YL took the gold medal, MONA was awarded silver, and MIA I received the bronze medal.

Table A.1: The final standings of the LOA tournament at the 5th Computer Olympiad.

Place	Program	YL	MONA	MIA I	Points
1	YL	-	3-1	3-1	6
2	MONA	1-3	-	3-1	4
3	MIA I	1-3	1-3	-	2

The CMG Sixth Computer Olympiad

The sixth Computer Olympiad was held in Maastricht in 2001. Again three programs competed in the LOA tournament. This time, they were: YL, MIA II, and APPRENTICE (see Björnsson and Winands, 2001 for a detailed report). The version of YL that participated was fundamentally the same as last year's, although some small, but important improvements were made. Primarily, the new version had a better understanding of blocked positions; the search algorithm was augmented with multi-cut pruning (giving an additional ply of search), repetition detection, and an enhanced move-ordering mechanism; finally, some minor bugs were fixed here and there. MIA II is described in this thesis. APPRENTICE was a newcomer to the scene (although its author Don Beal (UK) is a veteran computer-game programmer). It utilised the same search engine as was implemented in Beal's chess program and as such it employed many of the standard game-tree search techniques. However, its evaluation function was rather rudimentary. It evaluated how close each side was to connecting by approximating how many moves it would take for the side to connect if it were allowed to move its pieces freely.

The final standings of the LOA tournament are shown in Table A.2. After the normal rounds, YL and MIA II tied for the first place. Thus a play-off match was necessary to decide the winner. Two games of 30 minutes each were played and YL won them both.

YL and MIA II were noticeably stronger than they were last year. This was apparent from the games they played each other. They turned out to be long and hard-fought fights.

Table A.2: The final standings of the LOA tournament at the CMG 6th Computer Olympiad.

Place	Program	YL	MIA II	APPRENTICE	Points
1	YL	-	2-2	4-0	6 (+2)
2	MIA II	2-2	-	4-0	6
3	APPRENTICE	0-4	0-4	-	0

Selected Games

MIA II vs. YL (Game 2)
1. c8-c6 a4-c2 2. d8-b6 a6-c4 3. b8-b5 h2-e2 4. g8-g6 a2-d2 5. g6xc2 h4-f2 6. b6xf2 h6-e3 7. e8-e4 a3-c5 8. g1-g2 h3-g4 9. g2xg4 h5-d5 10. g4-d4 a5-b6 11. f8-f5 h7-f7 12. e1-g3 e3-c3 13. g3-e3 c3xe3 14. f2-g3 e3-c3 15. g3-e3 c3xe3 16. e4-e7 c5xe7 17. f5-f2 d5xb5 18. c6-e6 b5-c6 19. b1-d3 c6xc2 20. f1-f4 f7xf4 21. e6xc4 e7-e4 22. c4-b3 f4-g3 23. c1-a1 a7-b7 24. d4-b4 b7-d5 25. a1-b2 c2-c3 26. b3xd5 b6-c7 27. b2-a3 c7-e5 28. b4-d4 g3-g4 29. d5-e6 e5-f4 0-1

YL vs. MIA II (Game 3)
1. d1-b3 h4-f2 2. b1-b4 h7-f7 3. g8-g6 h2-e2 4. f8-c5 h3-e3 5. b8-b5 h5-d5 6. c1-c4 h6-h5 7. f1-g2 h5xc5 8. c8xc5 e3xb3 9. d8-b6 d5-d6 10. g6-f5 f2xf5 11. e1-f2 e2-f3 12. g1-g3 a2-d2 13. c4-c6 a7-a2 14. e8xa4 f7-c4 15. c6xf3 b3xf3 16. f2-h4 d2-d4 17. g2-h3 f5-e4 18. h4-h6 e4-c6 19. h6-g7 a3-e3 20. g3-e5 a2-b2 21. e5xe3 b2-c1 22. e3xc1 f3-d5 23. g7-g8 c4-f7 24. b4-b7 f7-d7 25. g8-g7 a5-b4 26. b6-a5 d6-c7 27. a4-b3 c6-e6 28. b7xd5 d7xb5 29. h3-g2 e6-c6 30. g2-f1 d4-f4 31. d5-d6 f4-d4 32. c1-d2 a6-b7 33. f1-g2 b7xb3 34. a5xc7 d4-b2 0-1

The Seventh Computer Olympiad

The seventh Computer Olympiad was held in Maastricht in 2002. Four programs competed in the LOA tournament: YL, MIA III, (T-T), and PETE (see Björnsson and Winands, 2002 for a detailed report). YL was substantially improved with respect to the previous year's version, featuring more extensive LOA domain knowledge and a better-tuned evaluation function. MIA III is described in this thesis. (T-T) – pronounced as "uruuru" – by Jun Nagashima (Japan), and PETE by Inge Wallin (Sweden) were newcomers. (T-T) used an innovative selective-extension algorithm, called realisation-probability search (first introduced in the Shogi program GEKISASHI that won the 12th CSA tournament in 2002). PETE's author is one of the main contributors to the GNU GO project. The final standings of the LOA tournament are shown in Table A.3.

YL won its third consecutive gold medal, MIA earned the silver, and (T-T) the bronze. The fact that PETE lost all its games did not reflect rightly on the program's

Table A.3: The final standings of the LOA tournament at the 7th Computer Olympiad.

Place	Program	YL	MIA III	(T-T)	PETE	Points
1	YL	-	2.5-1.5	4-0	4-0	10.5
2	MIA III	1.5-2.5	-	3-1	4-0	8.5
3	(T-T)	0-4	1-3	-	4-0	5
4	PETE	0-4	0-4	0-4	-	0

playing strength. It played the opening and middle game very respectably, whereas the endgame was its weak spot – typically throwing away any advantage it might have had. Both YL and MIA were again noticeably stronger than last year, a direct consequence of the competitive nature of these Computer Olympiad events.

Selected Games

MIA III vs. YL (Game 3)

1. e1-g3 a3-d3 2. d8-b6 a4-c2 3. f1-f3 a2-d2 4. d1-g4 h5-f7 5. c8-f5 h7-e7 6. b1-b4 h4-f6 7. b8-b5 c2xf5 8. b4xe7 f6xb6 9. g1-g5 f5xb5 10. c1-b1 d3-d5 11. f3xd5 h2-f2 12. b1-b4 d2xg5 13. e7xg5 h6xf8 14. b4-d4 f8-e7 15. e8-c6 f2-f4 16. c6-e4 b5-c6 17. g5xe7 b6-e6 18. e7-f6 h3-g2 19. g8-h8 e6xe4 20. h8-e5 c6xf6 21. g3-g6 f7-d7 22. g6xe4 a5xd5 23. g4-f3 1-0

YL vs. MIA III (Game 4)

1. b1-b3 h4-f2 2. b8-b6 a2-d2 3. b6xf2 This is a somewhat unusual opening move. Typically, Black continues to reinforce his blockade along the b-file, instead of attacking White's blockade. However, the move played is an interesting choice. **3. ... h5-f7 4. e1-g3 h7-e7 5. d8-b6 d2-d4 6. d1-e2 a6-d3 7. g8-g5 a5-b4 8. f8xh6 e7xg5 9. h6xh3** The white piece on **h2** is now completely blocked by the black pieces (see Figure A.1a). Normally, this would be a clear advantage for Black, although in this case it is not so clear because there are decentralised black pieces. Sooner or later Black must release the blockade to activate his pieces. Both programs evaluate this position as slightly favourable for Black. Maybe, a better plan for White would have been to play 8. ... h2-h5 (or h3xh6) instead of grabbing a piece with 8. ... e7xg5. **9. ... a7-c7 10. h3-f5 a4-a2 11. e2-e4 b4xe4 12. c8-e6 a2-a4 13. f1-g2 e4xg2 14. b6-b4 c7-b6 15. c1-c2 g2-f3 16. g1-g4 a3-a5 17. e8-c6 a5-d5** Black is now clearly getting the upper hand (see Figure A.1b). Although White has a firm grip on the centre, Black circumvents White's stronghold by wrapping around the centre. White's pieces are more scattered and the piece on **h2** is still somewhat poorly placed. Black now finishes the game convincingly. **18. c6-f6 d4xf6 19. g3-e5 b6-c7 20. f2-g3 h2-f2 21. g3-d6 c7-c5 22. b3-b5 f7-e8 23. c2xa4 f3-d1 24. g4-d4 c5-c4 25. a4-d7 d1-e1 26. d4-b6 e1xe5 27. d7-c7** 1-0

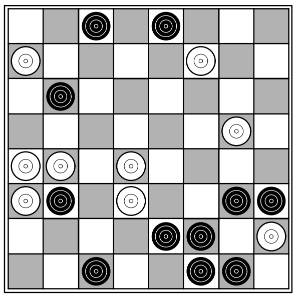
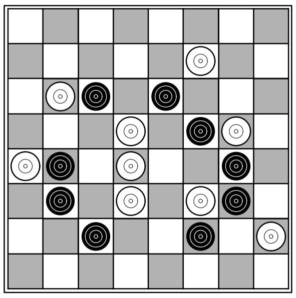

Figure A.1: (a) After **9. h6xh3**. (b) After **17. ... a5-d5**.

