Learning Search Decisions

Learning Search Decisions

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. dr. A.C. Nieuwenhuijzen Kruseman,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op donderdag 11 december 2003 om 10.00 uur

door
Levente Kocsis

Promotor: Prof. dr. H.J. van den Herik
Co-promotor: Dr. ir. J.W.H.M. Uiterwijk

Leden van de beoordelingscommissie:
        Prof. dr. A.J. van Zanten (voorzitter)
        Prof. dr. A. de Bruin (Erasmus Universiteit Rotterdam)
        Prof. dr. C. Posthoff (University of The West Indies)
        Prof. dr. E.O. Postma
        Prof. dr. H. Visser

Contents

List of Figures

List of Tables

Preface

Since my childhood I am interested in the game of chess, especially in playing chess. I participated in several tournaments with much success and also with much fun. During the last years of my undergraduate studies, I looked for a combination of chess and computer science. After some deliberation I chose for machine learning and chess. When I was offered to continue this research at the Institute of Knowledge and Agent Technology (IKAT) in Maastricht, I did not hesitate. I was rewarded by the beauty of Maastricht, and by the support of many people, whom I want to acknowledge here.

First of all, I would like to thank my supervisor Jaap van den Herik. He thought me valuable lessons on doing research and writing scientific articles. Next, many thanks go to my daily adviser Jos Uiterwijk, with whom I shared my interest in research and chess. Without the help of both of them this thesis would never have appeared.

Moreover, I would like to thank the members of the Search and Games group for the fruitful discussions. Erik van der Werf gave me the opportunity to exchange ideas even at home. Besides our collaboration in LOA, Mark Winands kept me up-to-date about things in Maastricht and in The Netherlands. I enjoyed to work with both of them on Magog.

Next, I would like to thank my colleagues at IKAT for making me feel comfortable. Similarly, many thanks go to IKAT's administrative staff. With my paperwork-phobia, I could not have survived without their help.

During my stay in Maastricht I very much appreciated the support of my friends. Thanks are due to Nick Szirbik for his effort in getting me to The Netherlands. Edit Köllő and Natascha Bourdonskaia made my leisure time enjoyable. My team mates in Liège and Nijmegen provided me the thrills of chess tournaments.

Finally, I am indebted to my parents and my brother for their understanding and support in everything I did.

Levente Kocsis
Maastricht, October 2003

Chapter 1

Introduction

This thesis is in the field of machine learning and games. It presents research on learning algorithms suitable for improving decisions made during game-tree search. In this chapter we introduce our definition of search decisions and we provide a classification of search decisions. We argue why learning can be employed for search decisions, which leads to the formulation of our research question.

1.1 Games and search

Humans have always been interested in games, as entertainment and as an intellectual exercise. Both aspects can be found in today's computer games: video games are a basic entertainment tool for almost every kid and programs for chess-like games have reached (and some surpassed) the level of human intellectual capabilities. Chess-like games are the focus of this thesis. They were considered from the beginning of AI as a major test bench. Two reasons are obvious: (1) because reaching the level of human grandmasters is a worthy challenge; (2) because the abstraction of the games provides an ideal environment for developing and testing various algorithms. In some games the computers achieved a level above that of the best humans (such as in checkers [97] and Scrabble [102]) and in some they even reached perfection [46] (as in Go-Moku [4] and in nine men's morris [32]). In chess, today's programs are playing at a level of a top grandmaster; a peak performance was shown in 1997, when DEEP BLUE defeated the best human player in an exhibition match [99, 101]. However, in games like Go or shogi (japanese chess), the play of a computer program is at amateur level. All in all, it is clear that the challenge of reaching grandmaster level is (at least) partially fulfilled, but the domain of games is far from being 'dead'. Even presupposing that in the future humans are no longer worthy opponents to computers, games will remain to be a favourable test domain.

In games, as in many other domains of problem solving, search is a necessity.

1

This necessity is a result of the complexity of the domain that prohibits the humans as well as the computers to recognise instantly the next move. The role of the search is to expand the possible move sequences, and to compare the positions at the end of the sequences (i.e., in the leaf nodes). In most of the games, both for humans and for computers, the deeper the search, the better the move choice is. Unfortunately, deeper searches explore exponentially more positions and thus the required time is growing exponentially. Since the time for a move is limited, the number of positions explored during the search is also limited. Due to this limitation, there is a burden on every player to search efficiently, i.e., to explore the most promising lines rather deeply, while spending little time on the less promising ones. By doing so, it is possible to choose good moves in a limited time.

1.2   Search decisions

A typical game program relies for search on some variation of the $\alpha\beta$ algorithm [56]. Over the years several enhancements were developed to make the search more efficient. For every game program it has to be decided which search enhancements should be used, and what should be the values of the various parameters employed by the particular enhancement. The list of decisions influencing the search is very long, and any enumeration is likely to be incomplete. Among the search decisions to be taken there are: the structure of the transposition table, the move ordering, the depth of the search (or the allocated time), the 'quietness' of a position (or which moves are tactical), the search extensions, the size of the aspiration window, the method of forward pruning, the application of singular extensions and so on.

Most of these decisions are made off line by the programmer and some on line, i.e., during the play. Often a search decision can be considered at two levels. For instance, the order in which the moves are investigated at interior nodes is an on-line decision. To solve the decision a move-ordering scheme is implemented including, e.g., a transposition table, a killer heuristic, and a history heuristic. We note that the history heuristic has some parameters (e.g., the value with which the history score of a move is increased if the move produced a cutoff). The values for these parameters have to be decided too, and usually it is done off line.

In summary, search decisions are the decisions made in a game program, either on line or off line, which affect in some way the game-tree search.

1.2.1   Classes of search decisions

All search decisions are assumed to have some effect on how the search tree is expanded. However, they differ in the way of influencing the search. Some of them may produce a smaller search tree, but do not change the move chosen by the search given a search depth. Other search decisions are influencing the shape of the search (investigating some moves more deeply while ignoring others), and,

thus, may change the preferred move. Most of these are attempting to reach a compromise between the size of the search tree (which should be minimised) and the quality of the move made (which should be maximised). Some of the search decisions are even not obeying this simple rule. They aim at a possibly sub-optimal move for some 'speculative' reason such as expecting a mistake from the opponent, or gaining time. Almost as a consequence of the above differences, search decisions differ in the way their efficacy can be evaluated. Based on this difference, we define three classes of search decisions.

- *Class-P* search decisions are evaluated using only game positions and disregarding the search tree or game in which they occur.

- *Class-S* search decisions are evaluated using search trees without reference to actual move sequences, but not using only positions.

- *Class-G* search decisions are evaluated on complete move sequences (as they occur in game trees).

Usually, class-P search decisions can also be evaluated depending on their performance in the search (such as the size of the search tree). This evaluation might be even more important for the sake of the performance of the game program. However, from subsequent chapters it will become clear that the possibility to evaluate these decisions disregarding the whole search tree or the whole game results in more (time-)efficient learning algorithms. The same holds for class-S search decisions. They can also be evaluated using move sequences, but it is faster to do so using only search trees. To understand the classification better some examples are discussed in the following.

The main example for class-P search decisions is move ordering. The simplest way to evaluate a move ordering is to compare how often the move ranked highest is actually the best move; or how often the best move is ranked in the front moves. Other search decisions included in this class are detections of certain static features of the position which may influence the search engine (e.g., evaluating the status of certain groups in Go).

Class S is the most frequently occurring class of search decisions. This class includes the various search enhancements such as transposition tables, iterative deepening, search-extension schemes, forward-pruning techniques, and so on. For most of these, both the design of the enhancement and its parameters represent instances of class-S search decisions. Typically, they improve the quality of the move chosen by the search as well as the size of the search tree required to choose the move. Thus, they can be evaluated on these two terms: the quality of the move, and the size of the search tree.

Class-G search decisions have a more global effect in the game. For instance, time allocation is dividing the remaining time over the moves still to be made in the game. Another instance is opponent modelling. The idea is to speculate on a possible mistake of the opponent which might occur later in the game. These decisions can be evaluated only with the game results in hand (i.e., with a completed move sequence).

1.3   Learning in games

This section gives a short overview of the learning algorithms employed in games. A more extensive overview can be found in [31].

In most of the games, three phases can be distinguished: the opening, the middle game, and the endgame. Accordingly, game programs to select the move use algorithms specialised for each phase of the game. In the following we will discuss the learning algorithms structured according to the game phases.

In the opening phase, game programs rely on an opening book. Learning in this phase is focused on various tasks. First, if a manually constructed opening book exists, then learning can play a role on selecting which lines the program should play, and which lines should be avoided [49]. Second, the opening book can be extended by learning, or if no opening book is available it can be constructed from scratch [19, 20, 30, 67]. Third, if a program is using a relatively stable opening book, it is preferable to avoid playing the same (possibly faulty) line over and over again [8, 27, 47, 94, 100].

In the middle game two major components of the programs are involved: the search engine and the evaluation function. The learning algorithms that try to improve the search engine are discussed separately in more detail in subsection 1.3.1. The learning algorithms for the evaluation function are discussed below.

In most game programs the evaluation function is a weighted combination of several features. The most research effort in the domain of learning in games went into tuning the weights of the evaluation function. The first results date back to Samuel [94], who tuned the evaluation function of his checkers program using temporal-difference learning. His approach was replicated by several researchers [8, 9, 109], with the most successful application being in backgammon by Tesauro [108]. Besides temporal-difference learning, other approaches have been tested with some success too. The most successful of these were supervised-learning algorithms [6, 17, 64, 107]; for genetic algorithms some moderate results were reported [21].

While tuning the weights received much attention, the automatic construction of evaluation-function features was somewhat neglected. The learning algorithms for tuning the weights that use a multilayer neural network can be regarded as implicitly constructing features, the features being the hidden neurons. There were a few attempts to construct symbolic features automatically (e.g., [29, 65, 115]), but without notable success.

In the endgame, most game programs rely on endgame databases [84]. A notable exception is LOA (see [123]). Since, these databases contain factual and exact information about the game no learning is required. If the number of pieces is still relatively high, the programs use similar tools to those employed for middle-game positions. Consequently, the research on endgame-specific learning algorithms is rather limited. Most of this research is more general, and use endgames only as testbed (e.g., [39, 45, 76]). A promising idea is suggested in [91], where the endgame databases are used for training the evaluation function in awari. We believe that further research on the pre-endgame-database phase would be rewarding.

1.3.1 Learning search decisions

The efficiency of the search engine of a game program depends on a multitude of search decisions. Optimising the search decisions by hand is rather difficult for humans, since they have to take into account all effects of the search decision on the search. The alternative is to use learning algorithms for optimisation.

Learning algorithms for search decisions have been applied successfully in the past to improve the efficiency [74, 75] and, more recently, to improve the quality [3, 28, 85] of planning. These algorithms derive and refine search-control rules that have a symbolic representation, often expressed as predicate clauses. The learning algorithms employed for optimising these rules are some variations on the 'symbolic' learning algorithms such as explanation-based learning [75] and inductive logic programming [83]. However, these rule-based learning algorithms cannot be used to improve search decisions in games, simply because most of the search decisions in games are not represented as rules. Such rule-based systems were employed in the early days of computer chess (e.g., in [117]), but they have proven to be less efficient than the brute-force search engines used in today's game programs.

Although the importance was recognised [31], learning search decisions in games has received little attention so far. In the following we briefly describe the attempts found in our literature search in this area. Some of the algorithms are discussed in more detail in the chapters 3, 4, and 5, depending on the class of search decision.

Some of the existing search techniques have a 'built-in' learning mechanism. Major examples are the Multi-ProbCut [18] and the Realization-Probability Search [110, 111]. Analogously, the history heuristic [95, 96] and the transposition tables [16] can be considered as implementing a kind of on-line learning. These very specific learning algorithms cannot be used to learn any other type of search decisions.

A number of learning algorithms is designed for a specific type of search decision, but they are somewhat separated from the search decision to be improved. These include the learning algorithms for move ordering (e.g., the chessmaps heuristic [34, 35]), and the learning algorithms for selecting a set of promising moves in games with a high branching factor [23, 26, 52, 116].

Recently, there were three attempts to develop more general learning algorithms for search decisions. These were tested for a specific type of search decision but they can be applied for other type of search decisions too. The first of these was training a neural network with a genetic algorithm to prune unpromising branches of a search tree [77]. This approach can be easily modified for other types of search decisions, as we did for time allocation (see subsection 5.2.3). A second attempt was to formalise the problem of learning search decisions as an MDP [38]. Some experimental results were given for time allocation. Third, and perhaps most promising is the sequence of two gradient-descent algorithms developed for learning search extensions [11, 13, 14]. These two algorithms can, most likely, be employed for learning most of the class-S search decisions.

1.4   Research question

From the previous sections it is obvious that search decisions play an important role in the performance of game programs. Moreover, we established that search decisions are difficult to tune manually, and automatic tuning of search decisions using learning methods would be rewarding. Given these observations, we formulate our research question:

> *Which learning algorithms can improve search decisions?*

To answer this question, we will choose an instance for each of the three classes of search decisions described in subsection 1.2.1. For these instances of search decisions, we will design new learning algorithms, which will be tested experimentally.

1.5   Thesis outline

The thesis is organised as follows. Chapter 1 contains an introduction to the topics of this thesis: games, search, search decisions, and learning algorithms. In this we chapter suggest a classification of search decisions in class-P, class-S, and class-G search decisions.

Chapter 2 describes the environments employed for testing the learning algorithms for search decisions developed in the thesis. There are two test environments in the game of Lines of Action, the NN-population environment and the MIA environment, and one test environment in the game of chess, the CRAFTY environment.

Chapter 3 considers an instance of class-P search decisions: move ordering. This chapter presents a new move-ordering algorithm, the Neural MoveMap heuristic. The heuristic uses a neural network to estimate the likelihood of a move being the best in a certain position. The neural network is trained with large pattern sets consisting of game positions labelled with the best move. During the search, the moves considered more likely to be the best are examined first. In the experiments we investigate the various details of the heuristic, while comparing the heuristic with the state-of-the-art move ordering techniques. From the experimental results we conclude that the Neural MoveMap heuristic is able to improve the move ordering in all test environments.

Chapter 4 considers an instance of class-S search decisions: forward pruning. This chapter describes two new algorithms for forward pruning. The first algorithm is the TS-FPV algorithm. The TS-FPV algorithm is a tabu-search based algorithm for class-S search decisions that have a discrete representation. The algorithm is applied for learning forward-pruning vectors. The second algorithm is the RL-FPF algorithm. This algorithm is using reinforcement learning for class-S search decisions with continuous representations. The algorithm is applied for learning forward-pruning functions. Experiments with forward pruning are described for both algorithms. From the experiments we conclude that

forward pruning is beneficial and a forward-pruning scheme can be tuned either by TS-FPV or by RL-FPF.

Chapter 5 considers an instance of class-G search decisions: time allocation. A new learning algorithm for class-G search decisions is introduced, the Meta-Actor-Critic algorithm. This algorithm uses a meta-level actor-critic architecture. The application to time allocation of an existing genetic algorithm, SANE (Symbiotic, Adaptive Neuro-Evolution), is described too. These two algorithms are tested experimentally in both test games (LOA and chess). From the experiments, we conclude that the learning algorithms can improve time allocation, although the learning time can be prohibitive.

Chapter 6 presents the final conclusions and suggestions for future research.

Appendix A describes briefly three existing learning algorithms that are used several times in the thesis: temporal-difference learning, SANE and RPROP. Appendix B presents the generation of the NN-population test environment. Appendix C presents examples for the move ordering performed by the Neural MoveMap heuristic in 100 test positions.

Chapter 2

The test environments

This chapter describes the environments employed for testing the algorithms developed in the thesis. A test environment consists of a game and a game program. Two games were employed: Lines of Action (LOA), described in section 2.1 and chess, described in section 2.2. Both games are two-person zero-sum games with perfect information. LOA is a game with a smaller complexity, a shorter game length and three relatively uniform game phases (i.e., the number of pieces and the strategies involved are not changing drastically during the game). These features make LOA an attractive domain for generating new ideas, since they can be tested fast. Chess has a larger complexity, and received most attention from the game-research community. As a consequence, the available chess programs are fine-tuned, and use the state-of-the-art techniques. Thus, chess is a suitable domain for evaluating the algorithms developed. Using two games for testing provides some indication on the generalisation of these algorithms.

2.1 Lines of Action

The rules of the game Lines of Action are described in subsection 2.1.1 together with the most important game properties. Then the following two subsections describe the game programs used as test environment. Subsection 2.1.2 describes the NN-population environment and subsection 2.1.3 the MIA environment.

2.1.1 The game

Lines of Action [93] is a two-person zero-sum chess-like connection game with perfect information. It is played on an 8 × 8 board by two sides, Black and White. Each side has twelve pieces at its disposal. 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). The players alternately move a piece, starting with Black. A move takes place

9
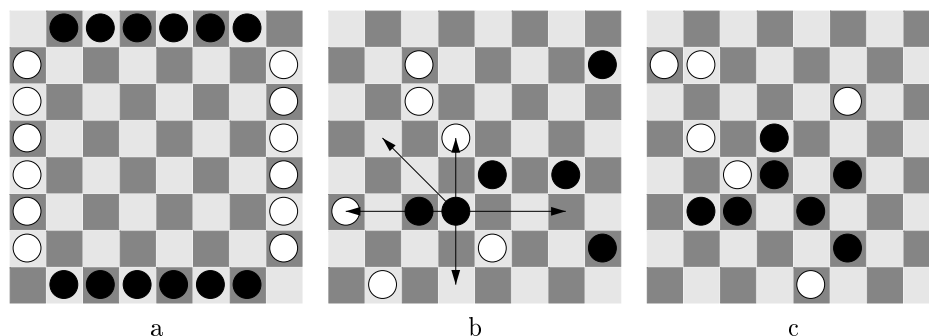
Figure 2.1: (a) The initial position of LOA. (b) An example of possible moves in a LOA position. (c) A terminal LOA position.

in a straight line (along files, ranks or diagonals), exactly as many squares as there are pieces of either colour anywhere along the line of movement (see Figure 2.1b). A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. 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 (for an example, see Figure 2.1c). In the case of simultaneous connection, the game is drawn. The connections within the unit may be either orthogonal or diagonal. If a player cannot move, this player loses. If a position with the same player to move occurs for the third time, the game is drawn.

The rules presented above are those used at the 7th Computer Olympiad (Maastricht, 2002). The state-space complexity is estimated to be 1.3×10^{24} and the game-tree complexity 10^{65}, with a branching factor of 30 and an average game length of 38 plies [118]. Although the number of pieces is slightly decreasing during the game, the strategies employed by LOA players remain mostly the same. Some of these strategies are described in [118]. Similar to humans, most computer programs do not have a clear distinction of the game phases. They use the same evaluation function throughout the whole game, and roughly the same search engine too. Additionally, in the opening phase, for the first few moves the programs use some small opening books, and in the endgame some of the programs employ some specific modules to detect a forced win [123]. The first computer program for LOA was written in 1975 at the Stanford AI lab [25]. The current best programs are YL, MONA and MIA [15].

2.1.2   The NN-population environment

The NN-population test environment consists of a simple search engine and a set of 100 neural-network evaluation functions. Each of the evaluation functions is combined with the search engine, and thus the environment includes 100 game programs. The existence of a large number of evaluation functions provides

ample diversity to the test environment. This is quite useful for evaluating the performance of the learning algorithms for search decisions.

The search engine uses the MTD(f) algorithm [87] with the history heuristic [95] for move ordering.

The evaluation functions use a feed-forward neural network with a hidden layer consisting of 40 neurons. The output and the hidden neurons use a sigmoidal activation function. A position is encoded by 65 input units. Each square is associated to a neuron (+1 for a black piece, −1 for a white one, 0 for an empty square) with an additional neuron indicating the side to move. The output neuron represents the minimax estimation of the position. The generation of the 100 neural-network evaluation functions is described in appendix B.

2.1.3   The MIA environment

The MIA environment relies on the tournament program MIA (Maastricht In Action)[1]. MIA has been written in Java and runs on every well-known operating system. In the following we describe briefly the search engine and the evaluation function of MIA. Further details on the program and its performance can be found in [15, 118, 124].[2]

MIA performs an $\alpha\beta$ depth-first iterative-deepening search. The program uses a two-deep transposition table [16], null moves [24] and PVS [71]. Before the null move is tried, the transposition table is used to prune a subtree or to narrow the window. As far as move ordering is concerned, the transposition move, if applicable, is always tried first. Next, the last two killer moves [2] are tried. All the other moves are ordered decreasingly according to their scores in the history table [95]. In the leaf nodes of the tree a quiescence search is performed. This quiescence search looks at capture moves, which form or destroy connections [124].

The evaluation function used in MIA consists of seven features. Feature 1 is the *concentration*. Positions with concentrated groups of pieces are preferred over positions where the pieces are scattered on the board. For details on this feature see [122]. Feature 2 is the *centralised centre-of-mass*. Positions with a somewhat more centralised centre of mass are preferred. Feature 3 is the *centralisation*. Pieces in the centre are preferred above pieces on the edges. Feature 4 is the *quad feature*. This feature looks at solid formations in the neighbourhood of the centre-of-mass by using quads. Details of this feature can be found in [124]. Feature 5 is the *mobility*. A bonus is given for the number of moves one has. Feature 6 is the *wall feature*. A wall is a group of pieces which blocks the opponent's pieces at the edge. Positions with walls are favoured. Feature 7 is the *side to move*. The weights of these features were tuned by TD learning [122].

---

[1] MIA can be played at the website: http://www.cs.unimaas.nl/m.winands/loa/.

[2] Below we describe the details of the MIA version used in our experiments. Meanwhile the development of MIA has progressed and the latest version, called MIA IV, has several other features introduced [119].
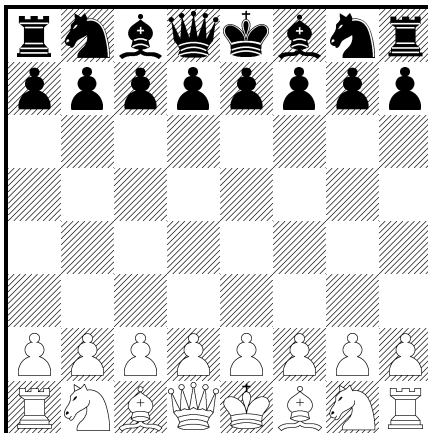
Figure 2.2: The initial position of chess.

2.2 Chess

In subsection 2.2.1 we briefly describe the game of chess. For testing the search decisions in chess, we employ the tournament program CRAFTY. An overview of CRAFTY is given in subsection 2.2.2.

2.2.1 The game

Chess is a two-person zero-sum game with perfect information. It is played on an 8×8 board by two sides, White and Black. White starts. The initial position is shown in Figure 2.2. The goal of the game is to checkmate the opponent King. A draw can occur in case of stalemate, repetition of positions, if neither player can checkmate the opponent's King with any series of legal moves, or if 50 consecutive moves were made without the movement of any Pawn and without a capture. The game can also end by agreement or by exceeding the time limit. For a complete description of the chess rules see [1].

The state-space complexity is estimated to have an upper bound of 1.8×10^{46} [22], and the game-tree complexity 10^{123}, with a branching factor of 35 and an average game length of 80 plies [4].

The game has three distinguishable phases: the opening (the first ten to fifteen moves), the middle game, and the endgame (when both sides have only a few pieces left). The three phases require rather different strategies by the players. In the opening phase computer programs as well as humans rely on some opening book. Most of the games are decided in the middle game, when both the positional evaluation and the search play a crucial role. In endgames with only up to six pieces computer programs are relying on endgame databases. In more complex endings computer programs are mostly relying on the techniques employed for the middle game, although some of the game knowledge has been

adapted to this phase, such as the evaluation function. The complex endgame is the part of the game where so far the human planning appears to be superior. Overall, the computer programs are able to play on the level of the top human grandmasters. For an overview of the state of the art of chess programs see [98].

2.2.2 The CRAFTY environment

The test environment for chess is using the tournament program CRAFTY[3] [49, 50, 51]. CRAFTY is one of the strongest chess programs whose source code is publicly available. On the on-line chess servers it consistently ranks among the highest-rated players, with a rating in the range of 2400–2500. In our experiments we used version 17.9.

CRAFTY is a fairly traditional chess program. The search engine relies on the PVS algorithm, with a transposition table and null moves. The search has two parts, the full-width search and the quiescence search. The full-width part uses fractional-ply search-depth increments. It is extended in five cases: if the move is check, if the move is a recapture, if the side to move is in check and has exactly one legal move to go out of check, if a passed Pawn is advanced to the sixth or seventh rank safely, or if the null-move search detects a mate threat. The exact extension values vary with the version of the program. The quiescence search includes pawn promotions and capture moves that appear not to lose material by using a static exchange evaluator.

For move ordering, CRAFTY considers subsequently (1) the move from the transposition table, (2) the capture moves, (3) the killer moves and (4) the remaining moves sorted in decreasing order according to their history-heuristic scores.

The main features of the evaluation function are (1) the material score, (2) the pawn scoring which considers the placement of Pawns, (3) the piece scoring which evaluates the placement of each piece as well as piece mobility, and (4) the king safety which considers the pawn structure around the King along with material threatening an attack.

---

[3]The source code of CRAFTY can be downloaded from ftp.cis.uab.edu/pub/hyatt.

Chapter 3

Move ordering

This chapter is based on[1]

1. L. Kocsis and J.W.H.M. Uiterwijk. Learning move ordering in chess. In J.W.H.M. Uiterwijk, editor, *Proceedings of the 6th Computer Olympiad Computer-Games Workshop*. Technical Report CS 01-04, IKAT, Maastricht, The Netherlands, 2001,

2. L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Move ordering using neural networks. In L. Monostori, J. Váncza, and M. Ali, editors, *Engineering of Intelligent Systems, Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2001), Lecture Notes in Artificial Intelligence, Vol. 2070*, pages 45–50. Springer-Verlag, Berlin, 2001,

3. L. Kocsis, J.W.H.M. Uiterwijk, E.O. Postma, and H.J. van den Herik. The Neural MoveMap heuristic in chess. In *Proceedings of the 3nd International Conference on Computers and Games (CG 2002), To appear in the "Lecture Notes in Computer Science" series*, 2002,

4. M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Learning in Lines of Action. In J.W.H.M. Uiterwijk, editor, *Proceedings of the 7th Computer Olympiad Computer-Games Workshop*. Technical Report CS 02-03, IKAT, Maastricht, The Netherlands, 2002,

5. M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Learning in Lines of Action. In H. Blockeel and M. Denecker, editors, *Proceedings of the 14th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2002)*, pages 371–378, 2002,

6. M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Temporal difference learning and the Neural MoveMap heuristic in the game of Lines of Action. In Q. Mehdi, N. Gough, and M. Cavazza, editors, *Proceedings of the 3rd International Conference on Intelligent Games and Simulation (GAME-ON 2002)*, pages 99–103, 2002.

This chapter deals with the problem of move ordering. The efficiency of the $\alpha\beta$ algorithm is closely related to the size of the expanded search tree and

---

[1]The author would like to thank Springer-Verlag, the editors of GAME-ON 2002, and his co-authors for the permission of reusing relevant parts of the articles in this thesis.

depends largely on the order in which the moves are considered in interior nodes. Inspecting good moves first increases the likelihood of cutoffs, decreasing the size of the search tree. We introduce a new move-ordering heuristic that relies on learning. The heuristic is tuned and evaluated by experiments.

To determine to which class of search decisions move ordering belongs, we have to look for ways of evaluating move ordering. One way is to measure how often the move ranked highest is actually the best move. This measure does not require any reference to a particular search tree or move sequence, and thus move ordering is an instance of the class-P search decisions.

In section 3.1 we describe the move-ordering techniques employed in most of the game programs. Section 3.2 presents the Neural MoveMap heuristic, a new move-ordering technique that relies on learning. The experimental set-up is given in section 3.3. The experiments with move ordering are described in section 3.4. Section 3.5 provides the conclusions on the move-ordering algorithms.

3.1   Existing techniques

The move-ordering techniques can be distinguished by their dependency of a search algorithm. Search-dependent techniques rely on information gained from previous phases of the search. Search-independent techniques rely on game-dependent knowledge. In the following we provide an overview of both types of move-ordering techniques.

3.1.1   Search-dependent move ordering

For search-dependent move ordering, we discuss five techniques below: iterative deepening, the transposition table, the killer heuristic, the history heuristic, and the countermove heuristic.

Most game programs are using *iterative deepening* to maximise the chances to search the best move first [103]. In the root of the tree, the program starts with a shallow search, and then, iteratively, the search is deepened. The idea is that the best move for a $(d-1)$-ply search is likely to be the best move (or at least one of the best moves) for a d-ply search. Henceforth, the move considered to be the best one in the previous iteration is examined first. This technique results in a rather good move ordering for the position in the root of the search tree [70]. For internal nodes iterative deepening is rarely used, due to the computational overhead.[2]

During the search information about the visited nodes is recorded in a *transposition table* [16]. One of the recorded data is the move producing a cutoff or leading to the best score. When the node is revisited and has to be re-searched, the best move from the previous search is considered first. The transposition table and the technique of iterative deepening have in common that they both

---

[2]Internal iterative deepening [5] is used by some programs when the position is not in the transposition table.

rely on previous searches of the position. Both offer a high quality choice as first move.

The *killer heuristic* [2] uses the idea that during the search, most moves are often refuted by the same move. The killer heuristic can be implemented in various ways. In its simplified form, the heuristic remembers at each search depth the move which caused the last cutoff. The next time at the same depth and if legal, the same move will be considered first. There are many improvements on this idea, such as using more killer moves, weighting the killer moves, and applying the killer moves at different depths [44]. After the introduction of the transposition table emphasis on killer moves diminished.

The *history heuristic* [95, 96] maintains a global history score for each (possible) move that indicates how often the move was found sufficient. A move is sufficient if it produces a cutoff or yields the best minimax score. When a move is found sufficient its history score is increased with an amount that decreases (usually exponentially) with the search depth. When an interior node is reached the moves are ordered by their history score.

The *countermove heuristic* [112] assumes that many moves have a 'natural' response, irrespective of the actual position in which the moves occur. The heuristic, for each non-terminal node, records the move considered to be the best, available to the opposing side. The move is recorded in a table similar to the one used by the history heuristic (with the difference that instead of scores the countermoves are stored). In every node of the search tree, if a legal countermove exists, it is given a first trial.

3.1.2 Search-independent move ordering

Below we discuss three existing techniques for search-independent move ordering. Moreover, we mention a new technique, further discussed in section 3.2.

The most common form of search-independent move ordering is the use of *game-dependent knowledge provided by a domain expert*. For instance, in chess the checking moves, captures and promotions may be investigated before silent moves.

In Chinese chess, the *guard heuristic* [54] is based on computing the guard information for each square of the board. The guard information indicates how safe the square is for a certain piece. The heuristic prefers moves with a safe target square or an unsafe origin square.

The *chessmaps heuristic* [34, 35] employs a neural network to learn the relation between the control of the squares and the influence of the move in the domain of chess. The control of the squares depends on whether one of the sides is able to move to that square without losing material. A move influences an area if one of the following conditions is fulfilled: (1) the piece is moving to that area, (2) the moving piece attacks that area, or (3) the moving piece uncovers an attack to that area. Depending on the control of the squares, the heuristic tries to determine which areas in the position are important. Moves that influence these important areas are investigated first. A disadvantage of the chessmaps heuristic is the extensive time overhead necessary to compute

the auxiliary structures [34]. This is a severe obstacle for its incorporation in competitive tournament programs.

An idea similar to the chessmaps heuristic is used in the *Neural MoveMap (NMM) heuristic* presented in detail in section 3.2.

3.2   The Neural MoveMap heuristic

The Neural MoveMap (NMM) heuristic uses a neural network to estimate the likelihood of a move being the best in a certain position. The neural network is trained with large pattern sets consisting of game positions labelled with the best move. During the search, the moves considered more likely to be the best are examined first.

The NMM heuristic has similarities with two learning approaches employed for games: the chessmaps heuristic [34, 35] and the comparison paradigm [107, 113]. The chessmaps heuristic was introduced in section 3.1. The comparison paradigm was originally used for tuning evaluation functions. In [107], a neural network was trained to compare two board positions. The positions are encoded in the input layer, and the output unit represents which of the two is better. The paradigm was employed to evaluate moves in Go [26, 116]. Accordingly, a neural network was trained to rate the expert moves higher than randomly chosen moves. Besides neural networks other approaches to represent the binary relation between moves (i.e., the comparison between two moves) were tested. These include inductive logic programming [52], and multivariate decision trees [114].

Both the chessmaps heuristic and the NMM heuristic are using a neural network for move ordering. However, there are two main differences between the two heuristics. First, in the chessmaps heuristic the neural network is trained to evaluate classes of moves, while in the NMM heuristic the neural network is tuned to distinguish between individual moves. Second, in the chessmaps heuristic the target vector represents the influence of the move on the squares of the board, while in the NMM heuristic the target information for a neural network is directly the likelihood of the move being the best in a certain position.

The NMM heuristic and the comparison paradigm have similar training information. In both cases a move is labelled for the training as the best. The difference between the two consists mostly in their purpose. The NMM heuristic is designed to be a move-ordering heuristic used in the internal nodes of the search tree, and thus it has to be fast. In contrast, the applications of the comparison paradigm are usually slower, and cannot be employed in internal nodes in a search-intensive game program due to speed limitations. When they are employed in a game program, they are designed for selecting directly the move in the root or biasing the search towards a subset of moves.

The essence of the NMM heuristic is rather straightforward. Yet, the heuristic has several details that are crucial for the heuristic to be effective, i.e., to be fast and to result in a small search tree. The details include: the architecture of the neural network (subsection 3.2.1), the construction of the pattern sets

used for training the neural network (subsection 3.2.2), and the way the neural network is used for move ordering during the search (subsection 3.2.3). In subsection 3.2.4, we formulate the questions on the NMM heuristic that need an experimental answer.

3.2.1 The architecture of the neural network

The neural network used for move ordering has to be provided with the current board position and the legal moves in the position. The board position can be encoded in the input of the neural network in a similar way as for evaluation functions. This can be done either using the raw board position as input or using some precomputed features. In the following discussion (as is the case in the experiments) we assume that encoding of the raw board position is used.

In LOA, we assign one input unit to each square of the board, with $+1$ for a black piece, -1 for a white piece and 0 for an empty square. An additional unit is used to specify the side to move. In total there are 65 units representing the board.

In chess there are more types of pieces, and consequently we assign six input units (one for every piece type) to each square. For each unit we have $+1$ for a white piece, -1 for a black piece, and 0 if the piece type is not present on the square. In total we have one active input unit for each piece on the board. Again, an additional unit is used to specify the side to move. Consequently, the encoding consists of 385 ($6 \times 64 + 1$) units.

The method of how to encode the moves leads to a more difficult choice. In the following we investigate three schemes of encoding the moves, viz. the MEI encoding (Moves Encoded in the Input), the MEOS encoding (Moves Encoded in the Output, Sparsely), and the MEOC encoding (Moves Encoded in the Output, Condensed). Below we analyse the potential weaknesses of each of them.

The MEI move-encoding scheme

The first move-encoding scheme investigated is the MEI encoding scheme. Besides the board position, this scheme inputs the selected move to the neural network. In the games investigated (LOA and chess), a move has two components: a 'from' part (denoting the piece to be moved) and a 'to' part (the destination of the move). For instance, we may assign eight input units to the file and eight to the rank of the 'from' part, and similarly eight input units to the file and eight to the rank of the 'to' part. For every move, four units corresponding to the 'from-file', 'from-rank', 'to-file' and 'to-rank' of the move are set to 1, the rest of the units are set to 0. Consequently, the size of the input layer allocated for the move is 32 units. This is additional to the units allocated for representing the board position. In the output layer one single unit representing the estimated quality of the move under discussion can be used. The move-encoding scheme is illustrated in Figure 3.1.
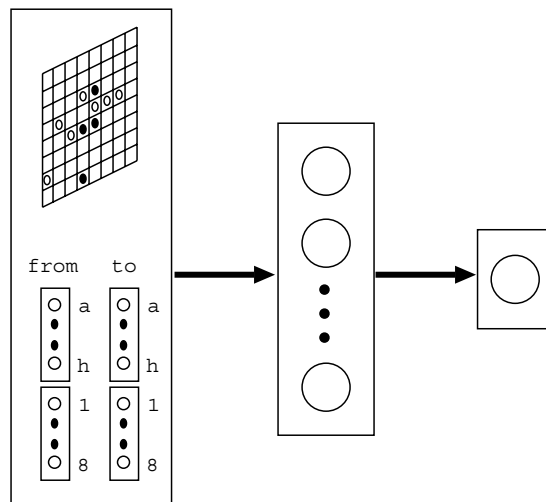
Figure 3.1: The MEI move-encoding scheme of the NMM heuristic.

If we encode the board position in an identical way as we did for the NN population in LOA (see subsection 2.1.2) and we use a hidden layer with a comparable size as for the evaluation functions (subsection 2.1.2), then the resulting neural network has a similar propagation time as the evaluation function. The neural network, when used for move ordering, has to inspect all legal moves in all internal nodes of the search tree, while the neural network for the evaluation function (the most time-consuming component usually) has to inspect only all leaf nodes. Since the ratio between the number of leaf nodes and the number of internal nodes is usually less than the branching factor, the gain from a better move ordering with this encoding scheme will most likely be lost by the overhead produced. Thus, the weakness of the MEI scheme is the time overhead.

The MEOS move-encoding scheme

A second move-encoding scheme, called the MEOS encoding, results from an analysis of the MEI encoding. To make the estimation faster, in the MEOS scheme the quality of all moves in a certain position is estimated simultaneously, instead of estimating separately for every move, as in the MEI encoding. However, encoding of all moves at once in the input vector of the neural network is not feasible. An alternative is to have separate output units for all possible moves, an output vector with 4096 units.[3] The resulting encoding scheme is shown in Figure 3.2. At first glance, the activation time is huge. However, we only have to compute the outputs corresponding to the legal moves of the

---

[3]Both in LOA and chess there are 64 squares on the board. Since the moves are identified by the current location and the new location of the piece to move, in total there is a maximum of $64 \times 64 = 4096$ possible moves.
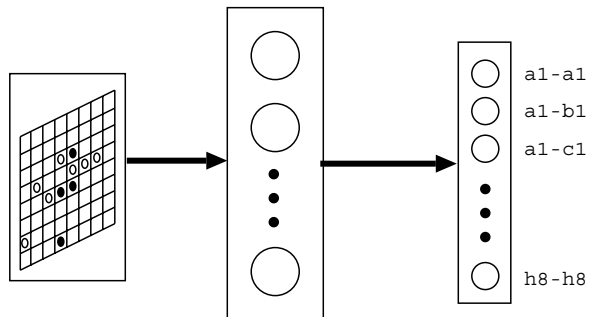
Figure 3.2: The MEOS move-encoding scheme of the NMM heuristic.

input position. Consequently, the effective size of the output vector is only the branching factor of the game. If we compare the number of effective weights for this encoding scheme with the number of weights of the evaluation function, we can observe that with the same hidden-layer size, the time necessary for move ordering in an internal node is similar to the time used by the evaluation function. Considering that the number of internal nodes is fewer than the number of leaf nodes, with identical hidden-layer sizes, the time overhead of this move ordering will be relatively small. If the size of the hidden layer is somewhat smaller (or there is no hidden layer at all), the overhead will be negligible. Thus, the weakness of the MEI scheme is not present in the MEOS scheme.

A potential weakness of this move-encoding scheme is that, since the number of real weights is high, the generalisation during the learning phase is likely to be weak.

The MEOC move-encoding scheme

A third move-encoding scheme, called MEOC encoding, is designed to overcome the weakness of generalisation for MEOS encoding. The MEOC encoding scheme is similar to the MEOS scheme, except that we now have one output unit for every 'from' square and one output unit for every 'to' square ($64 + 64 = 128$ units) instead of one output unit for every move. The scheme is illustrated in Figure 3.3. The estimation of a move will be the mean of the activation value of the output unit corresponding to its 'from' part and that of the output unit corresponding to its 'to' part. So, the training information for a certain move is generalised to the moves with the same origin or destination. To reduce the effective size of the output layer, the same technique can be applied as for the MEOS scheme. Now we have to compute the activation in the 'from' half of the output vector corresponding to the pieces of the player to move (a number of units equal to the number of pieces that can move) and the activation in the 'to' part of the legal moves (a number of units less than or equal to the branching factor). If we compare this move-encoding scheme to the previous
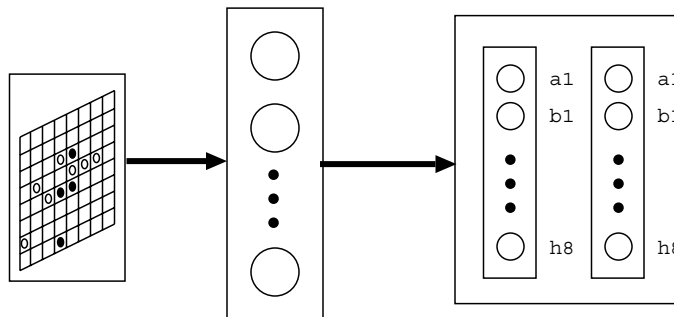
Figure 3.3: The MEOC move-encoding scheme of the NMM heuristic.

one, we observe that the real size of the output is reduced from 4096 to 128. This reduction enforces some generalisation between the moves, reducing thus the weakness of the MEOS scheme.

The effective size of the output is increased by approximately the number of pieces the side to move has. This increases the overhead (thus a potential weakness), but the increase is rather small.

Summarising the analysis of the three move-encoding schemes, we expect that the MEI encoding has a significant time overhead, and the MEOS encoding a poor generalisation. The MEOC encoding is expected to generalise better than the second scheme, but it is somewhat slower. Since the analysis does not lead to a clear conclusion, we investigate all three move-encoding schemes in our experiments (see subsection 3.4.1).

3.2.2   The construction of the pattern sets

A pattern (i.e., training instance) consists of a board position, the legal moves in the position, and the move which is considered to be the best. From these three components, determining the legal moves by an algorithm poses no problem. The choices on the other two components are more difficult. There are two questions to be answered: Where are the best moves coming from? and Which positions should be included in the training phase?

The source of the best move

In each training instance, one of the legal moves is labelled as the best. The labelling can have two sources: (1) the move played in the game (as specified in a game database, if available), and (2) the move suggested by a game program (the one that will use the neural network for move ordering). If we choose the second source, the neural network might have the advantage to incorporate the program's bias (e.g., a preference to play with Bishops). However, the disadvantage of the second source is that an additional computation is needed

for analysing all the positions. For answering the question on the source of the best move, we will compare the two sources experimentally in subsection 3.4.2.

The positions to be included

Chess games are usually divided into three phases: opening, middle game and endgame. In each of these phases different strategies are used. In middle-game positions, the original opening usually has a substantial influence on the strategies employed by chess players. Consequently, it can be beneficial to cluster the training positions by opening. In this case, we train several neural networks each specific to a certain opening. Alternatively, the positions are mixed, and only one neural network is trained. To answer the question on the positions, we compare experimentally in subsection 3.4.2 the choice of using opening-specific neural networks with that of using only one network.

3.2.3   Using the neural network during the search

When the neural network is used during the search the moves are ordered according to the network's estimation of how likely a certain move is the best. The move ordering has to be placed in the context of the move orderings already existent in the game program. In the following, we describe three approaches to include the neural network in the search: (1) the pure neural-network approach, (2) the straightforward-combination approach, and (3) the weighted-combination approach. The first two approaches can be implemented very simple, whereas the third requires additional tuning (see below). All three approaches deal only with the moves ordered by the history heuristic in the original implementation, which is not using the NMM heuristic.

The pure neural-network approach

The first approach to use the neural network during the search is by replacing in every node of the search tree the move ordering of the history heuristic by that of the neural network.

The straightforward-combination approach

The second approach combines the move ordering of the neural network and that of the history heuristic as follows. We first consider the move rated best by the neural network followed by the moves as ordered by the history heuristic, of course excluding the move picked by the neural network.

The weighted-combination approach

The third approach combines the move ordering of the neural network and that of the history heuristic, similarly to the second approach, but in a more complex way.

The history-heuristic score is built up from information collected during the search process. Therefore, it has a dynamic nature, but it has little specific connection with the position under investigation. In contrast, the scores suggested by the neural network are specific to the position; they are static, and do not gain from the information obtained in the current search process. In principle, a combination of the two scores can benefit both from information obtained in the search, and from information obtained off line by analysing a large number of positions. Such a combination can be more suitable for the position in which the moves have to be ordered.

A simple method to combine the history-heuristic score and the neural-network score is by adding them. Since the two scores are not in the same range at least one of them has to be scaled.

The distribution of the neural-network scores is specific to a certain neural network and does not depend on the search depth or opening line of a position.[4] A sample distribution is given in Figure 3.4, left. The neural-network scores are approximately normally distributed, and centred around a negative value, since most moves are not considered the best. The history-heuristic scores are known to become larger as the search progresses.

One way to normalise the history-heuristic scores is to divide them by the total number of history updates. After this normalisation, the history-heuristic scores in the different parts of the search tree are approximately in the same range. However, they are still in a range different from the neural-network scores. Therefore, we divide the normalised history-heuristic scores by a coefficient that we term the *history weight*. The resulting distribution, using the value 500 for the history weight (see below and subsection 3.4.4), is plotted in Figure 3.4, right. The history-heuristic scores are only positive, and the peak is concentrated near 0, since most of the moves rarely produce a history update during the search.

A second way to normalise the history-heuristic scores is by dividing them by the standard deviation among the scores in the current position. Experimentally, we observed that the two ways to normalise the history-heuristic scores lead to similar performances. Since the second one is computationally more expensive than the first one (we have to compute the standard deviation in every position), we choose to normalise by the number of updates.

In conclusion, the score used to order the moves (*movescore*) is given by the following formula:

$$movescore = nnscore + \frac{hhscore}{n\_hhupdate \times hhweight} \qquad (3.1)$$

where *nnscore* is the score suggested by the neural network, *hhscore* is the original history-heuristic score, *n_hhupdate* represents the number of times the history table was updated during the search, and *hhweight* is the history weight.