The Eighth Computer Olympiad

The eighth Computer Olympiad was held in Graz in 2003. Three programs competed in the LOA tournament: MIA IV, BING, and (T-T) (see Winands, 2003 for a detailed report). MIA IV is described in this thesis. According to its author, (T-T)'s evaluation function was considerably improved compared to the previous year's version. BING (Bing Is Not GnuChess) written by Bernard Helmstetter (France) was a newcomer. It reused a great deal of the code of GNU CHESS (mainly written by Chua Kong-Sian and Stuart Cracraft). The program BING used a new approach to LOA: a neural network with one hidden layer in the evaluation function. This neural network used as input the standard LOA evaluation-function features (i.e., distance to the centre-of-mass, mobility, quads, number of connections, etc.).

Each program as usual played four games against each other. BING and MIA played the first two games. MIA convincingly won the first game after a crucial strategic mistake by BING in the early phase of the game. Still, it did not pass unnoticed that BING played the first moves according to the main line of MIA's opening book. Moreover, I was a bit surprised by BING's good strategic play. The usual experience with new programs is that they do not play so well in the opening. Nevertheless, I was confident for the second game in which MIA played White. BING surprisingly opened the second game with the somewhat weak **1. c1-c3** (1. b1-b3 or 1. d1-b3 are considered better opening moves). After a few moves (for the moves of the game, see below), MIA was able to turn the first-player's advantage into its own and gradually improved its position on the board. However, after **13. b1-c2** (see Figure A.2a) its score started slowly to drop. Finally, MIA played **13. ... d7-c6.** Post-mortem analysis suggested that 13. ... e5xc3 was maybe better (but not that much). After the move played the position deteriorated and MIA lost. MIA apparently walked into a tactical trap and in contrast to the first game BING did not make any real mistakes.

On the second day (T-T) entered the arena. I expected that BING would lose points against the greatly improved (T-T). However, BING won the first two games. Then MIA won without many problems its first game against (T-T). In the next game (T-T) showed its strength. Again MIA ran into problems with White, and

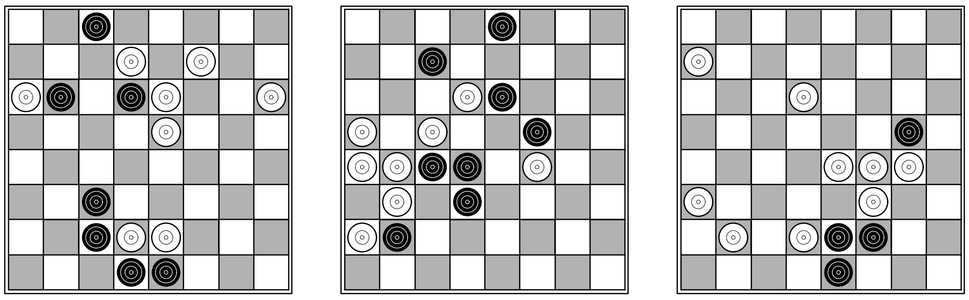

Figure A.2: (a) After **13. b1-c2**. (b) After **17. ... g5-f4**. (c) After **18. ... d4-g4**.

once more after it had reached a “favourable” position. However, that position was doomed because of a very deep and subtle tactical feature. (T-T) got the advantage back, but erred with **18. c7xa5** (Figure A.2b, possible alternatives are 18. c7-d7 or 18. d3xb3). Somewhat later MIA succeeded to escape. (T-T) finished its already unfortunate day with another loss against BING. After this day BING had 4 points, MIA 3 points and (T-T) 0 points.

The third day of the tournament turned out to be *decision day*. BING started well by winning again against (T-T), obtaining a good position to win the tournament. In the third encounter between MIA and BING, MIA played the black pieces, got a slight advantage out of the opening, maintained it, and gradually outplayed the opponent. In the last game against BING, MIA was again able to reach a decent position with White. Then BING sacrificed a few pieces to reach a slippery position. However, BING crashed after **18. ... d4-g4** (see Figure A.2c). According to MIA the game is probably a draw by repetition (19. g5-h4 g4-h3 20. h4-g5 h3-g4). BING was not going for the draw, it planned a weak move instead. Under the circumstances described above (a crashed program) Helmstetter resigned on behalf of the program, because it was short on time anyway. MIA won easily the last games against (T-T). The final standings of the LOA tournament are shown in Table A.4.

Table A.4: The final standings of the LOA tournament at the 8th Computer Olympiad.

Place	Program	MIA IV	BING	(T-T)	Points
1	MIA IV	-	3-1	4-0	7
2	BING	1-3	-	4-0	5
3	(T-T)	0-4	0-4	-	0

MIA won its first gold medal, BING earned the silver medal and (T-T) received the bronze medal. BING was definitely the surprise of the tournament.

Selected Games

MIA IV vs. Bing (Game 1)
1. e8-g6 a5-c7 2. d1-b3 a7-d7 3. b8-b5 a6-d6 4. c1-c4 a2-c2 5. b1-b4 a4-a6 6. b5xd7 a6-e6 7. g1-g4 h6xf8 8. g8-g5 d6xg6 9. b3xe6 a3-c3 10. e6xg6 c2-f5 11. e1-g3 c7-f4 12. b4-b5 f5xf1 13. c4-d5 c3-d2 14. c8xf8 d2xg5 15. d8xg5 f1-e1 16. b5-f5 e1-d1 17. g3-g7 f4-c4 18. f8-f6 d1xg4 19. g6-e6 1-0

Bing vs. MIA IV (Game 2)
1. c1-c3 a4-c2 2. c8-c5 h5-f7 3. b8-b6 a5xc5 4. g1xc5 a7-d7 5. f1xh3 a2-d2 6. e8-e6 h2-e2 7. c5xc2 a3-c5 8. c2xc5 h7xh3 9. g8-g7 h3xe6 10. g7-e5 h4-f2 11. f8-d6 f2xc5 12. d8-c8 c5xe5 13. b1-c2 d7-c6 14. c8xe6 c6xc3 15. b6-b5 d2-d5 16. e1xc3 e2xb5 17. c2-c4 d5-d2 18. d1-c2 e5xc3 19. d6-d4 f7-g7 20. e6-e5 d2xd4 21. c2-d3 g7-g6 22. e5-e4 1-0

MIA IV vs. Bing (Game 3)
1. b8-b6 h5-f7 2. e1-g3 h4-f4 3. c1xf4 h6xf4 4. b1-b3 h7-e7 5. g1-g4 h2-f2 6. d1xa4 a6-c4 7. f1-d3 e7-b4 8. a4-c6 h3-g2 9. g3-c3 f2-c5 10. d8xa5 b4xf8 11. a5-c7 a2-a5 12. c3-d4 a3-b2 13. d4-d6 a5-b4 14. e8-e7 g2-e4 15. d3-d5 b2-c3 16. g8-g6 e4-c2 17. b6-f6 a7-b8 18. g4-e6 b8-b5 19. c8-e8 c5-d4 20. e8xb5 c3xf6 21. c7xf7 f4-e3 22. b3-d3 f8-e8 23. c6-e4 c4-c6 24. g6xc2 b4xe4 25. c2-c4 1-0

Bing vs. MIA IV (Game 4)
1. d8-b6 h5-f7 2. b1-b4 h2-f2 3. b6xf2 h6xf8 4. c8xa6 h7-e7 5. b8-b6 f8xb4 6. e8-c6 h3xf1 7. d1-e2 a5-d2 8. a6-d3 a2xe2 9. g8-g6 e2-b2 10. g1-g3 f1xd3 11. g3xd3 h4-f6 12. g6-e4 f6-d4 13. b6-d6 e7xe4 14. c1-c3 a4xc6 15. d6-f6 b4-d6 16. c3-f3 c6xf3 17. f6-g5 f7-f4 18. d3-e2 d4-g4 Bing resigns 0-1

(T-T) vs. MIA IV (Game 2)
1. d1-b3 h4-f2 2. g1-g3 h7-f7 3. g8-g6 h2-e2 4. b1-d3 h6-f4 5. b8-b6 h3-f5 6. e1-b1 f4-d4 7. b1xf5 f2xb6 8. g3-f4 f7-d7 9. c1xa3 h5-h6 10. f8-d6 e2-e4 11. f4-f7 e4-b4 12. c8-c7 d7xf5 13. f1-c4 h6-g5 14. f7xf5 b6xb3 15. d8xd4 a6xd6 16. g6-e6 a7-c5 17. a3-b2 g5-f4 18. c7xa5 d6-a3 19. f5xc5 a3xd3 20. e6-d6 f4-f3 21. d6xd3 f3-e2 22. d4-d2 b3xd3 23. e8-e6 d3-c3 24. e6-b3 e2xc4 25. a5-b6 a4-d4 26. d2xd4 a2-a3 0-1

The Ninth Computer Olympiad

The ninth Computer Olympiad was held in Ramat-Gan, in 2004. Four programs competed in the LOA tournament: MIA 4++, Bing, YL, and Lola. MIA 4++ was a modified and enhanced version of MIA IV. MIA's author felt that the improvements were not a sufficient reason to justify an increase of the version number. According to their authors, Bing and YL did not change since their last tournament. Lola written by Rémi Coulom (France) was a newcomer.

The first day YL played its first two games against Lola and won them convincingly. MIA had to face a more difficult opponent: Bing. MIA, playing the black pieces, had a great start by defeating Bing in 17 moves. Afterwards, this was the fastest victory in the tournament. In the second game Bing had the advantage by playing the black pieces. It played the same opening, from which Bing defeated MIA last year. This time MIA handled the opening better and won the game. The last game of the day was YL against Bing. YL was the clear favourite to win this game, but an upset occurred. Direct from the start YL went into panic mode (assigning extra time to analyse the position). As a spectator I was surprised by the number of times YL went into panic mode during the game. This never happened in the past games YL played against MIA. YL lost the game and Bing was back in the tournament. Björnsson blamed the panic mode for the result and turned it off in the remaining games.

The second day turned out to be a remarkable day in the LOA tournament. Bing started with another victory against YL. Next, the old rivals MIA and YL met. YL had to win to keep a good chance of winning the tournament. The first game MIA maintained its first-player advantage and gradually outplayed YL. The second game was more interesting. The moves are given at the end of this section. It was one of the most balanced and "drawish" games of the tournament. After **8. ... a3xc1**, MIA equalled the position for White by trapping the black piece at **b1** (see Figure A.3a). During the game, MIA proposed two times a draw, which were declined by YL. The first one was a draw by repetition, the second one a draw by simultaneous connections. When YL detected the draw, MIA did not agree. Finally, the programs drew with the following sequence (see Figure A.3b): **21. d6-c5 f5-d7 22. c5-d6 d7-f5 23. d6-c5 f5-d7 24. c5-d6 d7-f5**. After this game MIA had good chances to keep its title, whereas YL's chances dwindled. YL now had to fight for the silver medal. Meanwhile Bing defeated Lola two times. The day ended with another strong game of Bing against YL (for the moves see below). After **6. ... h6xf8** (see Figure A.3c), YL replied with the strange **7. f3xf7**, which MIA and Bing regarded as a mistake (7. g1-g4 or 7. d8-f6 were better moves). YL lost the advantage, never fully recovered, and finally lost. Like last year, Bing had suddenly again a good chance to win the tournament.

The third day started with a confrontation between MIA and Bing. If Bing would win the two games against MIA, it would have a golden opportunity to win the tournament. However, Bing did not have the tools to select an alternative opening. Bing's author trusted that its random factor in the root would cause a surprise. Although the games were not the same, Bing lost in the same style as it had lost the previous games. In the same round YL played Lola. First, it seemed that YL would win the two games. But then the game Lola-YL became a draw by simultaneous connection due to a mistake of YL. The next round Bing was able to defeat YL for the fourth time in a row. This accomplishment has not been seen earlier in a LOA tournament against YL. Meanwhile MIA faced for the first time Lola and defeated it. In the next round YL got the opportunity to show its strength against MIA. Björnsson tried to trick MIA with an unconventional opening

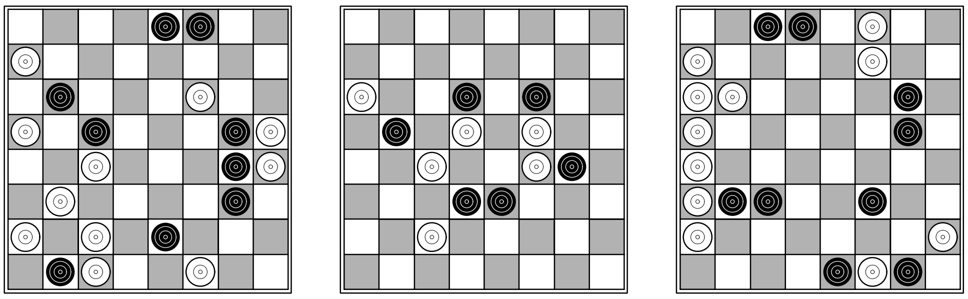

Figure A.3: (a) After **8. ... a3xc1**. (b) After **20. ... g5-d5**. (c) After **6. ... h6xf8**.

but the plan backfired and YL lost. In the next game MIA had the white pieces and was able to surprise YL in the opening. MIA won the game. With three games still to play MIA was already certain of the gold medal. In these rounds Bing defeated Lola again. Finally, on the fourth day MIA won the remaining three games against Lola without any problem. The final standings of the LOA tournament are shown in Table A.5. MIA 4++ took the gold medal, Bing was awarded silver, and YL received the bronze medal.

Table A.5: The final standings of the LOA tournament at the 9th Computer Olympiad.

Place	Program	MIA 4++	Bing	YL	Lola	Points
1	MIA 4++	-	4-0	3.5-0.5	4-0	11.5
2	Bing	0-4	-	4-0	4-0	8
3	YL	0.5-3.5	0-4	-	3.5-0.5	4
4	Lola	0-4	0-4	0.5-3.5	-	0.5

Obviously, YL's supremacy (reigning the LOA community from 2000 to 2003) has ended. MIA 4++ won the 9th Computer Olympiad LOA tournament without losing a single game. YL had to face a 7.5-point gap with MIA 4++. However, the biggest surprise was that Bing, too, defeated convincingly YL four times and finished in front of it. Lola may be stronger than the result shows. There were no quick victories by other programs against Lola, which is a sign of reasonable strength.

Selected Games

MIA4++ vs. Bing (Game 1)
1. e1-g3 a4-c2 2. d8-b6 h7-e4 3. g8-g5 a5-d5 4. b8-b5 a6xc8 5. b1-b4 a3xc1 6. d1-d3 c8-e6 7. b4xe4 a2-d2 8. g1-g4 h6-h1 9. f1-f3 h3-g2 10. g4xe6 h5xf3 11. g5xd5 d2-f4 12. b6-b4 h4-g5 13. f8-f5 h1-h3 14. g3-f2 h3-e3 15. e8-c6 f3-e2 16. b5xe2 a7-d4 17. b4-c5 1-0

Bing vs. MIA4++ (Game 2)
1. c1-c3 a4-c2 2. c8-c5 h5-f7 3. b8-b6 a5xc5 4. g1xc5 a7-d7 5. e1-e3 a2-d2 6. g8-g7 h6xe3 7. b6xe3 a3-a5 8. c5xc2 h3xe3 9. c3xe3 a5-c3 10. e3xc3 a6-c4 11. d8-f6 h4xf6 12. c3xf6 c4xc2 13. b1-b2 h2-g3 14. d1-e2 c2-e4 15. e2-c4 h7-h6 16. f1-f5 h6xf6 17. c4-e6 d2-d4 18. b2-b1 e4-d5 19. f8-e7 g3-h4 20. e8-d8 d4-e3 21. f5-h5 e3xe6 22. h5xf7 h4-h5 23. g7-f8 h5-f5 0-1

MIA4++ vs. YL (Game 1)
1. e8-g6 a5-c7 2. d1-b3 a7-d7 3. b1-b4 h7-e7 4. f1xh3 h2-e5 5. g8-g5 h4-e4 6. b4xe4 a3-d6 7. e1xe5 a2-a5 8. e4xe7 h5-g4 9. b3-b5 a6-e6 10. c1-c4 c7xc4 11. e7-g7 h6-h4 12. f8xd6 a5-b4 13. g7-e7 b4-c3 14. h3-f3 g4xc8 15. g1-g4 a4-a3 16. e7-g7 c3-b4 17. g7-f8 d7-c7 18. f3-f5 c4xg4 19. b5-c6 g4-f3 20. b8-b6 c8xf5 21. f8xf5 f3xf5 22. b6-c5 b4-c3 23. g5-g7 f5-f6 24. g7-e7 c7xe7 25. d8-c7 h4-g3 26. g6-f5 1-0