---

[4]Although the independence of the opening line might seem counterintuitive, this is what we observed experimentally.
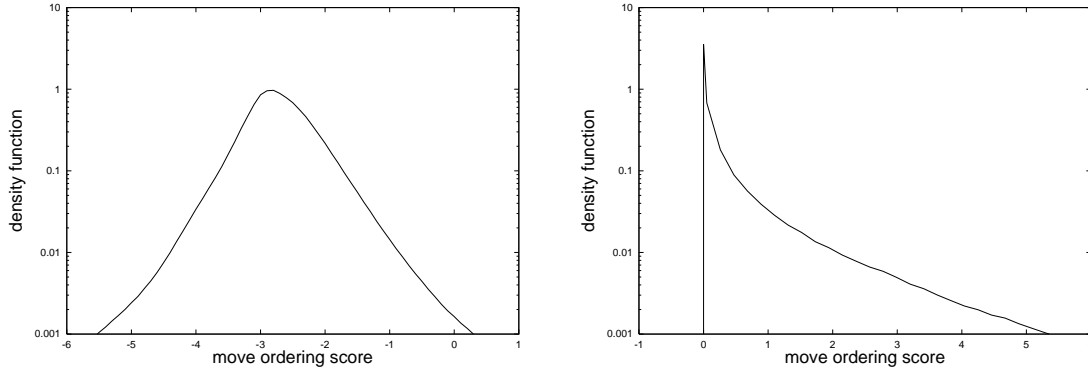
Figure 3.4: The distribution of scores for the neural network (left) and the history heuristic (right).

To use the above formula for the move score we have to choose a value for the history weight. If the weight is too large or too small, only one of the components of the score has influence. This might be beneficial in certain positions (or parts of the search tree), where one of the move-ordering methods performs poorly, but overall it is not desirable. In essence, there are two ways to set the history weight. The first one is using off-line experiments with different values, and choosing the value leading to the best performance in these experiments. The second one adapts the weight during the search. Such an adaptive method is outlined below.

In the case of a good move ordering, the moves causing a beta cutoff are inspected early. Consequently, we can use the situation in which the beta cutoff appears later as an error signal to modify the value of the history weight during the search. If the move causing a beta cutoff was considered later because both the neural network and the history heuristic scored it low, nothing could have been done. However, if, for instance, the neural network considered the move as promising, but it was scored overall lower than some other moves because the history heuristic scored it low, the history weight should be increased. We denote the situation in which the neural network scored a move producing a beta cutoff high, and the history heuristic low, by the term *nn-miss*. Analogously, we denote the situation when the history heuristic scored the move high, but the neural network low, by the term *hh-miss*. In the case of an nn-miss the history weight is increased by a small value β_{nn}, and in the case of an hh-miss the history weight is decreased by a small value β_{hh}. Depending on the update values, the history weight will converge to a value where the updates are balanced ($\Delta hhweight = 0$), or the ratio between the number of nn-misses (n_nnmiss) and the number of hh-misses (n_hhmiss) is inversely proportional to the ratio between the two update values:

$$n\_nnmiss \times \beta_{nn} = n\_hhmiss \times \beta_{hh} \qquad (3.2)$$

Experimentally, we observed that the exact values of β_{nn} and β_{hh} do not have a significant effect on the performance. They should have relatively small values (e.g., 0.1) in order to prevent large oscillations of the history weight, and should have similar magnitude (possibly equal) to give the same emphasis to both components.

3.2.4   Questions on the Neural MoveMap heuristic

In the previous subsections we presented the NMM heuristic. There are a number of questions regarding the heuristic that require an experimental answer. First we will enumerate these questions. Then we discuss them in somewhat more detail and refer to the subsection where they will be answered.

Below we list five questions regarding the NMM heuristic. The first three questions are on the details of the heuristic.

1. Which move-encoding scheme is the best?

2. How to construct the pattern sets?

3. How to use the neural network during the search?

The next two questions regard the performance of the heuristic.

4. How does the search performance of the NMM heuristic compare to the performance of other existing heuristics?

5. How good is the move ordering of the heuristic (disregarding any search)?

The first question requires the comparison of the three move-encoding schemes (viz. the MEI encoding, the MEOS encoding, and the MEOC encoding) described in subsection 3.2.1. The question is answered in subsection 3.4.1.

The second question originates from subsection 3.2.2, and has three parts: (1) What is the preferable size for the learning set?, (2) Is it better to label the positions by the moves played in the game or by the move suggested by a game program?, and (3) Is it better to have opening specific pattern sets? The question is answered in subsection 3.4.2.

For the third question we have to compare the three neural-network approaches (viz. the pure neural-network approach, the straightforward-combination approach, and the weighted-combination approach). These approaches were described in subsection 3.2.3. The question is answered in subsection 3.4.3.

The fourth question serves the evaluation of the NMM heuristic. It requires the comparison of the performance of the NMM heuristic with the performance of the history heuristic. Comparing the performance in more games (in LOA and in chess) provides an indication on the generality of the NMM heuristic. The question is answered in subsection 3.4.4.

The fifth question gives additional insights into the performance of the NMM heuristic. This comes down to investigating the quality of the moves predicted, disregarding any search. The question is answered in subsection 3.4.5.

The five questions above will be answered by the experiments described in section 3.4. Below we describe the experimental set-up.

3.3 Experimental set-up

In the experiments with respect to the five questions posed in subsection 3.2.4 we distinguish three phases: (1) the construction of the various pattern sets, (2) the training of the neural networks to predict the best move, and (3) the evaluation of the search performance of the move ordering.

In the first phase we construct the pattern sets. To construct a pattern set we have to select the positions, to generate the legal moves in the selected positions, and to label one of the legal moves as best. The details on these sets are described separately for the three test environments in subsection 3.3.1 (for the NN population), subsection 3.3.2 (for MIA) and subsection 3.3.3 (for CRAFTY).

During the second phase, the neural networks are trained to predict whether a certain move is the best. The training employs the pattern sets constructed in the first phase. The available patterns are divided into three sets: a *learning set* to modify the weights of the neural network, a *validation set* for deciding when to stop the training, and a *test set* to evaluate the move-ordering performance. In each epoch the whole learning set is presented to the neural network computing the partial derivatives of each weight with respect to the mean square of the output error (i.e., the difference between the target value and the actual output value). After each epoch, the weights are updated using the RPROP update rule [89] (for a short description, see appendix A.3). The target value for moves labelled as the best is 1, otherwise 0. The error measure for evaluation and stopping criteria is the rate of not having the highest output activation value for the 'best' move (i.e., the *error rate*). The error on the validation set is tested after each update. The training is stopped if the error on the validation set does not decrease after 100 epochs. The error rate is mostly used internally by the training algorithm, but it is also used for fast comparison between various neural networks and evaluating the predictive quality of a network.

The third phase measures the *search performance* of the neural network. The search performance represents the size of the search tree investigated by a game program using the neural network for move ordering. This measure is averaged over a set of test positions. The test positions are described together with the pattern sets in subsection 3.3.1, subsection 3.3.2 and subsection 3.3.3. In all figures in this chapter, the search performance is plotted relative to the performance when only the history heuristic is used, i.e., as node reduction. The *node reduction* is the size of the search tree investigated using the neural network for move ordering divided by the size of the search tree investigated using the history heuristic. The node reduction can be interpreted as the gain of using the NMM heuristic instead of the history heuristic.

3.3.1 The pattern sets for the NN population

The positions selected for the pattern sets for the NN population originate from self-play. The validation set and the test set contain approximately 5,000 board positions each. The size of the learning set varies. The positions appearing in

the test set and the validation set do not appear in the learning set. Similarly, the positions from the test set do not appear in the validation set.

The test set is labelled using 4- and 5-ply deep searches, while the learning set and validation set are labelled using 2- and 3-ply deep searches. The reason for a shallower search for the learning set is that it can generate large sets.

For measuring the search performance, we use a set of 600 positions covering complete game sequences.

3.3.2   The pattern sets for MIA

We used MIA to generate positions through self-play. During the games the program searched 4-ply deep with a random component in the evaluation function. For each position the move played by the program was stored as the best move for that position. The test and validation set included 3,000 position each, while the learning set included 605,780 positions. The test set, the validation set, and the learning set are disjunct.

The set of positions for measuring the search performance consists of 322 positions, which appeared in tournament play.

3.3.3   The pattern sets for CRAFTY

Chess games are usually divided into three phases: opening, middle game, and endgame. In each of these phases different strategies are used. We focus on middle-game positions. The middle-game positions are selected from game databases, classified by opening. A move is labelled best by one of two sources: by the choice of the player as is given in the game database, or by a game-playing program. For the latter, we employ CRAFTY with 2 seconds thinking time (approximately a 7-ply deep search).

We constructed four test sets, originating from the opening lines A30, B84, D85 and E97 [72]. Each of these sets consist of 3,000 positions. To each test set corresponds a validation set taken from the database with the same opening line as the test set. These sets also consists of 3,000 positions each. The positions from the test sets and the validation sets are different. The learning sets are taken from the same opening line as the corresponding test set or from a more general one. For instance, the test set A30 has three corresponding learning sets, viz. A30 with 122,578 positions, A3x with 685,858 positions, and ALL with 4,412,055 positions (see Table 3.1). Analogously, test set E97 has four corresponding learning sets: E97, E9x, KI and ALL (see Table 3.1). Other details on the learning sets are given in Table 3.1. Moreover, a sample of 100 positions from the test sets is given in appendix C.

The set of positions employed for measuring the search performance is identical to the positions from the test sets used in the training phase.

Code	ECO code	No. of positions	Opening line
A30	A30	122,578	Hedgehog System of English Opening
A3x	A30-A39	685,858	Symmetrical Variation of English Opening
B84	B84	92,285	Classical Scheveningen Var. of Sicilian Defence
SI	B80-B99	1,793,305	Sicilian Defence (Scheveningen, Najdorf)
D85	D85	124,051	Exchange Variation of Grünfeld Indian
GI	D70-D99	745,911	Grünfeld Indian
E97	E97	111,536	Aronin-Taimanov Variation of King's Indian
E9x	E90-E99	865,628	Orthodox King's Indian
KI	E60-E99	2,498,964	King's Indian
ALL	A00-E99	4,412,055	all openings

Table 3.1: Opening lines of the learning sets.

3.4 Experiments

This section describes the experiments with the NMM heuristic. The experiments are designed to provide the answers on the questions formulated in subsection 3.2.4.

The experiments comparing the three move-encoding schemes (viz. the MEI encoding, the MEOS encoding, and the MEOC encoding) are given in subsection 3.4.1. Subsection 3.4.2 deals with the choices on the construction of the pattern sets, in particular the size of the learning set, the source of the best move and the specificity of the pattern sets. Subsection 3.4.3 compares the three approaches in which the neural networks are included in the search (the pure neural-network approach, the straightforward-combination approach, and the weighted-combination approach). Additionally, for the straightforward-combination approach we describe the way the history weight is set. The performance of the NMM heuristic is evaluated in subsection 3.4.4. Subsection 3.4.5 gives insight into the predictive quality of the neural networks.

3.4.1 Move-encoding schemes

In this subsection we provide an experimental answer to the question, which of the three move-encoding schemes (viz. the MEI encoding, the MEOS encoding, and the MEOC encoding) is the best. To answer this question we employ the NN-population environment.

Using the experimental set-up described in section 3.3, we performed two experiments: (1) to obtain a neural network for each move encoding that minimises the error rate, and (2) to evaluate for each of these three neural networks the search performance.

Obtaining the three neural networks

In the first experiment, we trained the neural networks with the three move encodings, varying the size of the hidden layer and the size of the learning set. In the experiments, we noticed that the increase of the hidden layer has a
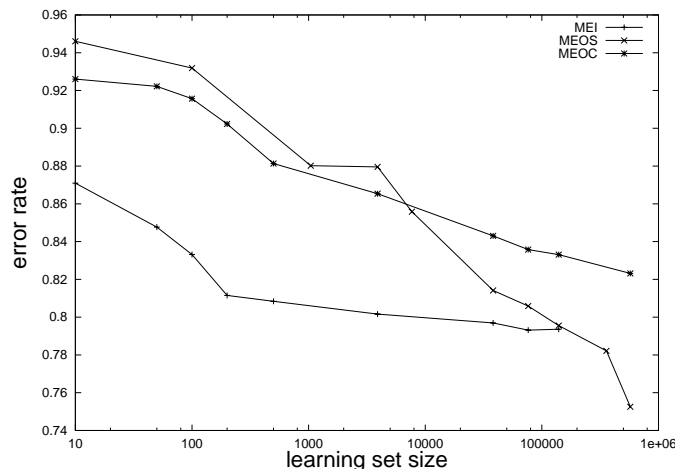
Figure 3.5: Dependency between the error rate on the test set and the size of the learning set for the three move-encoding schemes.

clear influence on the MEI encoding only, and only when large learning sets are used. Unfortunately, for this encoding the required time is already high with a small hidden layer. The remaining experiments with the NN population were performed with a hidden layer consisting of 10 units (fully connected to the input and output units) and direct connections from the input to the output units. The experiments with MIA and CRAFTY used only the direct connections from the input to the output units, without any hidden layer. The dependency between the error rate on the test set and the size of the learning set is plotted in Figure 3.5.[5] For each encoding, we selected the neural network that has the smallest error rate according to the figure.

Comparison of the three neural networks

In the second experiment, the three neural networks, corresponding to the three move encodings, are compared in terms of search performance. For this experiment we use the pure neural-network approach (subsection 3.2.3). The search performances corresponding to the three encodings are plotted in Figure 3.6 as function of the search depth. The MEI and the MEOS encoding seem to lead to similar results, and perform significantly better than the MEOC encoding. Since the time used by the MEI encoding to compute the estimated quality of the moves in a position is much higher than the time used by the other two, the MEOS encoding can be considered as the best choice. The following experiments (subsections 3.4.2 to 3.4.5) were performed using this choice.

---

[5]This figure will be discussed in more detail in subsection 3.4.2.
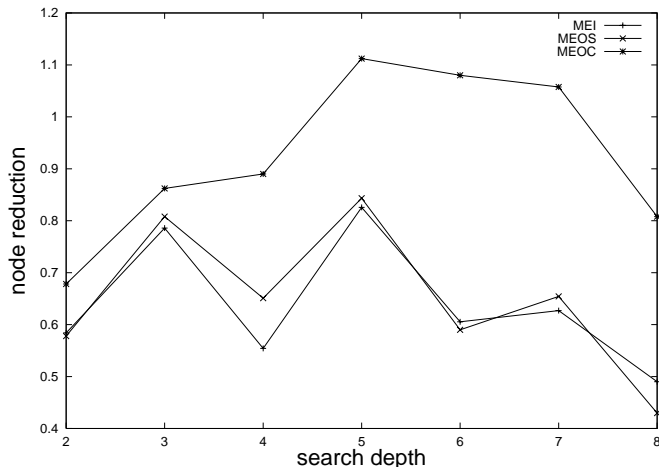
Figure 3.6: The search performance using the best neural network for each of the three move-encoding schemes as a function of the search depth.

3.4.2 Construction of the pattern sets

The questions on the construction of the pattern sets are about (1) the size of the learning set, (2) the source for the best move, and (3) the specificity of the pattern sets.

The question on the size of the learning set requires a large number of data points, and therefore it is preferable to test it in a simple environment such as the NN population. The source of the best move requires a large quantity of human games, and therefore it can be best tested in chess (using CRAFTY). The same holds for the specificity of the learning set, since extended knowledge of opening books is available only in chess.

Size of the learning set

To answer the question on the size of the learning set, we can use the result of the first experiment presented in subsection 3.4.1. In Figure 3.5 we plotted the dependency between the error rate on the test set and the size of the learning set. We observe that for all three move encodings the error rate decreases when the size of the learning set is increased.

Since the main evaluation criterion is the search performance and not the error rate, we tested in the third experiment whether the differences in error rate for the test set correspond to the search performance. In Figure 3.7 we plotted for four different search depths the search performance against the size of the learning sets which led to these networks using the MEOS encoding. We observe in the figure that the monotonic dependency between the error rate and learning-set size (noted for Figure 3.5) is preserved for search performance too.
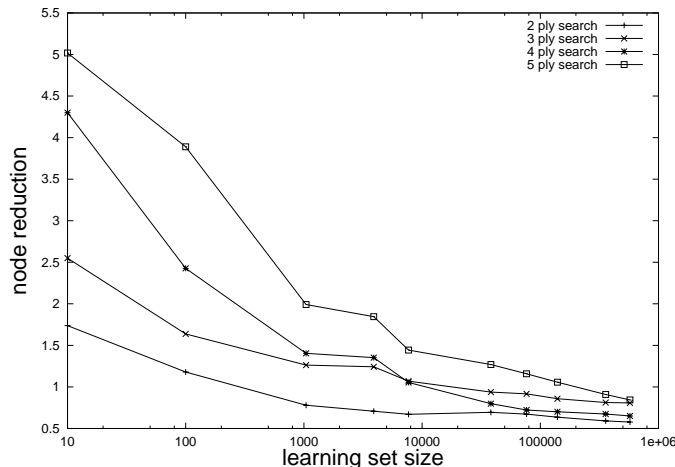
Figure 3.7: The search performance for four different search depths using for move ordering the MEOS encoding as a function of the size of the learning set.

Source for the best move

The fourth experiment is on the source of the best moves. It gives insight into the choice between the game-database move and the move suggested by CRAFTY as training target. For this purpose we used the test set E97 with two learning sets: E97 and E9x. First we focused on the learning set E97. We labelled the moves of E97 either by the moves from the game database or by CRAFTY's suggestions. Doing so we obtained two learning sets, and accordingly, after the training process, two neural networks (one for each learning set). Out of the three approaches to include the neural networks in the search (described in subsection 3.2.3), we only used the pure neural-network approach and the straightforward-combination approach. The weighted-combination approach needs additional computation to obtain the history weight, and therefore we left it out for this experiment (as well as the experiment for the specificity of the learning sets).

The search performances corresponding to the two neural networks are plotted in Figure 3.8, left. The graph shows the search performance as a function of the search depth for the neural network trained with the learning set labelled either by CRAFTY or by the game database. Since we used either the pure neural-network approach or the straightforward-combination approach, we have four curves. Similarly, Figure 3.8, right shows the search performances corresponding to the learning set E9x. We observe that if the pure neural-network approach was used, the CRAFTY labelling is beneficial for both E97 and E9x, since it investigates a smaller search tree. If the straightforward-combination approach was used the advantage disappeared. Since the CRAFTY labelling process is very time consuming[6] (especially for larger learning sets), we conclude
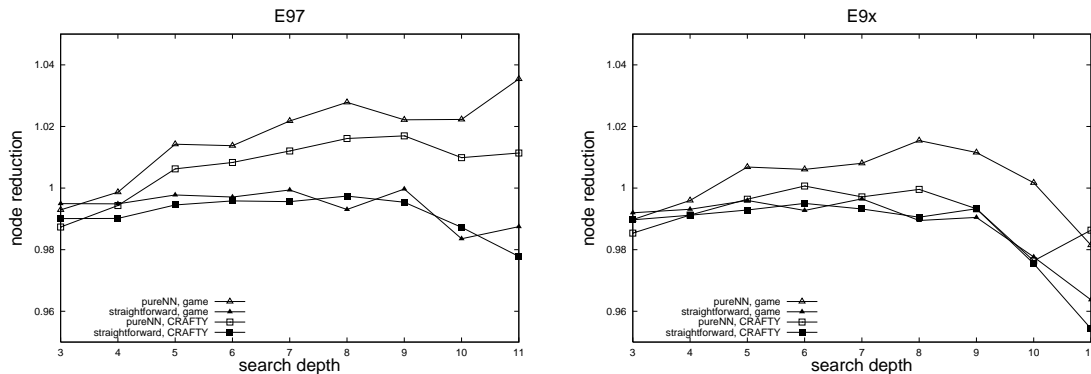
---

[6]E.g., to label E9x we needed 20 days.

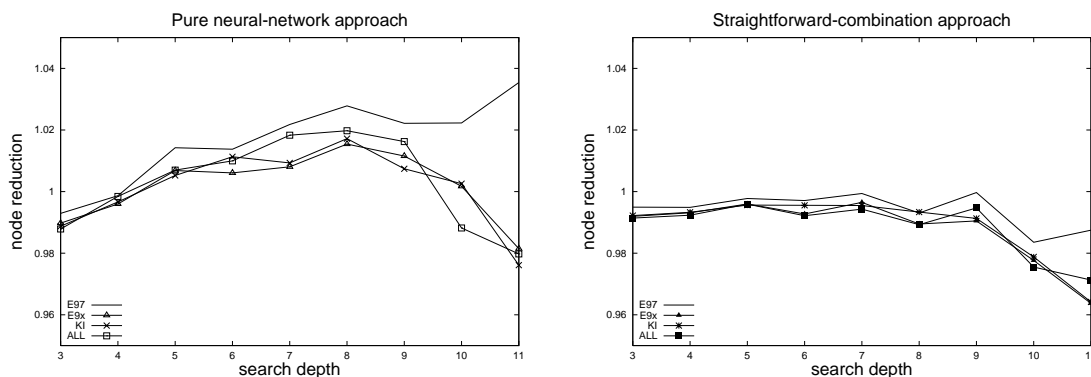Figure 3.8: Training the neural networks on game moves or on CRAFTY moves.

Figure 3.9: Training sets varying from specific to general.

that it is not useful to label the positions with a game program. Hence, it is preferable to use the moves from the game database. The following experiments are using only the moves resulting from the original games (as given in the game database).

Specificity of the pattern sets

The fifth experiment is designed to give an answer on how specific the learning set should be. Next to the two neural networks resulting from E97 and E9x, we trained two more neural networks with learning sets KI and ALL (using again E97 as test set). Out of the four sets the most specific learning set is E97, and then in this order E9x, KI, and ALL.

The search performances for the four neural networks are plotted in Figure 3.9. The performance of the neural networks corresponding to the learning sets E9x, KI and ALL were roughly equal. The neural network corresponding to the most specific learning set (E97) performed worse than the other
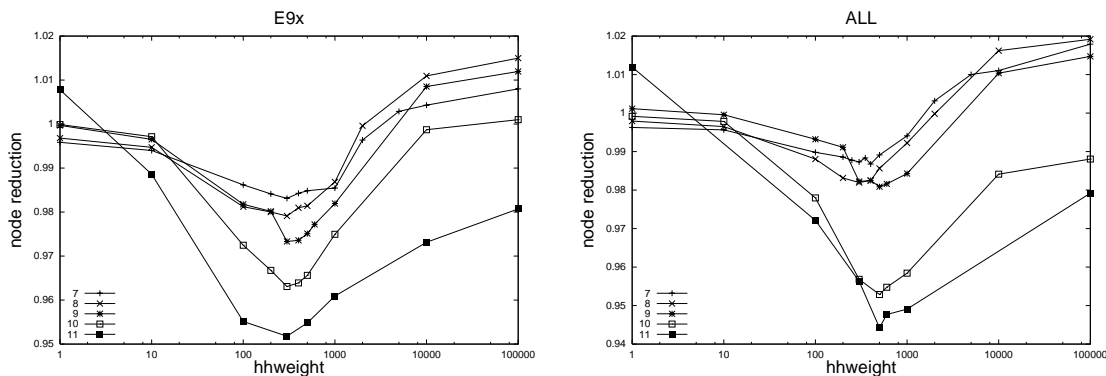
Figure 3.10: Performance of the weighted-combination approach under varying history weight. The performance is plotted for the E97 test set using two neural networks, E9x and ALL, and five search depths, from 7 to 11.

three for both the pure neural-network approach (Figure 3.9, left) and the straightforward-combination approach (Figure 3.9, right). The results suggest that using more specific learning sets does not improve the performance. For very specific sets, the performance may be decreased due to the lack of sufficient training data (in the case of E97: 111,536).

3.4.3   The three neural-network approaches

In this subsection we report on the sixth experiment, viz. on the three approaches how to include in the best way the neural network in the search. This question is closely related to the search engine and therefore it is preferable to use as test environment a strong program such as CRAFTY (in chess).

In subsection 3.4.2 we tested the performance using the test set E97. To gain a better picture of the performance of the three neural-network approaches we used three more test sets (A30, B84 and D85), next to E97. For each of the four test sets (A30, B84, D85 and E97) an opening-specific (A3x, SI, GI and E9x) and the most general (ALL) learning set were used. With these learning sets, we obtained four neural networks tailored to a certain test set and a neural network common to all test sets.

According to the pure neural-network approach and the straightforward-combination approach the neural networks were included in the search directly. Thus the corresponding performance was measured without any preparation. However, to measure the search performances for the weighted-combination approach, we first had to determine the history weight.

As described in subsection 3.2.3, the history weight can be set according to two methods, either off line or during the search. For the first method, we have to measure the search performance corresponding to different values of the history weight. Below we illustrate this mechanism for test set E97. Figure 3.10 plots
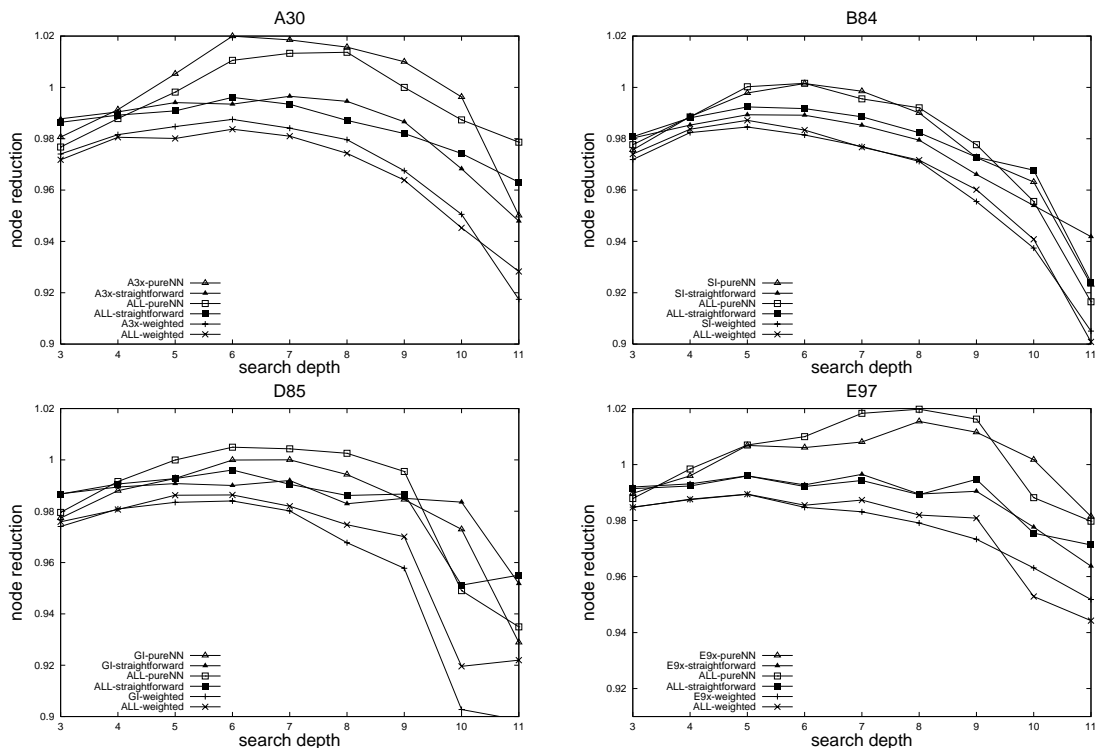
Figure 3.11: Comparing the three move orderings using the four test sets (A30, B84, D85 and E97).

the performance for different history-weight values, using the neural networks trained either with E9x (the left graph in Figure 3.10) or with ALL (right). The five curves correspond to the performances obtained for search depths 7 to 11. The best value for the history weight is the one that corresponds to the best search performance (i.e., the lowest point for a certain curve). We observed that the best history-weight value was not changing drastically with search depth. It does, however, change with the neural network.[7] The reason for this is that the best history weight was inversely proportional to the standard deviation of the neural-network scores. As measured experimentally, the standard deviation for E9x was almost twice as large as the standard deviation for ALL, and thus the best history weight for E9x was roughly the half of the best value for ALL.

For the second method, i.e., when setting the value for the history weight during the search, using the update rules described in subsection 3.2.3, we noticed that the history weight does indeed converge to a value where the equilibrium equation holds. The search performances are similar to the performances corresponding to the best static history weights from Figure 3.10.

---

[7] An appropriate history weight value for E9x is 300, and for ALL is 500.

Using the value for the history weight obtained according to the first method mentioned above, we measured the performance of the weighted-combination approach and compared it to the pure neural-network approach and the straight-forward-combination approach.

The performances of the three approaches using the neural networks mentioned earlier are plotted in Figure 3.11 (for test sets A30, B84, D85 and E97 separately). We observe that the weighted-combination approach outperformed the other two approaches on all test sets and search depths. Overall, the straightforward-combination approach seems to be better than the pure neural-network approach, although for the deepest search the difference in performance of the two approaches is less clear.

3.4.4   Performance of the Neural MoveMap heuristic

In this subsection, we compare the search performance of the NMM heuristic to the performance of the existing heuristics, in particular to that of the history heuristic. We investigate also how the NMM heuristic performed in different games.

First, we investigate the search performance in each of the three test environments (the NN population, MIA, and CRAFTY), and then we compare the results.

Performance in the NN population

For the comparison in the NN population, between the performance of the NMM heuristic and the performance of the history heuristic, we used the results of the second experiment from subsection 3.4.1. Since we already concluded that the MEOS encoding is the best choice (out of the three investigated encoding schemes), we focus only on this encoding scheme. For convenience, we reproduce the performance of the MEOS encoding as function of the search depth in Figure 3.12. We recall from section 3.3 that the performance is plotted as the size of the search tree investigated using the NMM heuristic divided by the size of the search tree using the history heuristic. Thus, in the figure the performance of the history heuristic represented by the value 1.

Comparing the performance of the NMM heuristic with the performance of the history heuristic, we observed that the reduction in the size of the tree (the speed-up) is between 20 and 50 per cent. Abstracting over the even-odd oscillation the gain initially slightly decreased until depth 5, then it increased again for deeper search trees.

Performance in MIA

In order to test more reliably the search performance of the NMM heuristic in LOA, we performed the seventh experiment, employing one of the best tournament programs available: MIA.[8]

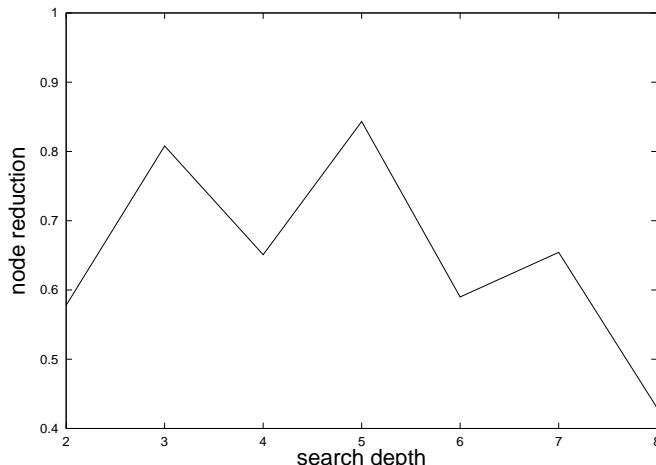[8] The author would like to thank Mark Winands for his assistance in this experiment.

Figure 3.12: The performance of the NMM heuristic in the NN population as a function of the search depth.

A neural network was trained with the pattern sets described in subsection 3.3.2. Since the evaluation function of MIA is faster than the evaluation functions from the NN population, we used the MEOS encoding and no hidden layer. In this way the overhead of the NMM heuristic during search is minimal.

The search performance was tested by using a set of 322 positions, which appeared in tournament play. During the search the pure neural-network approach (subsection 3.2.3) replaced the history heuristic, used otherwise (see subsection 2.1.3 for the move ordering in MIA).

In Figure 3.13 we plotted the performance of the NMM heuristics as a function of the search depth. We observed that the performance is decreasing until depth 5. After depth 6 the relative performance of NMM was improving with the depth. At depth 11 (which is the regular search depth under tournament conditions) the reduction was 22 per cent. The overhead of the NMM heuristic was 6 per cent. Consequently, the effective time reduction was 17 per cent.

Performance in CRAFTY

To compare in CRAFTY the performance of the NMM heuristic with the performance of the history heuristic, we used the experimental results from subsection 3.4.3, Figure 3.11. From these results we focused on the weighted-combination approach for the neural network trained with the learning set ALL. The performance, averaged over the four test sets (A30, B84, D85, and E97) is plotted in Figure 3.14.

We observed that the NMM heuristic outperformed the history heuristic with a slight margin (approximately 2 per cent) for search depths 3 to 9 and with an increasing amount for depths 10 and 11 (7.5 per cent for depth 11).
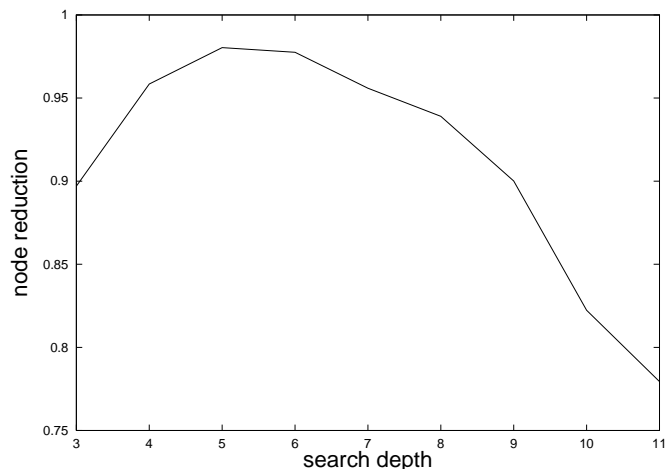
Figure 3.13: The performance of the NMM heuristic in MIA as a function of the search depth.

The time overhead for the NMM heuristic was approximately 5 to 8 per cent. However, this value can be decreased by not using the neural network for move ordering close to the leaves. The performance loss in this case was negligible (less than 1 per cent).

Comparison of the results

Considering the results presented in this section, we note that the NMM heuristic outperformed the history heuristic in all three test environments. In all test environments, and especially in MIA (Figure 3.13) and in CRAFTY (Figure 3.14), we observed a very similar pattern of results. For shallower search depths the NMM heuristic was slightly better than the history heuristic, then for medium depths the performance worsened slightly, and for deep searches the NMM heuristic outperformed the the history heuristic with an amount that increased with the depth. If we compare the performance of the NMM heuristic in chess to that in LOA, we may conclude that the heuristic is more beneficial in domains where there is little human knowledge on how to order the moves. Even in the presence of this little knowledge, it results in a notable improvement over the existent techniques.

3.4.5   Quality of the move ordering

In the eighth experiment, we provide some insight into the quality of the move ordering of the neural networks, disregarding the search. Since the author is expert only in chess, and to evaluate the quality of move ordering such knowledge is helpful, we used the CRAFTY environment. First, we investigate the predictive
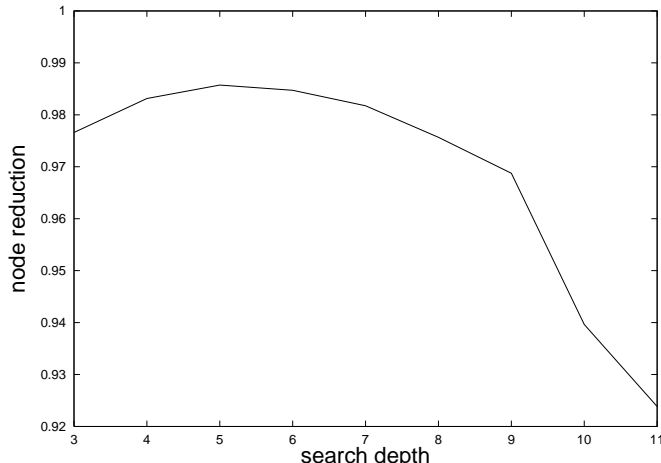
Figure 3.14: The performance of the NMM heuristic in CRAFTY as a function of the search depth.

test set	specific learning set	learning set ALL	CRAFTY's prediction
A30	0.33	0.32	0.39
B84	0.36	0.33	0.40
D85	0.35	0.31	0.45
E97	0.39	0.34	0.37

Table 3.2: Rate of correct predictions of the neural networks without any search and of CRAFTY using 7-ply searches.

quality of the neural networks (i.e., how often the best move is rated highest by the neural network), and then we illustrate the move ordering by three examples.

In order to test the predictive quality, we measured how often the move made in the game was considered to be the best by the neural network. The neural networks used for testing were those from subsection 3.4.4. The results are presented in Table 3.2. In the table, besides the success rate of the networks trained on specific learning sets and the ALL learning set, CRAFTY's predictive quality is listed for comparison purposes. We observe that the networks are able to predict the game move in one out of three positions. This performance is almost as good as that of CRAFTY with a 7-ply deep search.

We illustrate the move ordering of the neural network by three examples, representing positions where the best move is rather interesting. Other example positions are given in appendix C. The positions in the appendix were selected randomly from the test sets described in subsection 3.3.3.

Th first position, given in Diagram 3.1, has been taken from a game between two of the best chess players of the world, Kramnik and Anand. It is an open
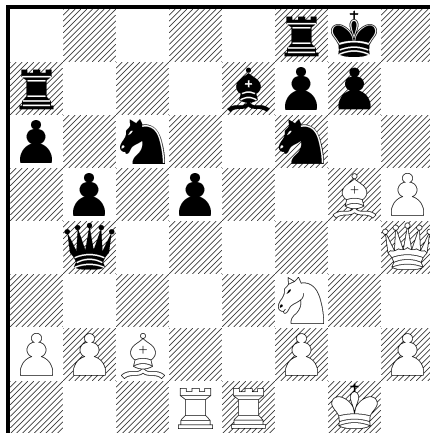
Diagram 3.1. A position from the Kramnik–Anand game played in Dortmund, 2001 (WTM). Kramnik's move was h6.
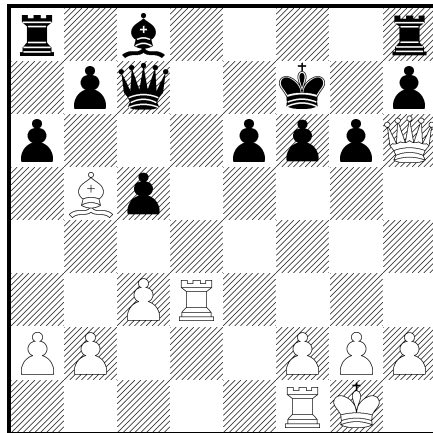The move ordering of the neural network:
Qb4 Ne5 Bf6 a3 h6 Qh3 Bb1 Bh6 Nd4 b3 Qd4 Bb3 Rd3 Qg3 Re2 Rb1 Bc1 Bg6 Rd5 Bf5 Qf4 Re3 Rf1 Bf4 Qg4 Kh1 Kf1 Re7 h3 Bd3 Ra1 Bd2 Re5 Rc1 Re6 Bh7 Rd2 Re4 Ba4 Be3 a4 Kg2 Be4 Nd2 Qe4 Rd4 Qc4

position, with some threats by White on the king side. The move made by Kramnik (h6), forces Black to a line of exchanges that leads in the end to an inferior ending for Black. Without any search, the complications on the board are hidden for the neural network. The network's main tool is to look for similarities between this position and the positions seen during the training, between the legal moves in this position and the moves labelled best in the learning set, respectively. Most of the pieces in this position are on familiar places. However, the move made by White is not frequent at all in middle-game positions. We ordered the moves using the scores of the neural network trained with the learning set ALL. Remarkably, the neural network scored the game move in the top five. Some other promising moves, like Ne5, Bf6 and Qh3 were also scored high.

The second position (Diagram 3.2) occurred in a game between the best player of the world, Kasparov, and the FIDE world champion, Ponomariov. White is a Pawn behind, and although the black King is not castled, there are relatively few pieces left on the board to build a successful attack. The last move of Ponomariov was 19...a6, attacking the white Bishop. Kasparov, instead of moving his Bishop, surprised his opponent with 20. Rh3 (threatening the queen sacrifice on h7), and won the game in the subsequent attack. The alternative move would have been Ba4. Observing the move ordering of the neural network, the move made by Kasparov is ranked the highest, while the second best move is ranked the third.

The third position, given in Diagram 3.3, has been taken from the 48th game

Diagram 3.2. A position from the Kasparov–Ponomariov game played in Linares 2002 (WTM). Kasparov's move was Rh3.

The move ordering of the neural network:

Rh3 Rfd1 Ba4 f4 Qf4 Rg3 Rd7 Bc6 Re1 Qh7 Qc1 Bd7 Qh4 Rd4 Re3 Rd6 Qh5 Qh3 Bc4 Be8 Rf3 Qe3 g3 c4 a4 b3 Kh1 Rc1 g4 f3 h4 Qd2 Rb1 h3 b4 Ba6 Qg5 Qg6 a3 Rd5 Ra1 Qg7 Rd2 Rdd1 Rd8 Qf8
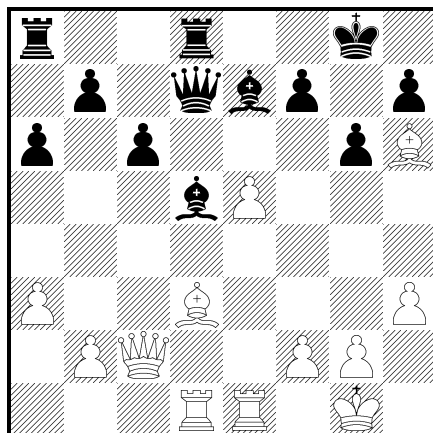
Diagram 3.3. A position from the 48th game of Kasparov–Karpov world championship match played in Moscow 1984/1985 (WTM). Kasparov's move was e6.

The move ordering of the neural network:

Bf4 Qd2 e6 Be4 f4 Bc4 Bg5 Re3 Bf8 b4 Re2 Bg7 Be2 Qe2 Re4 Rf1 Rc1 Bf1 Bg6 Rd2 h4 g4 Be3 Kh2 b3 Kh1 Qb1 Rb1 Ba6 Bf5 Qb3 Qa4 a4 Kf1 f3 g3 Bd2 Qc5 Qc4 Qc1 Qc3 Bc1 Ra1 Bb5 Qc6

of the world championship match between Kasparov and Karpov. This was the
last game of the first title match between the two players. After this game, won
by Kasparov, the match was interrupted because of its length. In the position
in the diagram, Kasparov destroyed the king side of his opponent by 22. e6 fe6
23. Bg6. In the ordering of the neural network, the move played is ranked third,
and the natural alternatives such as Bc4, Be4, and f4 are also ranked in the top
six moves.

3.5   Chapter conclusions

This chapter has considered the problem of move ordering. As we noted in
section 1.2.1 move ordering is the main instance of the class-P search decisions.
For this problem, a new heuristic in the form of a learning algorithm was de-
veloped, the Neural MoveMap heuristic. The heuristic uses a neural network
to estimate the likelihood of a move being the best in a certain position. The
moves considered more likely to be the best are examined first.

We explored the various details of the NMM heuristic, testing it experimen-
tally in the games of LOA and of chess. The experiments on the details had
the purpose of tuning the NMM heuristic. From these experiments we drew the
following conclusions.

- The best choice to encode the moves is to use one output unit for each
  possible move of the game (the MEOS encoding).

- The neural network should be trained with large learning sets. The choice
  of labelling with the game program or using the moves from the game
  database, and the choice of using learning sets specialised on opening
  lines does not influence (significantly) the performance.

- The best choice of using the neural network during the search is the
  weighted-combination approach.

From the experiments evaluating the performance of the heuristic, we drew the
following conclusions.

- The heuristic is more beneficial in domains where there is little human
  knowledge on how to order the moves, but even in the presence of this
  knowledge, it results in a notable improvement over the existent techniques
  (in particular over the history heuristic).

- The predictive quality of the neural networks is rather good, almost as
  good as a 7-ply search of CRAFTY in chess. This suggests that the neural
  networks can be useful for other tasks too, like forward pruning.

Chapter 4

Forward pruning

This chapter is based on[1]

Human game-playing is characterised by two main factors: (1) humans recognise which moves are promising, and (2) they only search the promising moves. This selectivity allows the human players to search deeply in the game tree without investigating a large number of nodes. In computer programs, the mechanism of ignoring (or not investigating) unpromising moves is termed forward pruning.

To determine the class of search decisions to which forward pruning belongs, we have to look for the evaluation criteria of forward pruning. The gain of using forward pruning is that the search has to explore a smaller tree given a certain search depth. Thus, the first criterion to evaluate forward pruning is the size of the search tree. By ignoring seemingly unpromising moves, there is a risk of missing good moves during the search. This can have the effect of choosing a weaker move after the search. Thus, the second criterion to evaluate forward pruning is the quality of the move(s) resulting from the search. Both criteria require the use of search trees, and consequently, forward pruning is a class-S search decision. We denote the size of the search tree by $ncount()$ and the quality of the move by $performance()$. These measures are usually averaged over a set of positions.

The chapter is organised as follows. In section 4.1 we describe the existing forward-pruning techniques and the existing learning algorithms that can be used for class-S search decisions. Section 4.2 presents TS-FPV, a new learning algorithm that uses tabu search for class-S search decisions that have a discrete

---

[1] The author would like to thank his co-authors for reusing relevant parts of the articles in this thesis.

representation. The algorithm is discussed for an instance of the discrete representations termed forward-pruning vectors (FPVs). A new learning algorithm that can handle continuous representations with a large number of weights, RL-FPF, is presented in section 4.3 (RL-FPF stands for Reinforcement Learning for Forward-Pruning Functions). The experiments with the TS-FPV algorithm are described in section 4.4, and the experiments with the RL-FPF algorithm in section 4.5. Section 4.6 presents the conclusions on the forward-pruning algorithms.

4.1   Existing techniques

Most game programs perform forward pruning based on information obtained by some extra search effort. Major examples of search-dependent forward pruning are adaptive null-move pruning [41], Multi-ProbCut [18] and Multi-cut pruning [12].