YL vs. MIA4++ (Game 2)
1. d1-b3 h7-f7 2. e1-g3 f7xb3 3. g1-g4 h3xf1 4. g8-g5 h6-f6 5. d8-b6 a4-c2 6. c8-c5 a6-c4 7. b8-e5 h2-e2 8. e5xe2 a3xc1 9. f8-d6 f6-f4 10. b6-e3 c1xc5 11. b1xb3 a2-d2 12. e8-f8 c5xg5 13. g3-d3 h5-e5 14. e2xe5 h4-f6 15. d6xf6 f1-f5 16. b3-b4 a5xe5 17. e3xe5 d2-f2 18. f8-d6 a7-a6 19. b4-b5 f2-e3 20. e5xe3 g5-d5 21. d6-c5 f5-d7 22. c5-d6 d7-f5 23. d6-c5 f5-d7 24. c5-d6 d7-f5 drawn by repetition 0.5-0.5

MIA4++ vs. YL (Game 3)
1. e8-g6 a6-d6 2. d8-b6 a7-c7 3. f8xh6 h7-f7 4. b8-b5 h5-e5 5. g8-e8 c7-e7 6. g1-g3 a5-c3 7. h6xd6 a3xd6 8. g6xd6 h3-f5 9. e1xc3 e5xc3 10. c8-e6 c3-d4 11. b1-b4 a2-d5 12. f1-d3 h2-g2 13. c1-c2 f5-c5 14. c2-e4 f7-h7 15. d3-b3 h7-f7 16. g3-e5 g2-h2 17. b6-c7 c5xc7 18. b3-b6 h4-h6 19. b6xd4 c7-d8 20. d4-f6 f7-d7 21. e8-g8 d7-f7 22. d1xd5 h6-g7 23. b4xe7 d8-f8 24. b5-b4 f8-d8 25. e4-c6 a4xc6 26. g8-e8 h2xe5 27. b4-c4 1-0

YL vs. MIA4++ (Game 4)
1. d1-b3 h4-f2 2. b8-b5 h7-f7 3. f1-c4 h5xe8 4. b1-b4 a3xc1 5. g1-g3 c1xc4 6. c8-c6 h2-e2 7. e1-d1 c4xc6 8. g8-g6 a6-e6 9. f8-d6 f7xb3 10. d8-d5 h3-f5 11. b5xf5 a5xd5 12. f5xd5 f2-d4 13. d6-f8 b3-d3 14. d1-b3 e2-e5 15. g3xd3 h6-g5 16. d3-f5 g5-f4 17. f8-h8 c6-f6 18. h8-f8 f4xb4 19. f8-d6 a7-c5 20. d5-e4 c5-c4 21. e4-b1 a4-e4 22. d6xd4 e8-d7 23. b1xb4 a2-b2 24. g6-h5 d7-d5 25. f5-f3 f6-d6 26. h5xe5 d6xb4 27. b3-d3 b4-c3 0-1

YL vs. Bing (Game 3)
1. b1-b3 h4-f2 2. b8-b6 f2xb6 3. e8-g6 h3xf1 4. c1-c3 h7-f7 5. g8-g5 h5-f3 6. d1xf3 h6xf8 7. f3xf7 f8-6 8. d8xd6 h2-f4 9. e1-b4 f4xd6 10. f7-f5 a7-b8 11. b3-e3 b8-b5 12. f5xb5 a2-b3 13. e3-e2 f1-f2 14. c3-c5 f2-e1 15. c5-c3 a3-b2 16. g1-e3 d6xb4 17. g5-d5 b3xd5 18. g6-d6 e1-d1 19. e2-c2 a6-a3 20. e3-d2 d5-c6 21. c3xa3 c6-c3 22. a3xc3 d1-c1 23. d6xb4 c1-a3 0-1

Appendix B

Pseudo Code

In this appendix the pseudo code of PN, PN^2, and PDS is given.

B.1 Pseudo Code for PN Search

Below we give the pseudo code for PN search, which was discussed in Subsection 4.2.1. For ease of comparison we use similar pseudo code as given in Breuker (1998). `PN(root, maxnodes)` is the main procedure of the algorithm. The procedure `evaluate(node)` evaluates a position, and assigns one of the following three values to a node: `FALSE`, `TRUE`, and `UNKNOWN`. The proof and disproof numbers of a node are initialised by `setProofAndDisproofNumbers(node)`. The function `selectMostProvingNode(node)` finds the most-proving node. Expanding the most-proving node is done by `expandNode(node)`. After the expansion of the most-proving node, the new information is backed up by `updateAncestors(node, root)`. The function `countNodes()` gives the number of nodes currently stored in memory.

```
//The PN-search algorithm
PN(root, maxnodes){

  evaluate(root);
  setProofAndDisproofNumbers(root);

  while(root.proof != 0 && root.disproof != 0 && countNodes() <= maxnodes){
    //Second Part of the algorithm
    mostProvingNode = selectMostProvingNode(currentNode);
    expandNode(mostProvingNode);
    currentNode = updateAncestors(mostProvingNode, root);
  }
}

//Calculating proof and disproof numbers
setProofAndDisproofNumbers(node){
   if(node.expanded) //Internal node;
     if(node.type == AND_NODE){
       node.proof = 0;
       node.disproof = INFINITY;
       for(each child n){
         node.proof = node.proof + n.proof;
         if(n.disproof < node.disproof)
           node.disproof = n.disproof;
       }
     }
     else{ //OR node
       node.proof = ProofNode.INFINITY;
       node.disproof = 0;
       for(each child n){
         node.disproof = node.disproof + n.disproof;
         if(n.proof < node.proof)
           node.proof = n.proof;
       }
     }
   else //Leaf
     switch(node.value){
       case FALSE:
            node.proof = INFINITY;
            node.disproof = 0;
       case TRUE:
            node.proof = 0;
            node.disproof = INFINITY;
       case UNKNOWN:
            node.proof = 1;
            node.disproof = 1;
     }
}

//Select the most-proving node
SelectMostProvingNode(node){
  while(node.expanded){
    n = node.children;

    if(node.type == OR_NODE) //OR Node
      while(n.proof != node.proof)
        n = n.sibling;
    else //AND Node
      while(n.disproof != node.disproof)
        n = n.sibling;

     node = n;
  }
  return node;
}

//Expand node
expandNode(node){
   generateAllChildren(node);
   for(each child n){
      evaluate(n);
      setProofAndDisproofNumbers(n);
      //Addition to original code
      if((node.type == OR_NODE && n.proof == 0) ||
         (node.type == AND_NODE && n.disproof == 0))
          break;
   }
   node.expanded = true;
}

//Update ancestors
updateAncestors(node, root){

   do{
      oldProof = node.proof;
      oldDisProof = node.disproof;

      setProofAndDisproofNumbers(node);
      //No change on the path
      if(node.proof == oldProof && node.disproof == oldDisProof)
        return node;
      //Delete (dis)proved trees
      if(node.proof == 0 || node.disproof == 0)
        node.deleteSubtree();

      if(node == root)
        return node;

      node = node.parent;
   }while(true)
}
```

B.2 Pseudo Code for PN^2 Search

The pseudo code for PN^2 search, which we have discussed in Subsection 4.2.2, is almost the same as that for PN. We only modified the `ExpandNode(node)` procedure. The modified procedure is given below. The function `computeMaxNodes()` computes the number of nodes which may be stored for the PN search, according to equation 5.4.

```
//Replace code
expandNode(node){
  //Call PN search, described in the previous section
  PN(node, computeMaxNodes());
  //Delete the subtrees of the children
  for(each child n){
    n.deleteSubtree();
    n.expanded = false;
  }
}
```