In contrast, search-independent forward-pruning techniques do not rely on additional search, but on some static analysis. In the early days of computer chess [10] search-independent forward pruning was a must due to the limited capabilities of the computers in that time. However, the techniques they used were not very sophisticated and need not much attention here. In recent years, only a few attempts to develop search-independent forward-pruning techniques have been published. The most important one is an extended form of futility pruning [40] developed for chess. This technique results in pruning those (frontier) nodes where the material balance plus a maximum positional score would produce a fail-high beta-cutoff. Moreover, forward pruning using some domain-dependent principles proved valuable for analysing life-and-death problems in Go [125]. A theoretical analysis [104] examining artificial trees concluded that forward pruning may be useful when there is a high correlation among the minimax values of sibling nodes in a game tree.

Most search-independent forward-pruning techniques rely on a large number of parameters. Since it is difficult to foresee the effect of these parameters on the search, hand-tuning them is very difficult. An alternative is to use automatic optimisation or learning methods for tuning.

One of the most promising algorithms to learn class-S search decisions was proposed by Björnsson and Marsland in [11, 14]. The algorithm was developed for learning search extensions but it can be applied to learn any set of continuous search parameters belonging to class S. It minimises the size of the search tree, while keeping the number of correct solutions (i.e., best moves provided with the set of positions) above a certain threshold. The learning is based on a gradient-descent algorithm, with the gradient being computed by altering each parameter in turn by a small amount and measuring the effect of it on the search (the size of the search tree and the number of correct solutions). In the experiments, the algorithm successfully improved CRAFTY's search-extension scheme, reducing the number of investigated nodes by 30 per cent. The main drawback of this algorithm is the time necessary for learning when a large number of parameters

has to be tuned. This drawback is a result of the fact that for every computation of a gradient the whole set of positions has to be searched.

A variant of the previous algorithm attempts to compute the gradients by learning from mistakes, or more precisely from the drop in the evaluation value for subsequent moves [11, 13]. The idea is that the evaluation value can decrease either because the move made was a mistake, or the new principal variation was not fully investigated. Thus, the algorithm tries to modify the extension values so that the new principal variation is part of the solution tree for the previous move too. The learning algorithm employs a 'cost model' that predicts how many nodes it takes to search an arbitrary position until a certain depth using the current extension parameters. The cost model is approximated on line for the positions involved in the search. The gradient-descent algorithm in every iteration modifies the weights in the opposite direction to the gradient of the cost model. In contrast to the previous algorithm, it estimates all the gradients in one iteration, and thus the learning time is not increasing significantly when more search parameters are tuned, but the necessity to approximate the cost model on line makes the algorithm complex and hard to use. It also depends on the validity of the underlying idea, which in certain cases may not hold.[2]

The learning part of the Realization-Probability Search [110, 111] can be regarded also as a form of learning search extensions. The search algorithm computes for each path a product termed "realization probability" that is obtained by multiplying the probabilities of the move categories to which the moves in the path belong.[3] The search is stopped after a path if the realization probability becomes smaller than some threshold. Iterative searches are performed not by increasing the search depth, but by decreasing the lower bound on the realization probability. The probability of a move category is computed (or learned) by collecting the frequency of how often a move belonging to this category was the best (found either in human games or in self-play). The search algorithm proved to be successful in the shogi tournament program GEKISASHI, which won the World Computer-Shogi Championship in 2002 [36].

Moriarty and Miikkulainen [77, 78, 79] used a neural network to prune unpromising moves in the game of Othello. The input of their neural network was represented by the raw board position; the output units corresponded to the possible moves (thus a similar architecture as the MEOS move-encoding scheme in the Neural MoveMap heuristic, see subsection 3.2.1). The weights of the neural network were evolved by a genetic algorithm, SANE (Symbiotic, Adaptive Neuro-Evolution, see appendix A.2). The experiments showed that the neural network improved the performance compared to the full-width search.

An important feature in the tuning of a search-independent forward-pruning technique is the way the forward-pruning technique is represented in terms of

---

[2]For instance, Björnsson and Marsland note that when a position is bad (the evaluation value is more than a Pawn down) the learning should be disabled, because sliding down towards the loss cannot be avoided.

[3]One might notice that the logarithm of the realization probability is the sum of the fractional extensions. In the SEX algorithm [66], this sum is termed the "interestingness" of the terminal node on the path.

its parameters. For instance, learning methods that rely on some derivative are not well suited for parameters with discrete values. Below we deal with both representations. The TS-FPV algorithm considers discrte values and is described in section 4.2. For many learning algorithms in a continuous space a large number of parameters (or weights) poses a serious problem. The RL-FPF algorithm solves this problem by estimating all the derivatives at once. The algorithm is described in section 4.3.

4.2   Forward-Pruning Vectors (FPVs)

Let us consider a search algorithm that investigates at each node only a subset of the legal moves, the moves considered as most promising by a move-ordering algorithm. The size of the subset to be investigated is specified by a number that is fixed for a certain search depth. Thus, a vector consisting of these numbers corresponding to different search depths determines the forward-pruning decisions. The elements of this vector are integer values, and so this example is an instance of a discrete representation for a class-S search decision. The integer-valued vector determining the forward-pruning decisions is termed FPV (**f**orward-**p**runing **v**ector). For the example described above the FPV can have the form $\vec{v} = (v_0, v_1, \ldots, v_D)$, with v_0 specifying how many moves have to be investigated at the root, v_1 at depth 1, and so on. Throughout this section we will use this example to make the discussion on FPVs more concrete. However, the algorithm developed for finding FPVs is not restricted to this specific example.

Since the state space of FPVs is very large, it is impossible to test each FPV, and to choose the best out of them. Consequently, we need some heuristic to explore this space. The design of such a heuristic strongly depends on how $ncount()$ and $performance()$ are determined. According to our experience, a reliable estimate for $ncount()$ can be obtained much faster than a reliable estimate for $performance()$. Thus, the large computation time needed for $performance()$ requires the heuristic to keep the number of FPVs for which $performance()$ is estimated as small as possible.

In subsection 4.2.1 we define the problem of finding FPVs as a combinatorial optimisation problem. Then we introduce a partial ordering over the FPVs, and we make two assumptions on monotonicity using the partial ordering. These two assumptions are crucial for designing the optimisation algorithm for FPVs. The algorithm, named TS-FPV, is presented in subsection 4.2.2.

4.2.1   Finding FPVs: an optimisation problem

The problem of finding an optimal FPV can be formulated as

$$
\begin{aligned}
\text{maximise} \quad & performance(\vec{v}) \\
\text{subject to} \quad & ncount(\vec{v}) \leq N \\
& \vec{v} = (v_0, v_1, \ldots, v_D) \\
& v_d \in W_d, \;\; d = \{0, 1, \ldots, D\}
\end{aligned}
$$

where D is the maximum search depth, W_d is the set of possible widths for depth d, and N is the maximum number of nodes allowed be to explored.

We define a partial ordering over the FPVs

$$\vec{u} \geq \vec{v} \quad iff \quad \forall d, \ u_d \geq v_d \tag{4.1}$$

We make two assumptions using the introduced partial ordering. First, we assume that if at some depth(s) the game-tree search is wider while at no depth it is narrower, the number of nodes explored does not decrease, i.e.,

$$\vec{u} \geq \vec{v} \Rightarrow ncount(\vec{u}) \geq ncount(\vec{v}). \tag{4.2}$$

Second, if at some depth(s) the search is wider while at no depth it is narrower, the performance does not decrease, i.e.,

$$\vec{u} \geq \vec{v} \Rightarrow performance(\vec{u}) \geq performance(\vec{v}). \tag{4.3}$$

Empirical support for these assumptions will be given in subsection 4.4.

If the assumptions hold, the problem at hand is similar to the well-known *integer knapsack problem* (for an overview of knapsack problems and algorithms, see [86]). Both problems operate on much the same solution space but in our case both the objective and the constraint function are non-linear in the elements of the FPVs, while for the integer knapsack problem they are linear. However, the assumptions highlight a similar property in the objective and constraint function: they are monotonic with respect to the introduced partial ordering. One approach to solve knapsack problems is to use approximation algorithms. These algorithms rely on the fact that the two functions are directly computable. In our problem, this is not the case, and thus approximation algorithms cannot be used. Another approach for knapsack problems is to use local-search algorithms. Due to the monotonicity of the two functions, these algorithms can be adapted for finding FPVs.

Optimisation algorithms using local search rely on the neighbourhood of two solutions. FPV $\vec{u}$ is an *upper neighbour* of $\vec{v}$ if at exactly one depth the first one produces a one 'step' wider search. In this case $\vec{v}$ is a *lower neighbour* of $\vec{u}$. The set of upper neighbours (UN) and lower neighbours (LN) can be defined as follows

$$UN(\vec{v}) = \{\vec{u} \mid \vec{u} \geq \vec{v}, \vec{u} \neq \vec{v}, \neg \exists \vec{w} \ such \ that \ \vec{u} \geq \vec{w} \geq \vec{v}, \vec{w} \neq \vec{v}, \vec{w} \neq \vec{u}\} \tag{4.4}$$

$$LN(\vec{v}) = \{\vec{u} \mid \vec{v} \geq \vec{u}, \vec{u} \neq \vec{v}, \neg \exists \vec{w} \ such \ that \ \vec{v} \geq \vec{w} \geq \vec{u}, \vec{w} \neq \vec{v}, \vec{w} \neq \vec{u}\} \tag{4.5}$$

The possible types of solutions, similarly to the integer knapsack problem, are illustrated in Figure 4.1. *Feasible* solutions are those FPVs which obey the *ncount* constraint, *infeasible* solutions are those which do not. The *maximal solution* in our case is the one which uses a full-width search, and the *minimal solution* is considering one move at each depth. Feasible solutions which do not have any upper feasible neighbouring solution are called *critical* solutions.

The critical solutions play an important role in most of the algorithms for the integer knapsack problem, and also in our case, since the optimal solution
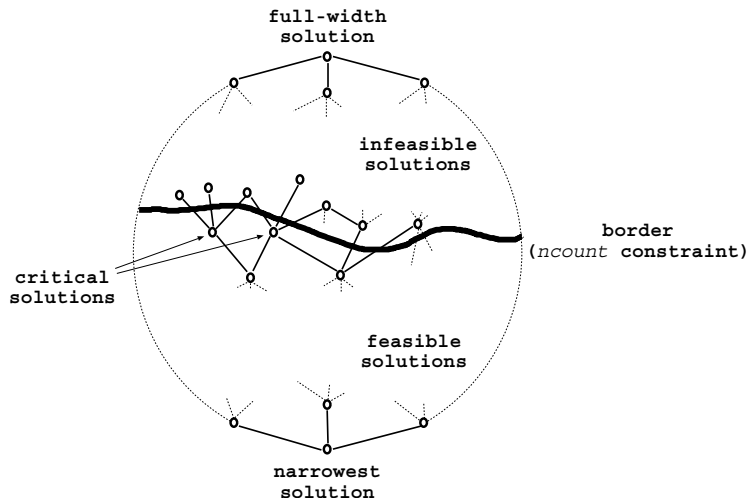
Figure 4.1: Solution space of the FPVs corresponding to a certain search depth. The circles represent FPVs, and the lines indicate the neighbourhood of two solutions.

is among the critical ones. This fact leads us to a 'requirement' for our optimisation algorithm: to estimate $performance()$ only for critical nodes. For the speed of the optimisation algorithm, it is also desirable to estimate $ncount()$ for as few non-critical solutions as possible, especially infeasible ones (since the estimation time for $ncount()$ is linear in the value of $ncount()$ itself).

4.2.2 The TS-FPV algorithm

In this section we describe the optimisation algorithm used to obtain FPVs. The algorithm, called TS-FPV, is a variation on tabu search (TS) [33]. It is in many respects similar to the algorithms suggested in [37] and [68] for knapsack problems.

The algorithm uses local search, selecting new FPVs from the neighbourhood of the current one. There are two selection criteria: *intensification* and *diversification*. Intensification is a selection criterion between candidate neighbouring solutions used to concentrate the search in the promising regions. Diversification is counterbalancing intensification by driving the search into new, unexplored regions. For the intensification criterion one can use the performance of the candidate solution or some faster estimate of it.

Diversification is realised by maintaining a long-term memory over the solutions. This memory stores the frequency of some attributes of the solutions met so far. Diversification favours solutions different from those 'stored'. In our implementation we stored the frequency of each possible width at each possible depth. When diversification is required the candidate solutions are ordered

decreasingly according to the sum of the frequencies of each value of the vector.

To avoid cycles during the search, a tabu list (TL) is maintained. The tabu list is a FIFO queue which records some properties (the tabu status) of the most recent solutions. The tabu status determines when a candidate solution is rejected. In our implementation the tabu status is the old value of the element modified in the new solution. In the tabu list we store the position of the element and the value. While this pair is in the list, solutions which have the same value on the position in question are considered tabu. The length of the tabu list is 7 (in most tabu-search implementations this number is between 5 and 9).

The TS-FPV algorithm consists of a loop including three phases. In the first phase, the search proceeds from the feasible region towards a critical solution, selecting always upper neighbours, using intensification as criterion. When a critical solution is reached its *performance*() is evaluated and if it is best so far the corresponding data structure is updated. The second phase consists of stepping away from the critical solution either towards the infeasible region by selecting an upper neighbour or towards the feasible region by selecting a lower neighbour. In either case the selection uses diversification as criterion. After the second phase the current solution can be either feasible or infeasible. We can reach the infeasible region either if an upper neighbour was selected or if our first assumption concerning *ncount*() was violated. If the current solution is infeasible, a third phase is required, in which the search proceeds from the infeasible region towards the feasible one. In this phase lower neighbours are selected using diversification. This phase stops when the new solution is feasible. The initial solution is selected either randomly, or by starting from the minimal solution. The latter has the advantage that the evaluation of *ncount* in this region is very fast. In the other case we can find our solution somewhere deep in the infeasible region. The loop ends if a certain number of cycles (*PATIENCE*) has been performed without finding a new best FPV. The pseudo code of the TS-FPV algorithm is given in Figure 4.2.

Finally, two remarks are in order. (1) We described the TS-FPV algorithm using a very specific representation for the FPVs. This representation is not a prerequisite for the algorithm. It can be applied to a much larger class of FPVs. (2) The algorithm is not assumed to be used for forward pruning only. It can be applied to other search decisions as well. However, it does depend on the monotonicity assumptions formulated in subsection 4.2.1. If these assumptions are most of the time true for a certain discrete search decision, then TS-FPV can be applied without any modifications.

4.3 Forward-Pruning Functions (FPFs)

During the search we are confronted in many nodes with the possibility of pruning the remaining moves. Such a decision is based on some search parameters, denoted *sp*. These parameters may include the search depth, the iteration depth, the number of moves already investigated in the node, the number of times the best value was changed in this node, and so on. The forward-pruning decision

```
select initial FPV v⃗;
best_perf = 0; cycle = 0;
initialise TL;
do {
    /* intensification */
    while (FeasibleUpperNeighbours(v⃗) \ TL != ∅)
        select v⃗ ∈ FeasibleUpperNeighbours(v⃗) \ TL;
    if (performance(v⃗) > best_perf) {
        best_perf = performance(v⃗);
        best_v = v⃗;
        cycle = 0;
    }
    /* step away from critical solution */
    if (randomchoice)
        select v⃗ ∈ UpperNeighbours(v⃗) \ TL;
    else
        select v⃗ ∈ LowerNeighbours(v⃗) \ TL;
    /* diversification */
    while (InFeasible(v⃗))
        select v⃗ ∈ LowerNeighbours(v⃗) \ TL;
    cycle++;
}
while (cycle < PATIENCE)
```

Figure 4.2: Pseudo code of the TS-FPV algorithm.

is represented by a function *FPF* (**F**orward-**P**runing **F**unction) that has as arguments the above-mentioned search parameters. If the value of this function is below a threshold (e.g., the value 0) the moves are pruned, otherwise they are not pruned. The FPF can be implemented for example as a neural network with weights w_i. The question is how to tune these weights by a learning algorithm.

In subsection 4.3.1 we introduce a gradient-descent update rule for tuning the weights of an FPF. The RL-FPF algorithm based on this update rule is described in subsection 4.3.2.

4.3.1   Tuning the weights

Obviously, training examples for an FPF are almost impossible to obtain. Consequently, supervised-learning methods can be ruled out. The alternative is to use some variation of reinforcement learning. Then the reward signals can be related to the size of the search tree and the quality of the suggested move.

For reinforcement learning a major problem is the credit-assignment problem. For instance, if the search expanded more nodes than allowed, it has to be determined which forward-pruning decisions were wrong and which were correct. The former have to be punished and the latter rewarded.

In order to modify the weights in a direction that maximises the reward we need some information of how the size of the search tree and the performance will change when a particular weight is changed (some form of gradient). A first

solution, given in [14], is to do an infinitesimal change in the weight and measure the effect on the size of the search tree and the performance, respectively. The main disadvantage of this method is that it has to do the measurement for each weight separately, which can be very time consuming if the number of weights is large. The learning method involved needs a relatively smooth gradient, which in turn requires averaging over a large test set.

A second solution is trying to obtain an approximation of the intended direction of the gradient. For this purpose the assumptions made in subsection 4.2.1 can be helpful again. For two FPFs, f_1 and f_2, and search parameters, sp, the partial ordering relation reads as follows:

$$f_1 \geq f_2 \quad iff \quad \forall sp, \ f_1(sp) \geq f_2(sp) \tag{4.6}$$

The two assumptions from equations 4.2 and 4.3 are changed as follows:

$$f_1 \geq f_2 \Rightarrow ncount(f_1) \geq ncount(f_2) \tag{4.7}$$

$$f_1 \geq f_2 \Rightarrow performance(f_1) \geq performance(f_2) \tag{4.8}$$

These assumptions imply that if we want to increase the performance we have to increase the values of f and if we want to decrease the size of the search tree we have to decrease the values of f. Thus, if the highest value that f can have is F_{np} (not prune) and the lowest is F_p (prune), then by minimising $(F_p - f)^2$ we decrease the size of the search tree and by minimising $(F_{np} - f)^2$ we increase the performance. Now, we can write the gradient-descent update rule for weight w_i given the forward-pruning decision f made at the moment t:

$$\Delta w_i(t) = -\frac{\partial (F_p - f)^2}{\partial w_i}(t) \cdot r^{ncount}(t) - \frac{\partial (F_{np} - f)^2}{\partial w_i}(t) \cdot r^{perf}(t) \tag{4.9}$$

where r^{ncount} is the reward function related to the size of the search tree, and r^{perf} is the reward function related to the performance.

Assuming that f is implemented as a differentiable function (such as a neural network or a linear combination of search parameters), computing the partial derivatives from equation 4.9 constitutes no problem. What remains is to choose the two reward functions, r^{ncount} and r^{perf}. These two functions determine the balance between the size of the search tree and the performance. If r^{ncount} is much higher than r^{perf}, the resulting forward pruning will favour very narrow trees, disregarding how good the suggested move is. In contrast, if r^{perf} is much higher than r^{ncount}, there will be no (or almost no) forward pruning. Choosing the appropriate functions can be a tedious process and depends on the specific experimental setting. In our experimental framework we attempt to find an FPF that maximises the performance while keeping the size of the search tree below a certain limit. The following discussion will go along this line.

Let us first investigate r^{perf}. Since we have performance information only related to a position, it is natural to have the same reward for all forward-pruning decisions within a search tree (expanding that specific position). Thus,

if we denote by T_p the set of moments when forward-pruning decisions were made while searching position p, then

$$r^{perf}(t) = r_p^{perf} \quad , \quad \forall t \in T_p \tag{4.10}$$

where r_p^{perf} is the reward for the quality of the move choice made in position p. If the performance for a position is measured in a binary way as correct move or incorrect move, then r_p^{perf} can be 1 if the move is correct and -1 if incorrect. If the performance is measured as the score difference between the (true) score of the move suggested by the search and the score of the best move, then r_p^{perf} can be exactly this difference (a more detailed description of this measure is given in subsection 4.4.4). In general r_p^{perf} can be implemented as the performance measure itself (scaled, optionally, with an arbitrary constant).

Since we choose to limit the number of expanded nodes (*ncount*) cumulated over the whole set of positions, r^{ncount} should be the same for all forward-pruning decisions. The simplest choice for this function is

$$r^{ncount} = \frac{ncount(f) - N}{N} \tag{4.11}$$

where N is the limit on the number of expanded nodes. This choice, however, does not have any significant constraint that would bind the search trees below the limit allotted (it is a simple linear function of *ncount*). To modify this we can induce a discontinuity around N using a piecewise function, e.g.:

$$r^{ncount} = \begin{cases} k_1 \cdot \frac{ncount(f)-N}{N} & , \quad ncount(f) < N \\ k_2 \cdot \frac{ncount(f)-N}{N} & , \quad ncount(f) >= N \end{cases} \tag{4.12}$$

with $k_2 >> k_1$. Whichever form the two reward functions take, it is always necessary to tune the ratio between them in order to explore the region below the limit on *ncount*. Usually, this tuning process is fairly simple.

The algorithm driven by the update rule of equation 4.9 can be used for any class-S search decision that obeys the assumptions from equations 4.7 and 4.8. For the specific case of forward pruning the update rule can be extended. If in a node of the search tree a move produces a change in the best value the forward-pruning decisions that permitted this (i.e., did not prune the move) can be strengthened. We denote the reward function for this situation with r^{cv} (reward for **c**hange in **v**alue). The target value is F_{np}. In contrast, we can punish those forward-pruning decisions that allowed to search moves that did not produce a change in the best value. We denote the reward function for this situation with r^{ncv} (reward for **n**o **c**hange in **v**alue). The target value is F_p. The resulting update rule is as follows:

$$\Delta w_i(t) = \quad -\frac{\partial(F_p-f)^2}{\partial w_i}(t) \cdot r^{ncount}(t) - \frac{\partial(F_{np}-f)^2}{\partial w_i}(t) \cdot r^{perf}(t)$$

$$-\frac{\partial(F_p-f)^2}{\partial w_i}(t) \cdot r^{ncv}(t) - \frac{\partial(F_{np}-f)^2}{\partial w_i}(t) \cdot r^{cv}(t) \tag{4.13}$$

Since these two components can be regarded as additional information that helps the optimisation process driven by the two main components (i.e., the size of the search tree and the performance), the values for r^{ncv} and r^{cv} should be significantly smaller than the values for r^{ncount} and r^{perf}.

4.3.2 The RL-FPF algorithm

Using equation 4.13 we can now design the learning algorithm, which we term RL-FPF (**R**einforcement **L**earning for **F**orward-**P**runing **F**unctions). The algorithm uses five data structures for the partial derivatives: dw^{ncount}, dw^{perf}, $dw^{ncv}[MAXPLY]$, $dw^{cv}[MAXPLY]$, dw^{total}. The first four data structures are for storing the four derivatives of equation 4.13 until the corresponding reward is known. The rewards r^{ncount} and r^{perf} are the same for all forward-pruning decisions occurring during the search, and thus the sums of the corresponding derivatives can be stored in one variable each. The rewards r^{ncv} and r^{cv} are different for the different search nodes, and thus the sums require different variables for all search depths. The fifth data structure, dw^{total}, corresponds to $\Delta w_i(t)$. The RL-FPF algorithm is inserted in the search algorithm of a game program in such a way that certain parts of the RL-FPF algorithm are activated under specifically-detailed conditions (events) of the search algorithm. Thus, the sequence of actions performed for the RL-FPF algorithm depends on the particular search algorithm used. Therefore, the RL-FPF algorithm is not presented in a sequential order, but as a list of actions triggered by events of the search algorithm. The algorithm uses a set of test positions that are searched by the game program. One pass of the test set is considered an epoch. The algorithm updates the weights of the FPF (i.e., batch learning) using a chosen update rule. As a matter of choice, in our experiments we use RPROP (see appendix A.3) for this purpose. Although other update rules should work as well, RPROP has the advantage that it is not using the size of the derivative (which is not very reliable for our algorithm), but only the direction of it. The search events and the actions of the algorithm are as follows.

- **Event**: a new epoch starts
  Action: dw^{total} and dw^{ncount} are set to 0

- **Event**: the search starts for a new position
  Action: dw^{perf} is set to 0

- **Event**: a new search node is visited
  Action: $dw^{cv}[ply]$ and $dw^{ncv}[ply]$ are set to 0

- **Event**: a possible forward pruning is considered
  Actions: f is computed using the current search parameters sp and weight vector w; $\partial f / \partial w$ is computed;
  $(\partial f / \partial w) \times (F_{np} - f)$ is added to dw^{perf} and to $dw^{cv}[ply]$;
  $(\partial f / \partial w) \times (F_p - f)$ is added to dw^{ncount} and to $dw^{ncv}[ply]$;
  the forward-pruning decision is made using f

- **Event**: the best value in a node is changed
  Actions: $dw^{cv}[ply]$ is multiplied with r^{cv} and added to dw^{total};
  $dw^{cv}[ply]$ and $dw^{ncv}[ply]$ are set to 0

- **Event**: a search is finished for a node
  Action: $dw^{ncv}[ply]$ is multiplied with r^{ncv} and added to dw^{total}

- **Event**: the search is finished for a position p
  Action: dw^{perf} is multiplied with r_p^{perf} (depending on the selected move)
  and added to dw^{total}

- **Event**: an epoch is finished
  Actions: dw^{ncount} is multiplied with r^{ncount} (depending on the number
  of nodes investigated in this epoch) and added to dw^{total};
  the weight vector w is updated using dw^{total}

So far, we described the RL-FPF algorithm with a focus on forward pruning. However, equation 4.9 can handle a broader type of class-S search decisions. Eventually, it can be extended with more problem-specific reward functions, as we did for forward pruning in equation 4.13. The necessary conditions for the algorithm are that the FPF has to be differentiable and the monotonicity assumptions should hold most of the time.

Two examples for class-S search decisions that obey the required conditions are search extensions and adaptive null-move pruning. In both cases the representation should be differentiable. Moreover, the monotonicity assumptions should be valid for both, as noted in the following. If the search is extended more deeply for a move class, the investigated search tree will be larger and the quality of the move chosen better than in the normal search (thus the assumptions are valid). In the case of the adaptive null-move pruning [41] the reduction factor (R) can be modified during the search. When the value of R is increased, the size of the search tree decreases, as well as the quality of the move. Thus the assumptions are valid again.

4.4   Experiments with TS-FPV

This section describes the experiments with the TS-FPV algorithm. First, we introduce in subsection 4.4.1 three forward-pruning schemes with discrete representations (FPV variants). We use two test environments for the experiments with TS-FPV: the NN-population environment and the CRAFTY environment. The experimental set-up for the NN-population environment is described in subsection 4.4.2. The experiments with NN population given in subsection 4.4.3 include a smaller experiment for testing the monotonicity assumptions (formulated in subsection 4.2.1) and the main experiment for measuring the performance of the FPVs. Subsection 4.4.4 describes the experimental set-up for the CRAFTY environment, while the results of the experiments for measuring the performance of the FPVs in chess are provided in subsection 4.4.5.

d \ it	1	2	3	4	5
1	?	?	?	?	?
2		?	?	?	?
3			?	?	?
4				?	?
5					?

a

d \ it	1	2	3	4	5
1	v1	v1	v1	v1	v1
2		v2	v2	v2	v2
3			v3	v3	v3
4				v4	v4
5					v5

b

d \ it	1	2	3	4	5
1	v1	v2	v3	v4	v5
2		v2	v3	v4	v5
3			v3	v4	v5
4				v4	v5
5					v5

c

d \ it	1	2	3	4	5
1	v5	v4	v3	v2	v1
2		v5	v4	v3	v2
3			v5	v4	v3
4				v5	v4
5					v5

d

Figure 4.3: FPV variants: (a) generic, (b) FPV-d, (c) FPV-it, (d) FPV-l.

4.4.1 Three FPV variants

In section 4.2, we discussed for simplicity the variant of FPV that uses one value for each search depth. This value represented the number of moves investigated in an internal node at the specific depth. Besides the search depth, we can consider another 'dimension': the iteration number. If we consider these two dimensions instead of a linear vector, the FPVs are more similar to triangular matrices (as illustrated up to depth 5 in Figure 4.3a). If all the values are treated separately, there is the risk that the space of FPVs becomes very large. Alternatively, we may constrain the values along one of the sides of the triangle. Then, we obtain three variants of FPVs: *FPV-d* uses the same value at a certain **d**epth (Figure 4.3b), *FPV-it* uses the same value in a certain **it**eration (Figure 4.3c), and *FPV-l* uses the same value at a certain distance to the **l**eaves (Figure 4.3d).

4.4.2 Experimental set-up for the NN-population environment

The TS-FPV algorithm explores the space of FPVs in order to find the FPV that maximises the quality of the suggested moves (i.e., *performance*()), while keeping the number of expended nodes during the search (i.e., *ncount*()) below a certain limit. Thus, for the experiments with TS-FPV, we have to measure the two functions, *ncount*() and *performance*(), we have to set a limit on *ncount*() and we have to specify the space of possible FPVs. We assess the algorithm by the performance of the best FPV found by the algorithm. This performance is

compared to the performance of a full-width searcher.

To measure *ncount*(), we count the number of nodes explored. There are mainly two ways to measure *performance*(): (1) by matching the moves suggested by the search against the moves known as correct for the test positions, and (2) by game results. In LOA it is difficult to obtain positions annotated with the correct move, and annotating positions using a game program is not reliable, since the evaluation functions from the NN population do not have the same quality as, for instance, the evaluation functions of the top chess programs. Consequently, to measure the performance of an FPV, we have to rely on game results. The diversity of the NN population (see subsection 2.1.2) provides a suitable environment for measuring performance in this way. The NN population includes 100 game programs. The performance of an FPV is the tournament performance of a test player (a medium strength player, chosen from the set of 100 programs) playing with both colours against each of the whole set of players. The test player is using the FPV-d variant for forward pruning, while its opponents are searching full width. For the intensification criterion we need a fast estimate of the performance. This is done by using a tournament with only 10 players.

The value for the limit on *ncount*() corresponds to the number of nodes used by the test player when searching full-width to a certain depth, d. The depth values we investigate are 1, 2, 3, 4, and 5. The 100 opponents are searching to depth d.

Specifying the space of possible FPVs implies choosing the set of possible width values for the FPVs (W_d) and the length of the FPVs (viz. the search depth of the test player using the FPV). W_d is chosen as $\{1, 2, 3, 4, 5, 6, 7, 10, 20, all\}$. The length of the FPVs is varied between 2 and 7. For each FPV length a different TS-FPV run is performed.

4.4.3   Experimental results with the NN population

In this subsection we describe two experiments: (1) a smaller experiment for testing the two assumptions underlying the TS-FPV algorithm, and (2) the main experiments for testing the FPVs that result from the TS-FPV algorithm.

In the first experiment, performed with the NN-population environment, we tested our two assumptions formulated in subsection 4.2.1. We compared how frequently the assumptions hold for neighbouring FPVs. In the case of *ncount*() this frequency was approximately 98 per cent and in the case of *performance*() between 92 and 95 per cent. These values suggest that for two arbitrary FPVs these assumptions are indeed reasonable.

In the second experiment, we measured the performance of the FPVs resulting from the TS-FPV algorithm in the NN-population environment. The performances of the best FPVs and the full-width searcher are plotted in Figure 4.4. Additionally to the full-width searcher and the FPV searcher we included as comparison the test player which searched full-width, but 1 ply deeper than its opponents (and the base-line full-width searcher). From Figure 4.4 we can conclude that forward pruning using the best FPVs obtained with our algorithm
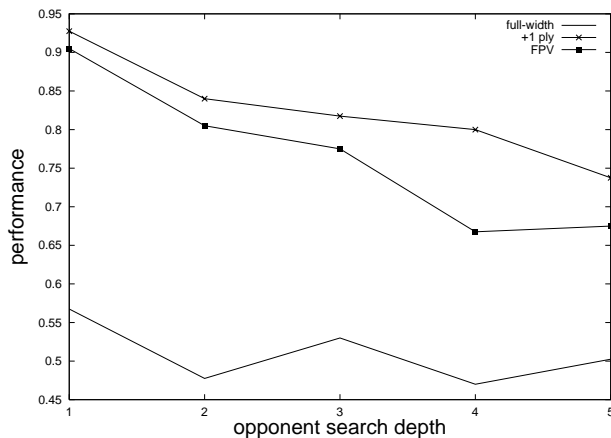
Figure 4.4: The tournament performance of the best FPVs, the full-width and the 1-ply-deeper searcher as a function of search depth.

has indeed beneficial effects. It gives the player almost the gain of an extra ply, which otherwise would require an increased amount of time.

As an illustration, we consider the best FPV values for opponent search depth 5 and their performances (see Table 4.1). The best FPVs are given in more detail in appendix D.1 (Table 4.1 is one of the five tables given there). In Table 4.1 the first line represents the result for the 5-ply full-width searcher, and the second line for the 6-ply full-width searcher (both in italics). The three best FPVs obtained by the optimisation algorithm for FPV length 6 and 7 are given. The best FPV uses a 7-ply search, and investigates 20 moves in the root, all moves at depth 1, 20 moves at depth 2, all moves at depth 3, 10 moves at depth 4, 7 moves at depth 5, and 4 moves at depth 6. The performance corresponding to this FPV is given in bold.

Although the limited number of data points (see appendix D.1) do not allow us to make too many conclusions about which is the best FPV length or which is the best FPV structure, we do observe some trends. For each of the five depths, the best FPV was found at a depth which is only 1 or 2 ply deeper than the reference depth. This fact suggests that the FPV player should not be too confident regarding the quality of the move ordering. Regarding the structure of the best FPVs, we can note that in most cases (except when the search was extremely narrow), the closer to the root the wider the FPV-based search was.

4.4.4 Experimental set-up for the Crafty environment

The forward pruning is implemented in the game program CRAFTY. In an internal node, CRAFTY considers subsequently (1) the move from the transposition table, (2) the capture moves, (3) the killer moves, and (4) the remaining (silent) moves sorted according to their history-heuristic scores. Out of these only after

depth	FPV values	performance
5	*all all all all all*	*0.5025*
6	*all all all all all all*	*0.7375*
6	all all 10 all 10 all	0.5725
6	all all all 20 20 6	0.537500
6	all all all all 10 6	0.502500
7	20 all 20 all 10 7 4	**0.6750**
7	20 all 10 all 10 7 6	0.6675
7	all all 10 20 10 7 6	0.6500

Table 4.1: FPV values and performances for opponent search of depth 5.

the first silent move a forward pruning is considered. To improve the move or-
dering the NMM heuristic is used with the weighted-combination approach and
the neural network *ALL* (see subsection 3.4.4).

To measure the performance we use a set of 3,000 test positions. The po-
sitions are middle-game positions, similar to those used for move ordering (see
subsection 3.3), but selected randomly without any clustering. The perfor-
mance is measured as score difference in the following way. For each legal move
in the test positions the minimax score is computed with a 12-ply deep search.
These scores are considered as the 'true' scores for the respective moves. The
performance for a position is measured as the difference between the score of
the move suggested using forward pruning and the score of the move suggested
without forward pruning with the reference search depth. The performance for
the whole test set is the average of the performances for the positions included
in the test set. According to our experiments with the score difference 3,000
positions are roughly sufficient to have statistically significant measurements.
If the performance is measured using self-play a few thousands of games are
needed (as argued in [42, 43]). Thus with the score difference the performance
can be measured an order of 100 times faster. For the intensification criterion,
the performance is estimated using 100 positions.

The values for the limit on *ncount*() correspond to the number of nodes used
by the regular search (i.e., without forward pruning), searching until a reference
depth (4, 5, 6, 7, and 8).

The set of possible FPV values in the experiments was {1, 2, 3, 4, 5, 10, 20,
30, 40, 50, *all*}. The reference search depth was varied between 4 and 8. The
length of the FPVs (and thus the search depth when forward pruning is used)
was 1 more than the reference depth. For each reference search depth and each
FPV variant a different TS-FPV run is performed searching for the best FPVs.

4.4.5   Experimental results with Crafty

In the third experiment, we measured the performance of the FPVs for the
three FPV variants in the Crafty environment. The performance of the best
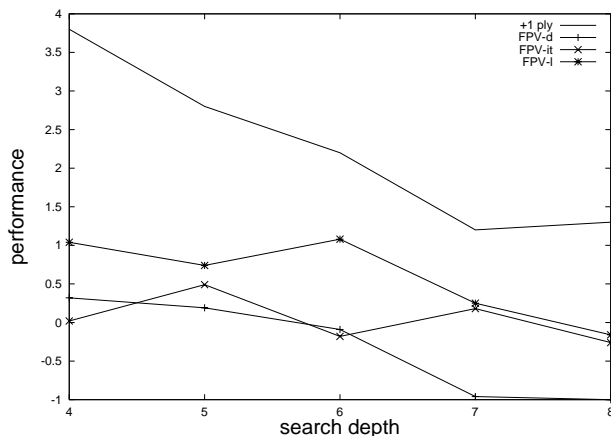
Figure 4.5: The performance (score difference) of the best FPVs and the 1-ply-deeper searcher as a function of the search depth.

FPVs for the three FPV variants and the performance of the 1-ply deeper full-width searcher is plotted in Figure 4.5. The performance of the reference search depth is always 0. By analysing the plot, we observed that out of the three FPV variants the best was FPV-l, then FPV-it, and then FPV-d. Naturally, by investigating much larger trees, the full-width searcher given an extra ply performed better than all three FPV variants (which is in accordance with the assumption formulated in equation 4.3). Compared to the reference value (i.e., 0), FPV-l performed better for all search depths except depth 8. The weaker performance for this depth might be due to the fact that in the time available for the experiment TS-FPV investigated only a small percentage of the critical solutions.[4] Thus, it is possible that some of the unexplored critical solutions had a better performance than 0. If this is not the case, then forward pruning using FPVs is too 'risky' for deeper searches (the gain of an extra ply is outweighed by the mistakes made by ignoring some moves). FPV-it had a similar performance as the reference value, and FPV-d performed worse most of the time. The odd-even oscillation for both FPV-l and FPV-it is interesting to observe. For FPV-l it is in the same 'phase' as the odd-even oscillation of the 1-ply deeper search, but for FPV-it it is reversed.

As an illustration, we consider the best FPV values for reference search depth 7 and their performances (see Table 4.2). The best FPVs are given in more detail in appendix D.2. Table 4.2 has three subtables corresponding to the three FPV variants. In each subtable, the first entry specifies the performance of the 8-ply full-width searcher (in italics), and the next five entries the performance of the

---

[4]The time necessary for evaluating the performance of an FPV grows exponentially with the search depth. This restricts the number of critical FPVs evaluated for deeper searches. Additionally, the number of critical solutions of the FPV space also grows exponentially with the search depth (i.e., the length of the FPVs). Thus, the deeper the search, the smaller the fraction of investigated critical FPVs and total number of critical FPVs.

FPV-d	
FPV values	performance
all all all all all all all all	1.2
all 5 20 all all 4 10 20	**-0.96**
all 20 20 40 10 10 20 5	-1.06
4 5 all 10 30 all all all	-1.12
all 4 all 30 all 10 5 20	-1.24
3 10 all 20 20 20 all all	-1.24

FPV-it	
FPV values	performance
all all all all all all all all	1.2
all 20 all 3 all 10 10 20	**0.18**
10 30 5 40 40 5 50 10	-0.68
all all all all all 30 30 10	-1.11
10 2 2 4 4 30 all 5	-2.96
all all 20 2 10 30 all 5	-3.08

FPV-l	
FPV values	performance
all all all all all all all all	1.2
all 3 all 30 10 10 all 20	**0.25**
10 10 10 all 20 10 all 20	0.09
10 4 10 10 all all all 20	0.09
4 10 20 30 20 10 all 20	0.06
20 5 all 30 10 10 all 20	0.00

Table 4.2: FPV values and performances of the three variants for reference search depth 7.

top five FPVs found by the TS-FPV algorithm. We recall from subsection 4.4.4 that only after the first silent move a forward pruning is considered, and thus, the values of the FPVs indicate the number of investigated silent moves. The FPV-d investigates all silent moves in the root, 5 at depth 1, 20 at depth 3, all at depth 4, all at depth 5, 4 at depth 6, 10 at depth 7, and 20 at depth 8. The performance of this vector is -0.96. The best FPV-it investigates all silent moves in the 1st iteration, 20 in the 2nd, all in the 3rd, 3 in the 4th, all in the 5th, 10 in the 6th, 10 in the 7th, and 20 in the 8th. The performance of this vector is 0.18. The best FPV-l investigates all silent moves at a distance of 8 from the leafs, 3 at a distance of 7, all at a distance of 6, 30 at a distance of 5, 10 at a distance of 4, 10 at a distance of 3, all at a distance of 2, and 20 at a distance of 1. The performance of this vector is 0.25. These three FPVs are illustrated in Figure 4.6.

Out of the three FPV variants, the most interesting one is FPV-l (since it has

a

d \ it	1	2	3	4	5	6	7	8
1	all	all	all	all	all	all	all	
2		5	5	5	5	5	5	5
3			20	20	20	20	20	20
4				all	all	all	all	all
5					all	all	all	all
6						4	4	4
7							10	10
8								10

b

d \ it	1	2	3	4	5	6	7	8
1	all	20	all	3	all	10	10	20
2		20	all	3	all	10	10	20
3			all	3	all	10	10	20
4				3	all	10	10	20
5					all	10	10	20
6						10	10	20
7							10	20
8								20

c

d \ it	1	2	3	4	5	6	7	8
1	20	all	10	10	30	all	3	all
2		20	all	10	10	30	all	3
3			20	all	10	10	30	all
4				20	all	10	10	30
5					20	all	10	10
6						20	all	10
7							20	all
8								20

Figure 4.6: The best FPV of each variant for reference depth 7: (a) FPV-d, (b) FPV-it, (c) FPV-l.

the best performance). For this variant we observe that the last two values of the best FPVs are *all* and 20. This pattern can be noticed for the three deepest searches in Table D.4. A reason behind this might be that the last value has to be relatively low otherwise the FPV is infeasible, and the one-but-last value is collecting the 'full' information in the next iteration.

4.5 Experiments with RL-FPF

This section describes the experiments with the RL-FPF algorithm. The RL-FPF algorithm depends strongly on the search algorithm. Therefore, we tested the RL-FPF algorithm only in the CRAFTY environment, which has the most well-tuned search algorithm. First, we describe the experimental set-up, with additional focus on the search parameters. This is followed by the experimental results.

4.5.1 Experimental set-up

To test RL-FPF in chess we have to include it in a chess program, we have to choose a representation for the FPF and we have to select the search parameters which are the input for the FPF. The chess program used is CRAFTY. We chose to represent the FPF as a neural network without a hidden layer. The reason to exclude a hidden layer was that the search would be very slow otherwise (a similar consideration as for the Neural MoveMap heuristic). The output of the neural network represented the choice of pruning at different 'checkpoints'. These checkpoints imply that we do not test for pruning before each move (that would be too time consuming), but only from time to time. Checkpoints were after 1, 2, 3, 4, 5, 10, 20, 30, 40, and 50 silent moves. If the decision suggested by the FPF was to prune, the remaining moves were not investigated. The search parameters were encoded in the input of the FPF. Since there is no hidden layer, to enhance the representation ability of the FPF we used very sparse encoding for most of the search parameters. For integer search parameters that have a small range we used binary encoding, with one unit for each possible value. For

numerical search parameters with a large scale we used either one unit or a 'digital' encoding. The digital encoding uses one unit for each power (e.g., for the number 524 using base 10 and the maximum value 1000, we use three units with the first unit having value 5, the second 2, and the third 4).

The list of search parameters tested in our experiments is the following.

- *IT* (binary encoding): iteration number

- *SD* (binary encoding): the search depth of the current node

- *D2L* (binary encoding): distance to the leaf

- *NCOUNT* (digital encoding or 1 unit): number of nodes investigated so far

- *ED* (digital encoding or 1 unit): the remaining search depth of the current node including the extensions [5]

- *NVC* (binary encoding): number of times the value of the current node has been changed

- *SVC* (digital encoding): the sum of the move numbers (i.e., the order in the move list) when the value of the current node has been changed

- *MOSCORE* (digital encoding or 1 unit): the move ordering score of the next move to investigate (using the history score, the neural network's score or the weighted combination of the two)

- *HASH* (2 units): one of the units represents whether a transposition move was available and the second whether the score of the transposition move is still the best

- *BOARD* (raw encoding): the board position

For training the FPF we used RL-FPF with the RPROP update rule. We tuned the ratios between r^{perf} and r^{ncount}, and between r^{ncv} and r^{cv}, respectively, using the shallowest depth (4), with only a few epochs. Essentially, the weights of FPF were modified very fast so that it reaches a region where the $ncount()$ was relatively stable afterwards (the region that in TS-FPV would be considered as a border region). The ratio between r^{perf} and r^{ncount} had to be set in such a way that the $ncount()$ in this region overlaps with the desired limit on $ncount()$. The ratio between r^{ncv} and r^{cv} was tuned in a similar way, but with r^{perf} and r^{ncount} being set to 0 (otherwise they would dominate). Moreover, the weights of the neural network were tuned using a learning set consisting of 3,000 positions selected in a similar way as the the test positions described in subsection 4.4.4.

The performance was measured and the value for the limit was set in the same way as described in subsection 4.4.4, using the same test positions. The test positions were different from the positions included in the learning set.

---

[5]This is the variable named 'depth' in CRAFTY. The parameters SD and D2L are disregarding the fractional extensions, and thus are somewhat different from ED.

4.5.2 Experimental results

In the following we describe two experiments: a preliminary experiment for selecting the most beneficial search parameters, and the main experiment for measuring the performance of the FPFs.

Selecting the search parameters

In the preliminary experiment, we tested which search parameters would be beneficial for the RL-FPF algorithm. The most beneficial search parameters proved to be *IT*, *SD* and *D2L*. The parameters *ED* (using digital encoding), *NVC* and *HASH* were also useful, although their effect was not very significant. The parameters *NCOUNT*, *SVC* and *MOSCORE* were not improving the performance at all for any of the available encodings. The parameter *BOARD* had a more special effect on the performance. It was very useful for improving the performance on the learning set, but very harmful on the test set. More precisely, it was the only search parameter that caused overfitting (and a very significant one!). Obviously, if the overfitting problem could be avoided, *BOARD* would be one of the most important and useful search parameters to use in the input. However, overfitting can be avoided only by training with a large number of positions that are sufficiently diversified. This was done for the Neural MoveMap heuristic by investigating millions of different positions. Yet, it cannot be done in this framework, since searching all these positions time and again is extremely time consuming.