B.3 Pseudo Code for PDS

In this section the pseudo code for PDS, which we have discussed in Subsection 5.1.2, is given. We use similar pseudo code as given in Nagai (1998) for the PDS algorithm. The proof number in an OR node and the disproof number in an AND node are equivalent. Analogously, the disproof number in an OR node and the proof number in an AND node are equivalent. As they are dual to each other, an algorithm similar to negamax in the context of minimax searching can be constructed. This algorithm is called NegaPDS. In the following, procedure `MID(n)` performs multiple iterative deepening. The function `proofSum(n)` computes the sum of the proof numbers of all the children. The function `disproofMin(n)` computes the minimum of the disproof numbers of all the children. The procedures `putInTT()` and `lookUpTT()` store and retrieve information to and from the transposition table. `isTerminal(n)` checks whether a node is a win, a loss, or a draw. The procedure `generateChildren(n)` generates the children of a node. By default, the proof number and disproof number of a node are set to unity. The procedure `findChildrenInTT(n)` checks whether the children are already stored in the transposition table. If a hit occurs for a child, its proof number and disproof number are set to the values found in the transposition table.

```
//Iterative deepening at root
procedure NegaPDS(root){

  root.proof = 1;
  root.disproof = 1;

  while(true){
    MID(root);
    //Terminate when the root is proved or disproved
    if(root.proof == 0 || root.disproof == 0)
      break;
    if(root.proof <= root.disproof)
      root.proof++;
    else
      root.disproof++;
  }
}

//Explore node n
procedure MID(n){

  //Look up in the transposition table
  lookUpTT(n, &proof, &disproof);
  if(proof == 0 || disproof == 0
  || (proof >= n.proof && disproof >= n.disproof)){
    n.proof = proof; n.disproof = disproof;
    return;
  }

  //Terminal node
  if(isTerminal(n)){
    if((n.value == true && n.type == AND_NODE) ||
    (n.value == false && n.type == OR_NODE)){
      n.proof = INFINITY; n.disproof = 0;
    }
    else{
      n.proof = 0; n.disproof = INFINITY;
    }
    putInTT(n);
    return;
  }

  generateChildren();
  //Avoid cycles
  putInTT(n);

  //Multiple-iterative deepening
  while(true){
    //Check whether the children are already stored in the TT.
    //If a hit occurs for a child, its proof number and disproof number
    //are set to the values found in the TT.
    findChildrenInTT(n);

    //Terminate searching when both proof and disproof number
    //exceed their thresholds.
    if(proofSum(n) == 0 || disproofMin(n) == 0 || (n.proof <=
    disproofMin(n) && n.disproof <= proofSum(n))){
      n.proof = disproofMin(n);
      n.disproof = proofSum(n);
      putInTT(n);
      return;
    }

    proof = max(proof,disproofMin(n));
    n_child = selectChild(n,proof);

    if(n.disproof > proofSum(n) && (proof_child <= disproof_child
      || n.proof <= disproofMin(n)))
      n_child.proof++;
    else
      n_child.disproof++;

      MID(n_child);
  }
}

//Select among children
selectChild(n, proof){
  min_proof = INFINITY;
  min_disproof = INFINITY;
  for(each child n_child){
    disproof_child = n_child.disproof;
    if(disproof_child != 0)
      disproof_child = max(disproof_child, proof);
    //Select the child with the lowest disproof_child (if there are
    //plural children among them select the child with the lowest
    //n_child.proof)
    if(disproof_child < min_disproof || (disproof_child == min_disproof
    && n_child.proof < min_proof)){
      n_best = n_child;
      min_proof = n_child.proof;
      min_disproof = disproof_child;
    }
  }
  return n_best;
}
```

Index

Summary

This thesis investigates how search can be guided by knowledge in such a way that the search space is traversed efficiently and effectively. For this task we focus on the question how to combine relevant knowledge with intelligent search. The more adequate the knowledge, the better the search.

Chapter 1 provides a brief introduction on games and Artificial Intelligence (AI). It then discusses the notion of *informed search*. It is well-known that regular search at least needs *terminal knowledge* to solve a problem. If the search process is using *directing knowledge* too, it is called informed search. The following problem statement guides our research.

> **Problem statement**: *How can we develop informed-search methods in such a way that programs significantly improve their performance in a given domain?*

To answer the problem statement we formulated four research questions on intricate topics of informed search. They deal with (1) the evaluation function, (2) competitive proof-number search algorithms, (3) forward pruning, and (4) move ordering.

Chapter 2 describes the test environment used to answer the problem statement and the four research questions. Conditions for a suitable test environment are formulated with emphasis on the notions *game* and *game program*. The game under consideration is the game of Lines of Action (LOA). The chapter provides some background information, the rules of the game, a variety of game characteristics, seven basic principles, and a review of the role of LOA in the AI domain. The search engine of the LOA tournament program MIA (Maastricht in Action) is described. It is used as test vehicle for all experiments in this thesis.

Informed search cannot exist without a decent evaluation function. For LOA, it is a challenge to build such an evaluation function, since it should incorporate the basic principles of the game and simultaneously increase the profitability. This challenge has led us to the first research question.

> **Research question 1**: *How can we build a strong evaluation function for Lines of Action?*

Chapter 3 answers the first research question by investigating which features are important for a LOA evaluation function. The features are based on the seven basic principles described in Chapter 2. It turns out that the following nine features are important: concentration, centralisation, centre-of-mass position, quads, mobility, walls, connectedness, uniformity, and player to move. The features resulted in the evaluator MIA IV, which is tested in a round-robin tournament against its predecessors MIA I, MIA II, and MIA III. The latter ones have performed well at previous Computer Olympiads (2000, 2001, 2002). Experiments show that the current MIA IV defeats them all with large margins. At all search depths investigated MIA IV wins at least 75 per cent of the games. We remark that many features in the evaluation function do not consume much time. By using precomputed tables and caching, they can be evaluated quite straightforwardly. The most important feature is concentration, followed by the mobility feature. All features are essential and contribute to the playing strength. There exist much interconnectedness and overlap between the features, which influence their performance. Therefore, all features have to be simultaneously fine-tuned (or heuristically optimised) in a careful way. The combination of the nine features mentioned has resulted in an evaluation function that significantly increased the playing strength of our LOA program compared to programs with less-sophisticated evaluation functions. With the present evaluator we gained first places at the 8^{th} and 9^{th} Computer Olympiad (2003, 2004).

Chapter 4 investigates several Proof-Number (PN) search algorithms. The description helps to formulate our second research question. The chapter starts by providing a short description of the original PN-search method, and two main successors of the original PN search, i.e., PN^2 search and depth-first variants of PN search such as *Proof-number and Disproof-number Search* (PDS). The original PN-search method is formulated as a best-first search algorithm. It has the drawback that the whole search tree has to be stored in memory. The search can end prematurely because of memory exhaustion. Recently, some PN variants have been constructed as depth-first search algorithms; yet they still behave as their corresponding best-first search algorithms. The advantage is that there is no longer a need to store the whole tree in memory. The disadvantage is that the PN variants have to re-generate the tree in each iteration. The PN-search algorithms can be applied in two different ways: offline and online. First, we concentrate on the *offline* application of the PN-search algorithms. The number of positions they can solve (i.e., the post-mortem analysis quality) is tested on a set of endgame positions. Besides establishing the number of solutions, we investigate to what extent the algorithms are restricted by their working memory *or* by the speed of the searching process. We observe that mobility and deleting (dis)proved pn_2 subtrees speed up PN and PN^2, and increase their ability of solving endgame positions. A comparison of the performance between PN, PN^2, PDS, and $\alpha\beta$ is given. It is shown that PN-search algorithms clearly outperform $\alpha\beta$ in solving endgame positions in LOA. However, the memory problems make the plain PN search a weaker solver for the harder problems. PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$. But PN^2 is restricted by its working memory, and PDS is three times slower than PN^2. Second, we briefly investigate the *online* application of PN search.

In particular, the real-time application of PN search during a game is examined. Finally, we conclude that our method (called PN-$\alpha\beta$), which combines PN search and $\alpha\beta$, outperforms plain $\alpha\beta$ search as implemented in the tournament program MIA.

When reducing the need for memory at the cost of additional searching, we arrive at a crucial memory characteristic of knowledge in an informed-search process. The memory problem of PN^2 and the speed problem of PDS shown in the previous chapter have guided us to the second research question.

> **Research question 2**: *How can we develop a proof-number search algorithm, which is competitive in speed and not restricted in working memory?*

Chapter 5 answers the second research question and presents a new proof-number search algorithm, called PDS-PN. It is a two-level search (like PN^2), which performs at the first level a depth-first PDS, and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS. Results of experiments with PDS-PN on a set of endgame positions are given. The experiments show that within an acceptable time frame PDS-PN is more effective for really hard endgame positions than $\alpha\beta$ and any other PN-search algorithm.

As stated before, we define *informed search* as search which applies a regular search process and uses directing knowledge. Therefore, we would like to investigate whether it is beneficial to improve forward-pruning methods, such as null move and multi-cut, in the Principal-Variation-Search (PVS) framework. This idea has guided us to the third research question.

> **Research question 3**: *How can we improve forward-pruning methods in the Principal-Variation-Search framework?*

Chapter 6 answers the third research question. Forward-pruning methods, such as multi-cut and null move, are tested at so-called ALL nodes. PVS is improved by four small but essential additions. The new PVS algorithm guarantees that forward pruning is safe at ALL nodes. Experiments show that multi-cut at ALL nodes (MC-A) when combined with other forward-pruning mechanisms gives a significant reduction of the number of nodes searched. Multi-cut at expected ALL nodes gives a safe reduction of approximately 40 per cent of the number of nodes searched at depth 14 in combination with null move and the regular multi-cut at CUT nodes (MC-C). Experiments suggest that parameters more aggressively chosen than MC-C lead to an additional improvement. In comparison, a (more) aggressive version of the null move (variable null-move bound) gives less reduction at expected ALL nodes than our algorithm. We observe that MC-A still searches 22 per cent fewer nodes than variable null-move bound at expected ALL nodes. Moreover, MC-A was able to increase significantly the playing strength of the program MIA. From these observations we may conclude that MC-A is a valuable enhancement of PVS.

Move ordering is an instance of directing knowledge in an informed-search process. In particular in $\alpha\beta$ search, move ordering is one of the main techniques to reduce the size of the search tree. Of these techniques dynamic move ordering is most successful. It is characterised by its dependence on information gained during the search. This has led us to the fourth research question.

> **Research question 4**: *How can we use information gained during the search to improve move ordering?*

Chapter 7 answers the fourth research question by describing a new method for move ordering, called the relative history heuristic. It is a combination of the history heuristic and the butterfly heuristic. Instead of only recording moves which are the best move in a node, we also record the moves which are applied in the search tree. Both scores are taken into account in the relative history heuristic. The relative history heuristic favours moves which are the good moves on average instead of moves which are the best move in absolute terms. When replacing the history heuristic by the relative history heuristic, experiments show that this method gives a reduction between 10 and 15 per cent of the number of nodes searched. The results were confirmed by the Go program MIGOS. Hence, we may conclude that the relative history heuristic is a valuable dynamic move-ordering technique in a game tree of considerable depth (more than 12 plies). Finally, we remark that the utility of increments other than 1 does not show a much better performance in the (relative) history heuristic for our LOA program MIA. The good performance of the increment of 1 could be the result of some domain-dependent properties. The relative history heuristic seems to be a valuable element in move ordering.

The last chapter of the thesis returns to the four research questions and the problem statement as formulated in Chapter 1. Taking the answers to the research questions above into account we see that there are several successful ways to improve the performance of informed-search methods. Our improvements of the evaluation function, proof-number search, forward pruning, and move ordering have given significant results in our LOA test domain. Yet, we were able to provide additional promising directions for future research. Finally, the question whether it is possible to solve the game of LOA is still open.

Samenvatting

Dit proefschrift onderzoekt op welke manier een zoekproces door aanwezige kennis zodanig gestuurd kan worden dat de zoekruimte efficiënt en effectief doorlopen wordt. Voor het beantwoorden van deze onderzoekstaak richten we ons op de vraag hoe we relevante kennis met intelligent zoeken kunnen combineren. Hoe handzamer de kennis is, des te beter gaat het zoeken.

Hoofdstuk 1 geeft een korte inleiding op het gebied van spelen en Artificiële Intelligentie (AI). Het introduceert het begrip van *geïnformeerd zoeken*. Het is algemeen bekend dat regulier zoeken tenminste *terminale kennis* nodig heeft om een probleem op te lossen. Als het zoekproces daarenboven gebruik maakt van *sturende kennis*, dan spreekt men van geïnformeerd zoeken. Dit leidt tot de volgende formulering van onze probleemstelling.

> **Probleemstelling**: *Hoe kunnen we geïnformeerde zoekmethoden ontwikkelen op zo'n manier dat programma's hun prestaties in een gegeven domein significant zien verbeteren?*

Om de probleemstelling te beantwoorden hebben we vier onderzoeksvragen geformuleerd over complexe onderwerpen op het gebied van geïnformeerd zoeken. Ze gaan over (1) de evaluatiefunctie, (2) concurrerende proof-number search algoritmen, (3) voorwaarts snoeien, en (4) de volgorde van te onderzoeken zetten.

Hoofdstuk 2 beschrijft de testomgeving die gebruikt wordt om de probleemstelling en de vier onderzoeksvragen te beantwoorden. De voorwaarden voor een geschikte testomgeving worden geformuleerd met nadruk op de begrippen *spel* en *spelprogramma*. We beschouwen het spel Lines of Action (LOA). Het hoofdstuk geeft enige achtergrond informatie, de regels, verscheidene spelkarakteristieken, zeven basisprincipes en een beschouwing over de rol van LOA in het AI-domein. Vervolgens wordt het zoekproces van het LOA-programma MIA (Maastricht In Actie) beschreven. In deze thesis wordt het gebruikt als testmechanisme.

Geïnformeerd zoeken kan niet zonder een geschikte evaluatiefunctie bestaan. In het geval van LOA is het een uitdaging om zo'n evaluatiefunctie samen te stellen. Immers, het moet de basisprincipes van het spel bevatten en tegelijkertijd moet het de winstgevendheid van de stelling die onderzocht wordt verhogen. Deze uitdaging heeft ons gebracht tot de eerste onderzoeksvraag.

Onderzoeksvraag 1: *Hoe kunnen we een sterke evaluatiefunctie opstellen voor Lines of Action?*

Hoofdstuk 3 beantwoordt de eerste onderzoeksvraag door te onderzoeken welke kenniselementen belangrijk zijn voor een LOA-evaluatiefunctie. De kenniselementen zijn gebaseerd op de basisprincipes beschreven in hoofdstuk 2. Het blijkt dat de volgende negen kenniselementen belangrijk zijn: concentratie, centralisatie, positie van het zwaartepunt, quads, mobiliteit, muren, verbondenheid, uniforme verdeling, en speler aan zet. De kenniselementen hebben geresulteerd in de evaluatiefunctie MIA IV, die getest wordt in een toernooi tegen zijn voorgangers MIA I, MIA II en MIA III. De laatstgenoemde drie evaluatiefuncties hebben goed gepresteerd op de vroegere Computer Olympiades (2000, 2001 en 2002). Experimenten laten evenwel zien dat de huidige MIA IV ze verslaat met ruime marges. Op alle onderzochte zoekdieptes wint MIA IV tenminste 75 procent van de partijen. We merken op dat veel kenniselementen in de evaluatiefunctie niet veel computertijd kosten. Aangezien gedeelten van de kenniselementen al van tevoren berekend zijn en vervolgens opgeslagen zijn in tabellen (*caching*), zijn de volledige kenniselementen snel te evalueren. Het meest belangrijke kenniselement is concentratie, op de voet gevolgd door mobiliteit. Alle kenniselementen zijn essentieel en dragen bij aan de speelsterkte. Er bestaat veel onderlinge verbondenheid en overlap tussen de kenniselementen, wat betekent dat ze elkaars prestatie beïnvloeden. Daarom zijn alle kenniselementen zorgvuldig en simultaan op elkaar afgesteld (heuristisch geoptimaliseerd). De combinatie van de negen kenniselementen heeft geresulteerd in een evaluatiefunctie die significant de speelsterkte van ons LOA-programma heeft vergroot in vergelijking met andere minder geavanceerde evaluatiefuncties. Met de huidige evaluatiefunctie heeft MIA twee eerste plaatsen veroverd, namelijk op de 8^{ste} en 9^{de} Computer Olympiade (2003, 2004).