We tried to overcome this problem by off-line preprocessing. First, we collected the information from the searches (using a larger number of positions). Then, we trained a neural network in a supervised way to predict whether a value change would occur in the 'current' node. This would effectively tune the weights for the third and fourth component in equation 4.13. After this preprocessing part the weights connected to *BOARD* were frozen, and RL-FPF was used to train the rest of the weights. Although this approach solved the problem of overfitting, it did not improve the performance at all. Despite of our failure to make this approach work satisfactorily, we believe that it is a promising line of future research, provided that a suitable mechanism is developed for collecting the training information for the supervised learning.

The performance of the FPFs

In the main experiment, we measured the performance of the FPFs resulting from the RL-FPF algorithm. This experiment used the search parameters *IT*, *SD*, *D2L*, *ED*, *NVC* and *HASH*. The performances of the FPFs corresponding to the different search depths are plotted in Figure 4.7. Again, the performance of the reference full-width searcher is 0. Additionally, the performances of the 1-ply-deeper searcher and the performances of the best FPV-l's are also plotted for comparison. We observed that the FPFs had superior performance to that of the FPV-l's, but was still worse than the 1-ply-deeper searcher (which is consistent with the assumptions formulated in equations 4.7 and 4.8).
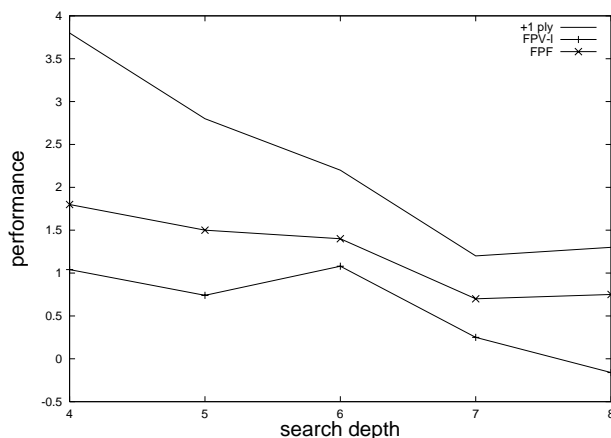
Figure 4.7: The performance of the FPF, the best FPV-l and the 1-ply-deeper searcher as a function of search depth.

4.6   Chapter conclusions

This chapter considered the problem of forward pruning, one of the instances of the class-S search decisions. Two new learning algorithms were proposed: TS-FPV and RL-FPF. TS-FPV is (one of) the first learning algorithms that is designed for discrete class-S search decisions. It uses a tabu-search algorithm to explore the space of the FPVs focusing on the most promising solutions: the critical FPVs. RL-FPF is a reinforcement-learning algorithm for tuning FPFs. The optimisation method of RL-FPF is similar to that of [14], but RL-FPF approximates the derivatives at once and does not need a cycle for each weight separately. As a consequence, the algorithm can tune an arbitrarily large number of weights without increasing significantly the learning time.

Both TS-FPV and RL-FPF rely on a partial ordering of their representation of the forward-pruning decision (and accordingly on the monotonicity assumptions). Under these assumptions both TS-FPV and RL-FPF are focusing their efforts on the most promising solutions (FPVs or FPFs), that are in the border region between the infeasible and feasible solutions. This focused behaviour gives the two algorithms some advantage over the more 'random' algorithms such as the evolutionary approach used in [77].

From the experiments we conclude that both TS-FPV and RL-FPF are able to tune a forward-pruning scheme superior to full-width search. The FPFs obtained from RL-FPF outperformed the best FPVs resulting from TS-FPV. This result is most likely due to the additional representation capabilities of the FPFs. Regarding the details of the representations for the two algorithms, we conclude the following: (1) out of the three FPV variants tested, the best is FPV-l, which uses the same value at a certain distance to the leaves; (2) for the FPFs, the most beneficial search parameters are the iteration number (IT), the search depth of the current node (SD), and the distance to the leaf ($D2L$).

Chapter 5

Time allocation

During a game a player has many times to decide which move to make. In most game programs the decision is made by a search engine. The search requires time, and in every game, a player is limited by a certain amount of time available for the whole game. The division of the (remaining) time over the moves still to be made in the game is called time allocation. The time allocated for a move often influences the choice of the move. In some positions the choice of the move is obvious, while in other positions it is more difficult to decide on which is the best move. Moreover, in some positions a move can be crucial for the outcome of the game, while in others making a slightly suboptimal move will not influence the result. Thus one might argue that it is important to use the time for those positions for which it has the most impact.

In the following we determine to which class time allocation belongs. A time-allocation mechanism is good if sufficient time is taken on the important moves, i.e. those moves that have a larger influence on the result. When a move is made, the influence of the move on the outcome of the game is not known. Thus, the way to evaluate a time-allocation mechanism is to play out the game using the mechanism and observing the outcome. Consequently, time allocation is an instance of class-G search decisions.

The chapter is organised as follows. In section 5.1, we describe the existing time-allocation techniques. In section 5.2, we introduce two algorithms for learning time allocation. Both algorithms can be applied for any class-G search decision. The experiments with time-allocation algorithms are described in section 5.3. Section 5.4 presents the conclusions.

---

[1]The author would like to thank Springer-Verlag and his co-authors for the permission of reusing relevant parts of the article in this thesis.

5.1 Existing techniques

In game programs, allocating time can be perceived as deciding on how deep
the search should be. Based on this idea, the time-allocation mechanisms can
be divided into three groups [69]:

- static mechanisms (mechanisms that decide the depth before starting the
  game);

- semi-dynamic mechanisms (mechanisms that decide the depth before the
  search);

- dynamic mechanisms (mechanisms that can change the depth during the
  search process).

In the following we will overview the time-allocation mechanisms according to
this division.

5.1.1 Static time allocation

Most game programs have the possibility to set the search depth to a fixed value.
The disadvantage of static time allocation is that it disregards anything specific
to the positions. Therefore, it is rarely used for tournament play. However, static
time allocation is frequent for experimental environments (e.g., [43]), since it is
providing stable testing conditions.

5.1.2 Semi-dynamic time allocation

Semi-dynamic time allocation received some attention from researchers studying
learning algorithms. In the following we mention two such approaches.

The first approach employs a context-sensitive classifier to recognise the class
of positions which gain from extra search [69]. The system was implemented
in the domain of checkers with some promising success. The drawback of this
approach is that it does not take into the account the influence of the move on
the outcome of the game.

The second approach uses reinforcement learning for choosing the search
depth depending on the speed of the pieces in the game of tetris [38]. The work
reported on this approach still appears to be in early stages.

5.1.3 Dynamic time allocation

Most game programs are using a hand-tuned dynamic time-allocation mecha-
nism. A major example is Hyatt's publication [48]. It considers some manually
programmed enhancements to a time-allocation mechanism on the basis of the
backed-up minimax value.

Some specific search algorithms have built-in time-allocation mechanisms.
Notable examples for this category are BPIP, Multi-ProbCut and null-move

pruning (see below). Besides these mechanisms some of the move-extension schemes can be perceived also as time-allocation mechanisms.

BPIP (Best Play for Imperfect Player) [7] employs an evaluation function that returns a probability distribution estimating the probability of various errors in valuing each position. Using the probability distribution the algorithm estimates the utility of the expansion of each leaf. It allocates extra time only if the expected gain from the extra search is higher than the "cost of time". The cost of time is a function of the remaining time and the number of moves to be made.

In Multi-ProbCut [18] a sequence of shallow searches is performed to detect whether a certain subtree will affect the minimax value of the root. If it is unlikely to affect the root value, the subtree is pruned, saving the time for probably more relevant lines.

The use of shallow searches for forward pruning appears also in adaptive null-move pruning [41]. In this case one side can make two moves in a row, and if in this way it still cannot achieve anything, this line will be pruned. The depth of the null-move search can vary with the depth of the position in the search tree or with some specific features of the position.

5.2 Learning time allocation

Time allocation is an instance of class-G search decisions. Hence time allocation is in the most general class of search decisions (see subsection 1.2.1). Below, in subsection 5.2.1 we will note that learning search decisions is a meta-level problem, and we will outline some of the properties of the meta-level problem. Then, in the following two subsections we propose two learning algorithms for class-G search decisions, which can be applied for time allocation. The first, described in subsection 5.2.2, is using the meta-level version of the actor-critic architecture. The second, described in subsection 5.2.3, is using the standard framework of a genetic algorithm.

5.2.1 The meta-level decision problem

In a game, the player to move has to select among several possible actions (the legal moves in the position). Selecting a move is termed an *object-level* decision problem. The player to 'solve' the object-level problem is searching in the space of possible continuations. During the search, it faces decisions that influence which possible continuations are investigated. These search decisions constitute a *meta-level* decision problem. The object-level search which is carried out as a consequence of a meta-level decision is termed *meta-level action.*[2]

When the player selects an object-level action (i.e., makes a move), it does so by comparing the utilities (or the approximation of the utilities) of these

---

[2]For instance, if at meta level it is decided for investigating a subtree up to a certain depth, then the search that investigates the subtree is a meta-level action. Often, a meta-level action is termed also a computational action.

actions. The utility or value of an action provides a prediction of the game result, if the action is taken. If the player is rational, it chooses the action that maximises the utility. Similarly to the object-level actions, the meta-level actions can be assigned utilities. The utility of a meta-level action derives from the time required for completing the meta-level action and the utility of the possible change of the player's intended object-level action. Again, if the player is rational, it chooses the meta-level action that maximises the utility of the meta-level action.

In the following, we outline two properties of meta-level problems. The first property of a meta-level problem is its increased complexity compared to that of the object-level problem. The size of the object-level space is the number of possible positions. The size of the meta-level space is the number of possible partial searches preceding the search decision in all possible positions. Thus the complexity of meta-level problems is usually much higher. The high complexity makes it intractable to compute the utilities of the meta-level actions. Even the currently available approximation methods are insufficient (e.g., MGSS [92] fails for search trees with conspiracy numbers [73] higher than 1). An attractive alternative to these algorithms is improving the selection between the meta-level actions through learning.

The second property of meta-level problems is that when there are more meta-level actions between two object-level actions they are approximately commutative [92] (e.g., it is not important which subtree is generated first if each of them will be investigated). This property influences the design of the learning algorithm described in subsection 5.2.2.

5.2.2   The Meta-Actor-Critic algorithm

In this subsection we introduce a value-prediction reinforcement-learning algorithm for search decisions.

At the object level, value-prediction algorithms are trying to approximate the value of a state (i.e., position) or of a state-action (i.e., the pair consisting of a position and a move in the position). If the values of the states are known, then an action is chosen by expanding the actions into subsequent states, and comparing the values of these states. In games, this is done by the usual search routines. If the values of the state-action pairs are known, the action can be selected directly by comparing these state-action values.

At the meta-level, testing each search decision is usually too time consuming. This requires the explicit computation of the state-action values [38].

When using reinforcement learning to learn the values of the search decisions, a basic idea is to reinforce the search decisions leading to a won game. The usual problem for reinforcement learning is how to distribute the credit over the decisions taken in the game. This is the first problem to solve. The second problem is the long lag before the reward, which can occur if the episode is long. The episode at the meta level has an other length than the episode at the object level (which is exactly the length of the game). The meta-level episode consists (also) of the search decisions taken, and the searches performed as a

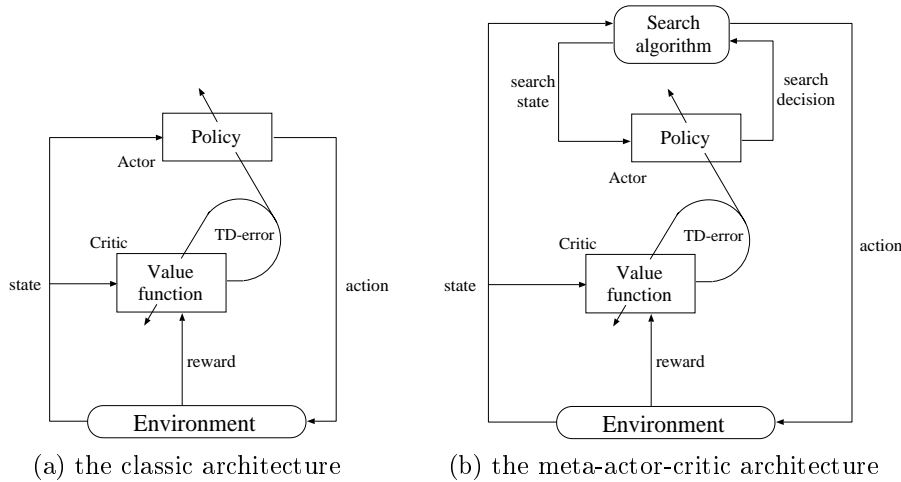(a) the classic architecture   (b) the meta-actor-critic architecture

Figure 5.1: Actor-critic architectures. In both architectures the critic learns the state-values of the object-level problem using the reward provided by its environment.

consequence. Thus, the meta-level episodes are usually longer than the object-level episodes (for some search decisions even much longer).

A solution to these two problems may come from the use of the TD error (see appendix A.1) of the object-level state value (i.e., the evaluation function) as a reward. In this case, we will reinforce those decisions that improved the position. Since the meta-level actions between two moves are approximately commutative the credit assignment can be made uniformly.[3]

The use of TD error as a reward leads us to an actor-critic architecture, where the evaluation function is the critic and the part that learns the search decisions is the actor. There are two main differences between the classic actor-critic (e.g., [63, 105, 106]) (Figure 5.1a) and the resulting meta-actor-critic architecture (Figure 5.1b). First, the state space where the actor operates can differ from the state space of the critic (usually the first includes the latter). Second, there can be more than one decision of the actor between two 'signals' from the critic. The first difference does not create a problem since the evaluation function still predicts the overall performance of the complete program. The second one is solved by the commutative nature of the meta-level actions.

To avoid the influence of the inaccuracy in the evaluation function on the final search policy we can use eligibility traces with the same eligibility factor as the discount rate of the critic [55]. The use of the eligibility also helps if the original problem is transformed into a POMDP [53].

Using the considerations above, we introduce the update rule for learning a search policy. The update rule is similar to the temporal-difference update rule

---

[3]We will investigate experimentally in subsection 5.3.2 whether the uniform credit assignment is suitable for multiple search decisions between two moves.

using eligibility traces [105]. It reads

$$\Delta v_i(t_k) = \alpha \cdot \delta(t_k) \cdot ev_i(t_k) \tag{5.1}$$

$$ev_i(t_k) = \beta \cdot ev_i(t_{k-1}) + \sum_{\tau=t_{k-1}}^{t_k} \frac{\partial}{\partial v_i}(\mathbf{y}(\tau) \cdot \mathbf{d}(\tau)) \tag{5.2}$$

where v_i denotes a weight of a neural network[4] implementing the search decision, ev_i is the associated eligibility, t_k is the time sequence of the external actions, τ denotes the moments of the search decisions, δ is the TD error[5] of the critic, α is the learning rate of the actor, β is the discount factor for the eligibility trace, $\mathbf{y}$ is the output vector, and $\mathbf{d}$ is the direction of the performed search decision associated to $\mathbf{y}$ with:

$$d_i = \begin{cases} 1, & \text{if a high value of } y_i \text{ is required} \\ -1, & \text{if a low value of } y_i \text{ is required} \\ 0, & \text{if the } i\text{th output unit does not influence the decision} \end{cases} \tag{5.3}$$

To illustrate how the values for $\mathbf{d}$ are set, let us consider a meta-level with five search decisions $(sd_1..sd_5)$ and a search state where only the first three (sd_1, sd_2, sd_3) are possible. If we encode the search decisions using five output units $(y_1..y_5)$, and assume that the choice among them is made with the use of the maximum operator, then the direction vector associated to sd_1 is $(1, -1, -1, 0, 0)$, to sd_2 is $(-1, 1, -1, 0, 0)$ and to sd_3 is $(-1, -1, 1, 0, 0)$.

The learning algorithm that uses equations 5.1 and 5.2 for updating the search policy we call the Meta-Actor-Critic (MAC) algorithm.

For illustration, the pseudo code of the algorithm for a semi-dynamic time allocation is given in Figure 5.2. The code consists of a main loop that iterates over a number of games (NG). Inside the main loop there is a second loop that iterates over the moves of one game. The game is played between two sides, viz. the side that learns a time-allocation mechanism, and an opponent. When the learning side is to move, four tasks are performed: (1) a search depth (sd) is selected (2) the eligibilities are updated, (3) the weights are updated, and (4) a move is selected using a search routine. The search depth is selected depending on the output of the neural network $(\vec{y})$. Assuming one output unit for each possible search depth, the selected depth is the one corresponding to the output unit with the highest value. Occasionally, the depth is selected random. The eligibilities are updated according to equation 5.2, with the direction vector having 1 for the selected search depth, and -1 for th rest. If the next move is not the first move to be made by the learning side, the weights are updated using equation 5.1. In the equation, the TD error of the critic is replaced by the difference between the state value of the current position, V, and the state

---

[4]The update rule is given for a neural network, but it can be used for other differentiable function approximators as well.

[5]If the critic estimates the outcome of the game and not the 'reward', then we must use the negamax value of the evaluation function.

```
initialise v⃗;
for (g = 0; g < NG; g++) {
    startgame;
    while (not gameover) {
        if (learning side to move) {
            /* select search depth */
            set the input of the neural network;
            compute y⃗;
            select sd;
            /* update eligibility */
            forall j: dⱼ = (sd == j)?1 : −1;
            ev⃗ = ev⃗ + ∂(y⃗ · d⃗)/∂v⃗;
            /* update weights */
            compute V;
            if (not first move)
                v⃗ = v⃗ + α · (prev − prevV) · ev⃗;
            prevV = V;
            /* search and move */
            move = search(sd);
            makemove(move);
        }
        else {
            move = getmove();
            makemove(move);
        }
    }
    v⃗ = v⃗ + α · (result − prevV) · ev⃗
}
```

Figure 5.2: Pseudo code of the MAC algorithm for semi-dynamic time allocation.

value of the previous position, $prevV$. The state value of the position is some function of the evaluation function.[6] The selection of the move uses the search engine of the game program with the selected depth.

If the MAC algorithm is used for dynamic time allocation, then the first two tasks of the pseudo code are inserted in the search engine at the place where the search depth is decided.

5.2.3 Genetic algorithms for time allocation

In subsection 5.2.2, we argued that a meta-level policy has to use state-action values. The approach taken to learn these values was that of value-prediction methods, that lead us to the Meta-Actor-Critic algorithm. An alternative to

---

[6]More details about state values are provided in appendix A.1. An example of function that maps the evaluation function to state values is given in equation 5.4.

value-prediction methods is to use genetic algorithms.

If we use genetic algorithms to evolve a neural network that implements the search policy, then the fitness assignment to a certain player is not different from the case of learning evaluation functions (i.e., some function of the game result).[7]   A genetic algorithm that can evolve neural networks is SANE (see appendix A.2). In the experiments, when we apply SANE for learning time allocation, we refer to it as the SANE-TA algorithm.

5.3   Experiments

This section describes the experiments with the time-allocation algorithms. In subsection 5.3.1, we compare experimentally the two learning algorithms for time allocation, viz. the MAC algorithm and the SANE-TA algorithm.   In this subsection, the two learning algorithms are compared for the case of semi-dynamic time allocation. The performance of the MAC algorithm can be influenced by the presence of multiple search decisions during the search (cf. footnote 3). Multiple search decisions appear only for dynamic time allocation. The experiments with the MAC algorithm for semi-dynamic and dynamic time allocation are described in subsection 5.3.2. Subsection 5.3.3 compares the performance of the MAC algorithm with the performance of existing time-allocation mechanisms. For illustrating the difference among time-allocation mechanisms an example is provided in subsection 5.3.4.

5.3.1   Comparison of the two learning algorithms

In this subsection, we compare experimentally the MAC algorithm and the SANE-TA algorithm in terms of performance and learning time. For the experiments, we employ the NN-population environment.

Experimental set-up

The two learning algorithms (viz. the MAC algorithm and the SANE-TA algorithm) are employed for learning a time-allocation mechanism. For each learning algorithm we measure the performance of the time-allocation mechanism and the training time.

The performance of a time-allocation mechanism is measured as the tournament performance of a test player using the time allocation. The better time-allocation mechanism results in a better tournament performance. The tournament performance of the test player (taken from the set of 100 players described in section 2.1.2) is measured in games played with both colours against the whole set (a total of 200 games). At each move three search depths are available (for decision) for the test-player: 1, 2 or 3.[8] The opponents are searching

---

[7]An example of choosing the fitness function for learning evaluation functions is described in appendix B.

[8]Time constraints on the experiments did not allow us to have a greater variety in search depths.

always to a depth of 2. The test player is constrained by the time available for a game. The time constraint is enforced not in seconds but in the number of nodes evaluated. The amount of nodes available for searching is incremented with a fix number (*incr*) after each move and diminished by the amount spent to decide on the move.[9] The value for *incr* is varied. The starting time is set to $5 \cdot incr$. If the available time is consumed, the test player is forced to move using the current search tree, without any further evaluations. The advantage of incremental timing for our experiments is that it already contains a uniform division of time per move. The task for a 'wiser' time-allocation method is to shorten the search at certain moves to save the time for later when, possibly, it is more helpful. The opponents of the test player do not have time constraints.

The training time for the two learning algorithms is measured as the number of tournaments. The time necessary for a tournament is linear in the time required for a game (or simply linear in *incr*).

For both learning algorithms the time-allocation mechanism is represented by a neural network. The input of the network consists of the position and the available time. The network has three outputs corresponding to the three search depths. The depth is decided in a winner-takes-all manner.

The time-allocation mechanism trained with the MAC algorithm is denoted *SD-MAC* (Semi-Dynamic Meta-Actor-Critic). For the learning algorithm, an additional momentum term is used in the update rule (equation 5.1). In equation 5.2 β is set to 0.9 (same as γ for the evaluation functions).

The time-allocation mechanism evolved by SANE-TA is denoted *SD-SANE* (Semi-Dynamic SANE). In the algorithm, the fitness value of a neural network is defined as the tournament performance.

Experimental results

The performances of the two time-allocation mechanisms for different values of *incr* are plotted in Figure 5.3. We observe that SD-SANE performs better then SD-MAC for almost the whole range of values of the increment.

The comparison between the time consumed by the two learning algorithms is given in Figure 5.4. In the figure, we observe that the time used by SD-SANE, is significantly higher than that used by SD-MAC. To have a more precise idea about time, the experiment for one tournament (200 games) took approximately 10 minutes, and thus the last data points for the genetic algorithm took in the order of some weeks.