Hoofdstuk 4 onderzoekt verscheidene (Proof-Number) PN zoekalgoritmen. De beschrijving helpt ons om onze tweede onderzoeksvraag te formuleren. Het hoofdstuk begint met het geven van een beschrijving van de originele PN-zoekmethode, en twee opvolgers van PN-search, namelijk PN^2-search en depth-first search varianten van PN-search zoals *Proof-number and Disproof-number Search* (PDS). Het originele PN-search algoritme is geformuleerd als best-first search methode. Dit heeft het nadeel dat de gehele boom in het geheugen opgeslagen moet worden. Het zoekproces kan voortijdig eindigen vanwege een tekort aan geheugen. Er zijn recentelijk enige PN-varianten geconstrueerd als depth-first zoekalgoritmen; zij gedragen zich echter wel als hun corresponderende best-first zoekalgoritmen. Het voordeel is dat de gehele boom niet in het geheugen opgeslagen behoeft worden. Het nadeel is dat de gehele boom in principe elke keer opnieuw opgebouwd moet worden. De PN zoekalgoritmen kunnen op twee manieren worden toegepast: offline en online. Eerst concentreren we ons op de *offline* applicatie van PN zoekalgoritmen. Het aantal posities dat ze kunnen oplossen (wat de kwaliteit van post-mortem analyse aangeeft) wordt op een verzameling eindspelposities getest. Behalve het vaststellen van het aantal oplossingen, onderzoeken we in hoeverre de algoritmen beperkt worden door hun werkgeheugen of door de snelheid van het zoekproces. We zien dat de mobiliteit en het verwijderen van bewezen subbomen op het tweede niveau van het PN^2-algoritme ertoe leiden dat PN en PN^2 meer eindspelposities oplossen en dit bovendien sneller doen. De prestaties van PN, PN^2, PDS en $\alpha\beta$ worden met elkaar vergeleken. Er wordt aangetoond dat de PN zoekalgoritmen $\alpha\beta$ overtuigend overtreffen in het oplossen van eindspelposities. De geheugenproblemen maken de normale PN-search echter een matige oplosser voor moeilijke problemen. PDS en PN^2 lossen significant meer problemen op dan PN en $\alpha\beta$. Maar PN^2 wordt beperkt door het werkgeheugen en PDS is drie keer langzamer dan PN^2. Na deze vergelijkingen onderzoeken we de *online* toepassing van PN-search. In het bijzonder wordt de real-time toepassing van PN-search gedurende het spel onderzocht. Uiteindelijk concluderen we dat onze methode (PN-$\alpha\beta$) de reguliere $\alpha\beta$ overtreft zoals deze in ons toernooi programma MIA is geïmplementeerd.

Wanneer we de behoefte aan geheugen minder maken door het toetsen van extra zoekprocessen, ontdekken we een cruciale geheugeneigenschap van kennis in het geïnformeerd zoekproces. Het geheugenprobleem van PN^2 en het snelheidsprobleem van PDS – besproken in hoofdstuk 4 – heeft ons tot de tweede onderzoeksvraag geleid.

> **Onderzoeksvraag 2**: *Hoe kunnen we een proof-number search algoritme ontwikkelen dat concurrerend is in snelheid en niet beperkt wordt in werkgeheugen?*

Hoofstuk 5 beantwoordt de tweede onderzoeksvraag en presenteert een nieuw proof-number search algoritme, PDS-PN. Het is een tweelaags zoekproces (zoals PN^2), die op de eerste laag een depth-first PDS toepast en op de tweede laag een best-first PN search toepast. Dus PDS-PN maakt selectief gebruik van de kracht van PN^2 en PDS. Resultaten van de experimenten met PDS-PN op een verzameling van eindspelposities worden gegeven. De experimenten laten zien dat in een aanvaardbare tijd PDS-PN effectiever is voor echte moeilijke problemen dan $\alpha\beta$ en enig ander PN zoekalgoritme.

We hebben *geïnformeerd zoeken* gedefinieerd als een zoekproces dat een regulier zoekproces toepast in samenhang met sturende kennis. Daarom onderzoeken we of het nuttig is om voorwaarts snoeimethoden, zoals *multi-cut* en *null move*, in het Principal-Variation-Search (PVS)-raamwerk te verbeteren. Dit heeft ons tot de derde onderzoeksvraag gebracht.

> **Onderzoeksvraag 3**: *Hoe kunnen we voorwaarts snoeimethoden in het Principal-Variation-Search raamwerk verbeteren?*

Hoofdstuk 6 beantwoordt de derde onderzoeksvraag. Voorwaarts snoeimethoden, zoals multi-cut en null move, worden getest op zogenaamde ALL-knopen. PVS wordt verbeterd met vier kleine essentiële toevoegingen. Het nieuwe PVS-algoritme garandeert dat voorwaarts snoeien veilig is in ALL-knopen. Experimenten laten zien dat multi-cut in ALL-knopen (MC-A) een significante besparing geeft in het aantal doorzochte knopen mits het gecombineerd wordt met een andere voorwaarts snoeiing. MC-A geeft een veilige besparing van ongeveer 40 procent in het aantal doorzochte knopen op diepte 14 in combinatie met null move en de reguliere multi-cut in CUT-knopen (MC-C). Experimenten suggereren dat parameters agressiever gekozen kunnen worden dan in MC-C. Dit leidt tot een extra verbetering. Een meer agressieve null move (*variable null-move bound*) geeft minder reductie in ALL-knopen dan ons algoritme. We zien dat MC-A nog altijd 22 procent minder knopen doorzoekt dan variable null-move bound in ALL-knopen. Het toepassen van MC-A leidt ook tot een significante verbetering van de speelsterkte van MIA. Uit deze observaties concluderen we dat MC-A een waardevolle verbetering is van PVS.

Het ordenen van zetten is een goed voorbeeld van sturende kennis in een geïnformeerd zoekproces. In $\alpha\beta$-search is het ordenen van zetten één van de hoofdtechnieken om de grootte van de zoekboom te verkleinen. Een belangrijke techniek is het dynamische ordenen van zetten, waarbij de ordening afhankelijk is van de informatie die verkregen wordt gedurende het zoekproces. Dit heeft ons gebracht tot de vierde onderzoeksvraag.

> **Onderzoeksvraag 4**: *Hoe kunnen we informatie die verkregen is tijdens het zoeken gebruiken om het ordenen van de zetten te verbeteren?*

Hoofdstuk 7 beantwoordt de vierde onderzoeksvraag door een nieuwe methode te beschrijven voor het ordenen van zetten, de relatieve historie-heuristiek. Het is een combinatie van de historie- en vlinder-heuristiek. In plaats van alleen zetten te registreren die de (een) beste zet zijn in een bepaalde knoop, registreren we ook hoe vaak een zet gespeeld wordt in de zoekboom. Beide statistieken worden gebruikt in de relatieve historie-heuristiek. Op deze manier begunstigen we zetten die gemiddeld goed zijn boven zetten die soms goed zijn. Wanneer we de historie-heuristiek vervangen door de relatieve historie-heuristiek, tonen experimenten aan dat deze methode een reductie geeft van 10 à 15 procent in het aantal doorzochte knopen. De resultaten zijn bevestigd in het Go programma MIGOS. Derhalve mogen we concluderen dat de relatieve historie-heuristiek een waardevolle techniek is om dynamische zetten te ordenen in een spelboom van een aanzienlijke diepte (meer dan 12 plies). Tenslotte merken we op dat het nut van de incrementen anders dan 1 niet veel aantoonbare verbetering laat zien in de (relatieve) historie-heuristiek voor ons LOA-programma MIA. De goede prestatie van het increment 1 kan ook het resultaat zijn van domeinspecifieke eigenschappen. De relatieve historie-heuristiek lijkt een waardevol element te zijn bij het ordenen van zetten.

In het laatste hoofdstuk keren we terug naar de vier onderzoeksvragen en de probleemstelling zoals die in hoofdstuk 1 zijn geformuleerd. Als we rekening houden met de antwoorden zien we dat er verscheidene succesvolle manieren zijn om de prestaties van geïnformeerde zoekmethoden te verbeteren. Onze verbeteringen van de evaluatiefunctie, proof-number search, voorwaarts snoeien en het ordenen van zetten hebben significante resultaten opgeleverd in ons LOA testdomein. Hierna geven we veelbelovende richtingen van vervolgonderzoek aan. Of het mogelijk is om LOA op te lossen blijft evenwel een open vraag.

Curriculum Vitae

Mark Winands was born in Valkenburg, in the province of Limburg (the most southern part of the Netherlands) on April 5, 1978. He attended secondary school, Stella Maris, in Meerssen from 1990 until 1996 and received the diploma *Gymnasium.* Immediately thereafter, he started a study Knowledge Engineering at the Universiteit Maastricht. In 1998 he received his B.Sc. degree. Then he left Maastricht for a while and continued his study first at the Baylor University, Waco, Texas (September 1998) and thereafter at the LUC, Diepenbeek, Belgium (1998–1999). In 2000, he received his M.Sc. degree in Knowledge Engineering, majoring in Artificial Intelligence, at the Universiteit Maastricht. In the same year he was chosen to participate in the Excellent Student Program 2000. As a prizewinner he received a study trip to Silicon Valley. After graduation, he worked as a Ph.D. student (AIO) at the Department of Computer Science (Institute for Knowledge and Agent Technology – IKAT), Universiteit Maastricht, The Netherlands. From the middle of June 2003 until the middle of September 2003 he worked as a visiting researcher at the Computer Games Research Institute (CGRI) of the Shizuoka University, Hamamatsu, Japan. The research resulted in several publications and this thesis. Besides performing scientific tasks, he was engaged in teaching, in representing the Ph.D. students in the council of the transnational University of Limburg (tUL), and in the organisation of several Computer Olympiads.

SIKS Dissertation Series

1998

1 Johan van den Akker (CWI[1]) *DEGAS - An Active, Temporal Database of Autonomous Objects*

2 Floris Wiesman (UM) *Information Retrieval by Graphically Browsing Meta-Information*

3 Ans Steuten (TUD) *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

4 Dennis Breuker (UM) *Memory versus Search in Games*

5 Eduard W. Oskamp (RUL) *Computerondersteuning bij Straftoemeting*

1999

1 Mark Sloof (VU) *Physiology of Quality Change Modelling; Automated Modelling of Quality Change of Agricultural Products*

2 Rob Potharst (EUR) *Classification using Decision Trees and Neural Nets*

3 Don Beal (UM) *The Nature of Minimax Search*

4 Jacques Penders (UM) *The Practical Art of Moving Physical Objects*

5 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

6 Niek J.E. Wijngaards (VU) *Re-Design of Compositional Systems*

7 David Spelt (UT) *Verification Support for Object Database Design*

8 Jacques H.J. Lenting (UM) *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000

1 Frank Niessink (VU) *Perspectives on Improving Software Maintenance*

2 Koen Holtman (TU/e) *Prototyping of CMS Storage Management*

3 Carolien M.T. Metselaar (UvA) *Sociaal-organisatorische Gevolgen van Kennistechnologie; een Procesbenadering en Actorperspectief*

[1]Abbreviations: SIKS – Dutch Research School for Information and Knowledge Systems; CWI – Centrum voor Wiskunde en Informatica, Amsterdam; EUR – Erasmus Universiteit, Rotterdam; KUB – Katholieke Universiteit Brabant, Tilburg; KUN – Katholieke Universiteit Nijmegen; RUL – Rijksuniversiteit Leiden; TUD – Technische Universiteit Delft; TU/e – Technische Universiteit Eindhoven; UL – Universiteit Leiden; UM – Universiteit Maastricht; UT – Universiteit Twente, Enschede; UU – Universiteit Utrecht; UvA – Universiteit van Amsterdam; UvT – Universiteit van Tilburg; VU – Vrije Universiteit, Amsterdam.

4 Geert de Haan (VU) *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

5 Ruud van der Pol (UM) *Knowledge-Based Query Formulation in Information Retrieval*

6 Rogier van Eijk (UU) *Programming Languages for Agent Communication*

7 Niels Peek (UU) *Decision-Theoretic Planning of Clinical Patient Management*

8 Veerle Coupé (EUR) *Sensitivity Analyis of Decision-Theoretic Networks*

9 Florian Waas (CWI) *Principles of Probabilistic Query Optimization*

10 Niels Nes (CWI) *Image Database Management System Design Considerations, Algorithms and Architecture*

11 Jonas Karlsson (CWI) *Scalable Distributed Data Structures for Database Management*

2001

1 Silja Renooij (UU) *Qualitative Approaches to Quantifying Probabilistic Networks*

2 Koen Hindriks (UU) *Agent Programming Languages: Programming with Mental Models*

3 Maarten van Someren (UvA) *Learning as Problem Solving*

4 Evgueni Smirnov (UM) *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*

5 Jacco van Ossenbruggen (VU) *Processing Structured Hypermedia: A Matter of Style*

6 Martijn van Welie (VU) *Task-Based User Interface Design*

7 Bastiaan Schonhage (VU) *Diva: Architectural Perspectives on Information Visualization*

8 Pascal van Eck (VU) *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*

9 Pieter Jan 't Hoen (RUL) *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*

10 Maarten Sierhuis (UvA) *Modeling and Simulating Work Practice BRAHMS: a Multiagent Modeling and Simulation Language for Work Practice Analysis and Design*

11 Tom M. van Engers (VU) *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002

1 Nico Lassing (VU) *Architecture-Level Modifiability Analysis*

2 Roelof van Zwol (UT) *Modelling and Searching Web-based Document Collections*

3 Henk Ernst Blok (UT) *Database Optimization Aspects for Information Retrieval*

4 Juan Roberto Castelo Valdueza (UU) *The Discrete Acyclic Digraph Markov Model in Data Mining*

5 Radu Serban (VU) *The Private Cyberspace Modeling Electronic Environments Inhabited by Privacy-Concerned Agents*

6 Laurens Mommers (UL) *Applied Legal Epistemology; Building a Knowledge-based Ontology of the Legal Domain*

7 Peter Boncz (CWI) *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*

8 Jaap Gordijn (VU) *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*

9 Willem-Jan van den Heuvel (KUB) *Integrating Modern Business Applications with Objectified Legacy Systems*

10 Brian Sheppard (UM) *Towards Perfect Play of Scrabble*

11 Wouter C.A. Wijngaards (VU) *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

12 Albrecht Schmidt (UvA) *Processing XML in Database Systems*

13 Hongjing Wu (TU/e) *A Reference Architecture for Adaptive Hypermedia Applications*

14 Wieke de Vries (UU) *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*

15 Rik Eshuis (UT) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*

16 Pieter van Langen (VU) *The Anatomy of Design: Foundations, Models and Applications*

17 Stefan Manegold (UvA) *Understanding, Modeling, and Improving Main-Memory Database Performance*

2003

1 Heiner Stuckenschmidt (VU) *Ontology-Based Information Sharing in Weakly Structured Environments*

2 Jan Broersen (VU) *Modal Action Logics for Reasoning About Reactive Systems*

3 Martijn Schuemie (TUD) *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*

4 Petkovic (UT) *Content-Based Video Retrieval Supported by Database Technology*

5 Jos Lehmann (UvA) *Causation in Artificial Intelligence and Law – A Modelling Approach*

6 Boris van Schooten (UT) *Development and Specification of Virtual Environments*

7 Machiel Jansen (UvA) *Formal Explorations of Knowledge Intensive Tasks*

8 Yong-Ping Ran (UM) *Repair-Based Scheduling*

9 Rens Kortmann (UM) *The Resolution of Visually Guided Behaviour*

10 Andreas Lincke (UT) *Electronic Business Negotiation: Some Experimental Studies on the Interaction between Medium, Innovation Context and Cult*

11 Simon Keizer (UT) *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*

12 Roeland Ordelman (UT) *Dutch Speech Recognition in Multimedia Information Retrieval*

13 Jeroen Donkers (UM) *Nosce Hostem – Searching with Opponent Models*

14 Stijn Hoppenbrouwers (KUN) *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*

15 Mathijs de Weerdt (TUD) *Plan Merging in Multi-Agent Systems*

16 Menzo Windhouwer (CWI) *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouse*

17 David Jansen (UT) *Extensions of Statecharts with Probability, Time, and Stochastic Timing*

18 Levente Kocsis (UM) *Learning Search Decisions*

2004

1 Virginia Dignum (UU) *A Model for Organizational Interaction: Based on Agents, Founded in Logic*

2 Lai Xu (UvT) *Monitoring Multi-party Contracts for E-business*

3 Perry Groot (VU) *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*

4 Chris van Aart (UvA) *Organizational Principles for Multi-Agent Architectures*

5 Viara Popova (EUR) *Knowledge Discovery and Monotonicity*

6 Bart-Jan Hommes (TUD) *The Evaluation of Business Process Modeling Techniques*

7 Elise Boltjes (UM) *Voorbeeld$_{IG}$ Onderwijs; Voorbeeldgestuurd Onderwijs, een Opstap naar Abstract Denken, vooral voor Meisjes*

8 Joop Verbeek (UM) *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale Politiële Gegevensuitwisseling en Digitale Expertise*

9 Martin Caminada (VU) *For the Sake of the Argument; Explorations into Argument-based Reasoning*

10 Suzanne Kabel (UvA) *Knowledge-rich Indexing of Learning-objects*

11 Michel Klein (VU) *Change Management for Distributed Ontologies*

12 The Duy Bui (UT) *Creating Emotions and Facial Expressions for Embodied Agents*

13 Wojciech Jamroga (UT) *Using Multiple Models of Reality: On Agents who Know how to Play*

14 Paul Harrenstein (UU) *Logic in Conflict. Logical Explorations in Strategic Equilibrium*

15 Arno Knobbe (UU) *Multi-Relational Data Mining*

16 Federico Divina (VU) *Hybrid Genetic Relational Search for Inductive Learning*

17 Mark Winands (UM) *Informed Search in Complex Games*