5.3.2 Semi-dynamic vs. dynamic time allocation

In this subsection we describe the experiments with the MAC algorithm for a semi-dynamic and a dynamic time allocation mechanism. A dynamic mechanism is more dependent on the search algorithm. Therefore, we employ the

---

[9]The incremental timing is widely used on Internet game servers and, with the help of electronic clocks, becomes more popular in human tournaments too. Its main benefit is that it avoids losing on time if a game lasts too long.
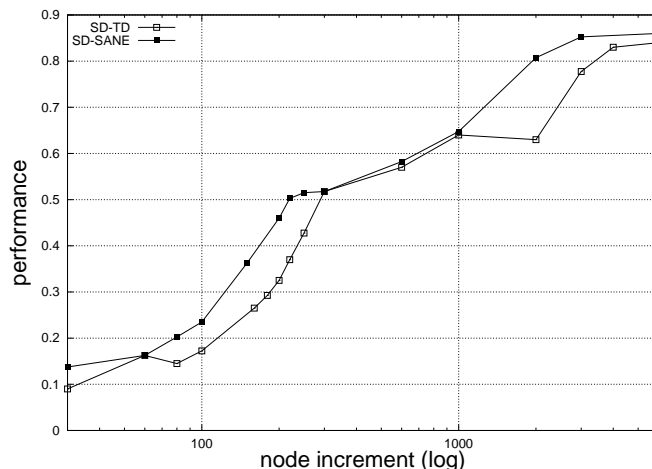
Figure 5.3: The performance of the two time-allocation mechanisms, for varying available number of nodes/move.

CRAFTY environment, since it has a superior search engine compared to the other test environments. In the semi-dynamic mechanism, SD-MAC, the search depth is decided before the search starts. In the dynamic mechanism, Dyn-MAC, after each iteration of the search it is decided whether to continue with the next iteration or to stop the search. For both mechanisms a neural network is used for the decision. The architecture of the two neural networks is explained below.

Experimental set-up

The experiments require (1) measuring the performance, (2) setting the time control, (3) choosing the architecture of the two neural networks, and (4) setting the parameters of the MAC algorithm. The first two are necessary for the evaluation of the time-allocation mechanisms, and the last two are necessary for the implementation of the time-allocation mechanisms.

Measuring the performance and the time control using the CRAFTY environment can be done in a similar way as in the NN-population environment (see subsection 5.3.1). The performance of a time-allocation mechanism is the tournament performance of the the adapted player that uses the mechanism playing against a static version of CRAFTY. To ensure variety in the games, 100 different starting positions are used. From each position both players start once with White and once with Black. Thus, the tournament consists of 200 games. The adapted player is constrained by the time available for the game, enforced in the number of nodes. As in subsection 5.3.1 incremental time is used, but since chess games are longer than LOA games, the starting time is set to $10 \cdot incr$. The static player is not constrained by time, and searches to a fixed depth, d. The
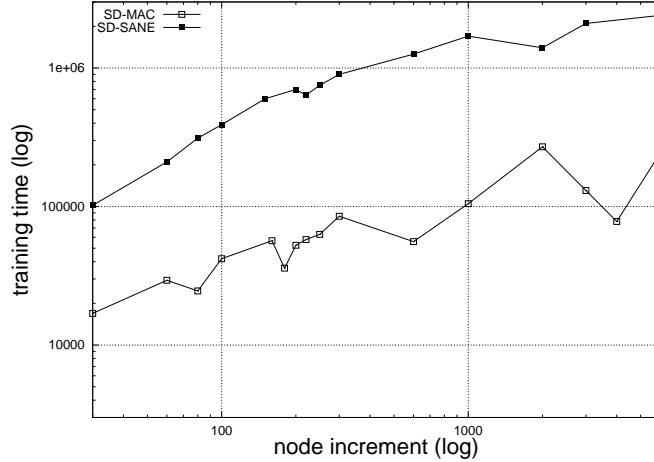
Figure 5.4: The amount of time used by the two learning algorithms to reach the performance plotted in Figure 5.7. The 'time' is measured as the number of tournaments (i.e., 200 games) multiplied by *incr*.

adapted player always completes a minimal two-ply search, and never searches deeper than $d + 1$.

Choosing the architecture of the neural networks for time allocation implies (1) selecting the input features, (2) choosing the output layer that implements the decision, and (3) setting the size of the hidden layer. For SD-MAC the most obvious input features are the board position and the remaining time. For the board position we use raw encoding. The remaining time is encoded as the number of nodes left divided by the value of the increment. For Dyn-MAC, additionally to the board position and the remaining time, we use in the input the iteration depth, the evaluation score resulting from the last two iterations and the number of nodes used so far (divided by *incr*). In SD-MAC the output represents the choice among the available search depth values. Thus each output unit corresponds to a value of the search depth (as in subsection 5.3.1). In our set-up, always three values are available. Given that the static opponent searches to depth d, the three values are: $d - 1$, d, and $d + 1$. The decision in Dyn-MAC is a binary one (i.e., to continue the search or not). Thus, we use only one output. If the value of the output is higher or equal than 0, the search proceeds with the next iteration, and if it is lower than 0, the search is stopped. For both neural networks, we use 10 hidden units.

In the MAC algorithm the neural networks are trained using equations 5.1 and 5.2. Special attention requires the TD error, δ, in the equation 5.1. In the NN-population test environment this is simply the difference between the evaluation score of two consecutive positions in the game. This was so because the evaluation functions of the NN population were trained by temporal-difference learning to approximate state values (i.e., the expected results). This is not
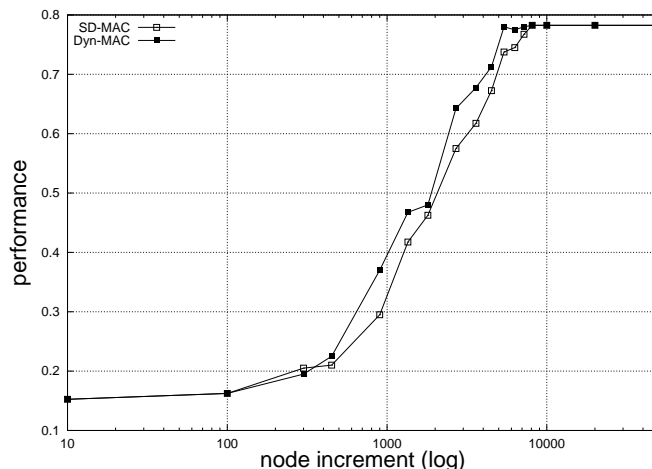
Figure 5.5: The performance of the semi-dynamic and dynamic time allocation for the MAC algorithm for depth 3.

the case with the evaluation function of CRAFTY and thus the evaluation score has to be transformed. Empirically, we noticed that using a sigmoidal function fulfils the requirement. The function is given below.

$$V(p) = \frac{2}{1 + e^{-0.05 \cdot score(p)}} - 1 \tag{5.4}$$

where $V(p)$ denotes the state value of a position p, and $score(p)$ is the evaluation score of position p returned by the search algorithm. Thus $\delta_t = V(p_{t+1}) - V(p_t)$. For β (from equation 5.2) we use the value of 0.95. Moreover, the update rule is combined with RPROP (see appendix A.3) using batch learning.

Experimental results

Using the MAC algorithm we trained a neural network for the semi-dynamic and the dynamic time allocation for two different search depths of the static opponent: 3 and 5. The resulting performance of the semi-dynamic and the dynamic time allocation corresponding to depth 3 is plotted in Figure 5.5 for various values of *incr*. Similarly, the performance corresponding to depth 5 is given in Figure 5.6. The interesting range of values for *incr* in the case of depth 3 is between 100 and 10,000. Below 100 the search will not be able to complete a minimal two-ply search. Above 10,000 it is possible to complete a four-ply search for every move. Thus outside of this range of values, the time-allocation mechanisms have little influence. Similarly, the interesting range of values for depth 5 is between 1,000 and 100,000.

We observe that the performance of the dynamic time allocation is superior to the performance of the semi-dynamic one for both search depths and most values of *incr*. Thus, we conclude that the MAC algorithm is able to use the
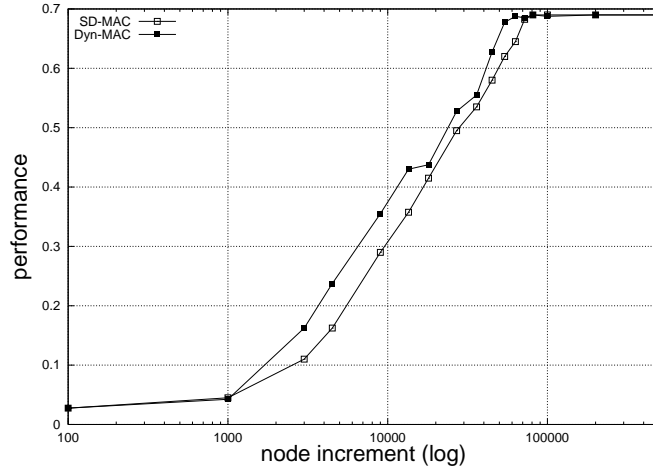
Figure 5.6: The performance of the semi-dynamic and dynamic time allocation for the MAC algorithm for depth 5.

additional freedom in the dynamic time allocation, and it is not hampered by multiple search decisions during one search process.

5.3.3 Performance of the Meta-Actor-Critic algorithm

In this subsection, we compare the time allocation resulting from the MAC algorithm to other time allocation mechanisms. In the NN-population environment, we compare the MAC algorithm to (1) a static time allocation, (2) a theoretical bound on the performance of semi-dynamic mechanisms which learn the "boards worth extra search" [69], and (3) SD-SANE. In the CRAFTY environment we compare the MAC algorithm to (1) a static time allocation, and (2) the time allocation of CRAFTY.

Experimental set-up for the NN population

The experimental set-up is roughly the same as in subsection5.3.1. Additionally to SD-MAC and SD-SANE, we measure the performance of two more time-allocation mechanisms: *static* and *SD-ES* (see below). Thus, in total we have four time allocation mechanisms.

The first mechanism is called *static* (cf. subsection 5.1.1). For each of the three possible search depths a performance is obtained. The resulting performance of the mechanism is the best performance for the three individual depths.

The second mechanism, *SD-ES* (Semi-Dynamic Extra-Search), implements a theoretical bound on the performance of semi-dynamic mechanisms which learns the "boards worth extra search" [69]. More precisely, the SD-ES searches to depth $k + n$ (instead of k) only if the extra search alters the move choice. This information is not available before the $k+n$-ply search, so it is necessary to
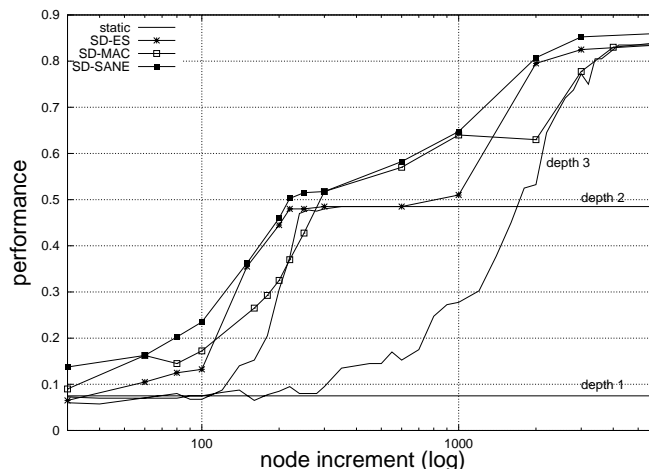
Figure 5.7: The performance of the four time-allocation mechanisms, for varying available number of nodes/move.

learn which boards will gain from extra search. For our experiments we assumed that we have an oracle. The oracle compares the move choices for both search depths. If the two choices are identical, it will suggest depth k, otherwise $k + n$. The search resource consumed by the oracle is not counted for the player. For k we use depth 1 or 2, and n is set to 1. The choice between the two values for k is made in a similar way as for the static mechanism.

The third and fourth mechanism (SD-MAC and SD-SANE) were described in subsection5.3.1.

Experimental result for the NN population

The performances of the four time-allocation mechanisms for different values of *incr* are plotted in Figure 5.7. The performance of the static mechanism should be interpreted as the maximum of the three curves (corresponding to the three possible depths). The performance of the SD-ES mechanism is represented as the combination of the performances corresponding to the two values of k.

Comparing the tournament performances of the four time-allocation mechanisms, the static one has the worst results. The relation between the performance of SD-MAC and the performance of SD-ES depends on the value of the increment. None of the two mechanisms appears to dominate the other. Finally, SD-SANE performs best over the whole range of values of the increment.

Experimental set-up for CRAFTY

The experimental set-up is roughly the same as in subsection 5.3.2. Additionally, we test two more time allocation mechanisms.
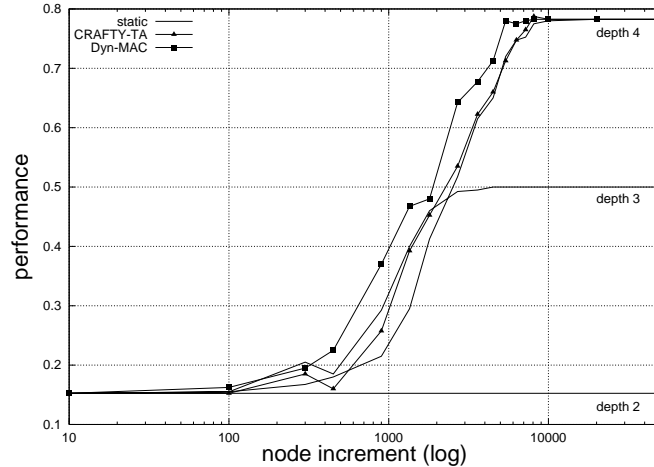
Figure 5.8: The performance of the time allocation mechanisms in CRAFTY for depth 3.

The first mechanism is called *static* (same as in subsection 5.3.1). For each of the possible search depths a performance is obtained. The resulting performance of the mechanism is the best performance for the three individual depths.

The second mechanism is the time allocation of CRAFTY. We denote this mechanism CRAFTY-TA. Roughly, it uses two main variables for stopping the search on time: *absolute_time_limit* and *time_limit*. The search is stopped either if the search used already more time than *absolute_time_limit* or it exceeded *time_limit* and the current evaluation score is not significantly worse than the last evaluation. In our implementation, we enforce the time in nodes searched. The values for the two (modified) variables is given below.

$$node\_limit = inc + rem\_nodes/35$$

$$absolute\_node\_limit = rem\_nodes$$

where *rem_nodes* is the number of remaining nodes. The time-allocation mechanism used in CRAFTY is described in more detail in [48].

Experimental results with CRAFTY

The performance of the static, the CRAFTY-TA, and the Dyn-MAC time allocation corresponding to search depth 3 is plotted in Figure 5.8 as function of the values of *incr*. The performance of the static mechanism should be interpreted as the maximum of the three curves (corresponding to the three possible depths, viz. 2, 3 and 4). The performances of the three time-allocation mechanisms corresponding to search depth 5 are plotted in Figure 5.9.

Comparing the static time allocation with the time allocation of CRAFTY, we observe that their performances are rather similar. This is most likely due to
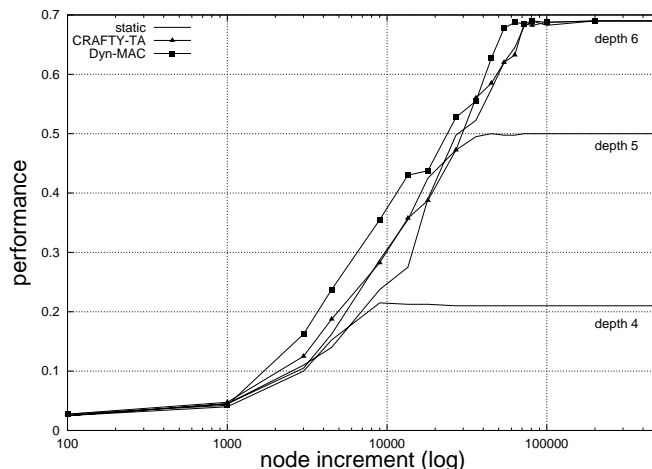
Figure 5.9: The performance of the time allocation mechanisms in CRAFTY for depth 5.

the incremental timing that favours the static time allocation. The Dyn-MAC mechanism improves on the existing time allocation in CRAFTY for almost all values of *incr* in the range of interest.

5.3.4   Example for time allocation

In this subsection we provide an example for time allocation. In the example we compare three time-allocation mechanisms: (1) static with depth 6, (2) CRAFTY-TA, and (3) Dyn-MAC (see subsection 5.3.3). We use the experimental set-up for CRAFTY described in subsection 5.3.3, with search depth 5 and increment 10,000 (cf. Figure 5.9). The player that uses one of the three time-allocation mechanisms plays White starting from the position given in Figure 5.10a. The position was reached after 10 moves. The games played by using the three mechanisms are given in Figure 5.11. Besides the moves we provide the search depth in brackets until the point where bifurcation taken place (move 12 and 21).

We observe that the static player searched on move 11 to depth 6, and left almost no time for the next move which had to be searched to a depth 2 only. This move (12. Bf4; see Figure 5.10b for the position) is most likely not a very good one, and later the game was lost by White. CRAFTY-TA and Dyn-MAC started with a shallower search than the static-depth player, and were able to make the right decision at move 12. The two followed the same sequence of moves until move 21 (Figure 5.10c). Although they made the same moves, Dyn-MAC continuously searched fewer plies than CRAFTY-TA, which made it possible to do a 6-ply deep search on move 21. The move made by Dyn-MAC (21. Kg2, with the idea of h3) appears somewhat superior to

a   b   c
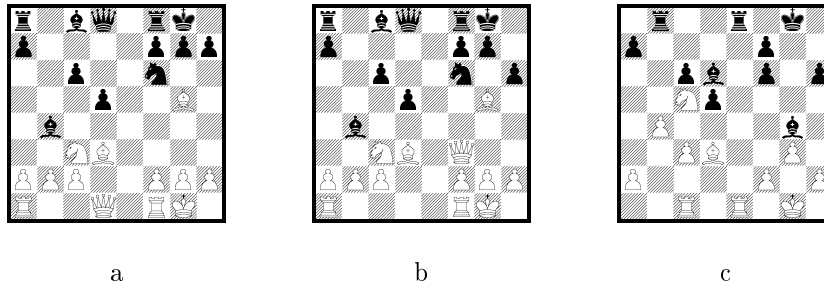
Figure 5.10: Critical positions for time allocation: a) starting position at move 11 (WTM), b) position at move 12 (WTM), c) position at move 21 (WTM).

static depth 6 – CRAFTY

11. Qf3 (**6**) h6 12. Bf4 (**2**) Bg4 13. Qg3 d4 14. Ne4 Nxe4 15. Qxg4 Nf6 16. Qf5 Bd6 17. Bd2 Qd7 18. Rfe1 Qxf5 19. Bxf5 Rab8 20. b3 Rfe8 21. f4 g6 22. Bd3 Nd5 23. Be4 Bxf4 24. Bxf4 Nxf4 25. Bxc6 Rec8 26. Be4 f5 27. Bd3 Nxd3 28. cxd3 Kf7 29. Rac1 Rc3 30. Kf2 Re8 31. Rxe8 Kxe8 32. Rxc3 dxc3 33. Ke2 h5 34. Kd1 h4 35. Kc2 h3 36. g3 Kd7 37. Kxc3 g5 38. Kd4 Kd6 39. b4 a6 40. a4 Ke6 41. b5 a5 42. Kc5 f4 43. gxf4 g4 44. f5+ Kd7 45. b6 g3 46. hxg3 h2 47. Kb5 h1=Q 48. Kxa5 Qd5+ 49. Ka6 Kc6 50. g4 Qd8 51. Ka7 Qxb6+ 52. Ka8 Qb7 0-1

CRAFTY-TA – CRAFTY

11. Qf3 (**5**) h6 12. Bxf6 (**4**) Qxf6 13. Qxf6 (**5**) gxf6 14. Na4 (**5**) Re8 15. c3 (**5**) Bd6 16. Rfe1 (**5**) Be6 17. b4 (**5**) Be5 18. g3 (**5**) Rab8 19. Rac1 (**5**) Bg4 20. Nc5 (**5**) Bd6 21. Na6 (**4**) Rbc8 22. Nc5 Rxe1+ 23. Rxe1 Be5 24. Rc1 Re8 25. a4 Bh3 26. Re1 Be6 27. Rc1 Bg4 28. f4 Bd6 29. Kf2 Bxc5+ 30. bxc5 Kf8 31. Re1 Rxe1 32. Kxe1 Ke7 33. a5 h5 34. Kd2 Ke6 35. Ke3 Bf5 36. c4 Bxd3 37. Kxd3 dxc4+ 38. Kxc4 a6 39. Kd4 f5 40. Kc4 Ke7 41. Kd4 f6 42. Kd3 Kd8 43. Kc4 Kd7 44. Kc3 Kd8 45. h3 Ke7 46. Kc4 Kd7 47. Kd3 Ke7 48. Ke3 Kd7 49. Kd3 Ke7 50. Kc4 Kd7 51. Kc3 Ke7 52. Kd3 Kf7 53. Kc4 Ke7 54. Kd3 Kf7 55. Kc4 Ke7 56. Kc3 Kf7 57. h4 Ke7 58. Kd3 Ke6 59. Kd4 Kd7 60. Kc3 Ke6 61. Kc4 Ke7 62. Kd3 Kd7 63. Kc3 Ke6 64. Kd4 Kd7 65. Kd3 Kc7 66. Kc3 Kd7 67. Kd3 Kc7 68. Kc4 Kd7 69. Kb3 Kc8 70. Kc4 Kd7 71. Kd4 Ke6 72. Kc4 Kd7 1/2-1/2

Dyn-MAC – CRAFTY

11. Qf3 (**4**) h6 12. Bxf6 (**4**) Qxf6 13. Qxf6 (**4**) gxf6 14. Na4 (**5**) Re8 15. c3 (**4**) Bd6 16. Rfe1 (**4**) Be6 17. b4 (**4**) Be5 18. g3 (**4**) Rab8 19. Rac1 (**4**) Bg4 20. Nc5 (**4**) Bd6 21. Kg2 (**6**) Bxc5 22. bxc5 Kg7 23. Rxe8 Rxe8 24. Rb1 Be2 25. Bf5 Re5 26. g4 h5 27. h3 Re7 28. Kg3 hxg4 29. hxg4 Bc4 30. a3 Re2 31. Rb7 a5 32. Ra7 Re1 33. Rxa5 Rc1 34. Bd7 Rxc3+ 35. Kg2 d4 36. Bxc6 d3 37. Ra8 f5 38. gxf5 d2 39. Rd8 Rd3 40. Rxd3 Bxd3 41. Ba4 Kf6 42. Kf3 Bb5 43. Bd1 Kxf5 44. Ke3 Ke5 45. Kxd2 Kd4 46. a4 Bc6 47. a5 Bb7 48. Be2 Kxc5 49. a6 Bc6 50. Ke3 Bd5 51. Bd3 Bc6 52. f4 Bg2 53. Be4 Bxe4 54. Kxe4 Kb6 55. Ke5 Ka7 56. f5 Kxa6 57. Kf6 Kb5 58. Kxf7 Kc4 59. Ke6 Kd3 60. Ke5 Ke3 61. f6 1-0

Figure 5.11: Example games for time allocation.

the move made by CRAFTY-TA (21. Na6 with Nc5 back on the next move).
Dyn-MAC eventually won the game, while CRAFTY-TA drew. Of course, the
moves emphasised above are not the only (important) decisions for the results.
We understand that the outcome of the games are influenced by other moves
as well, and also by the moves of the opponent. Nevertheless, the example
provides a clear understanding of the difficulties that an intelligent program
may face when deciding on a search decision.

5.4   Chapter conclusions

This chapter considered time allocation, one of the instances of the class-G
search decisions. We proposed two learning approaches for time allocation.
First, we introduced a new learning algorithm for class-G search decisions, the
MAC algorithm. The algorithm uses a modified actor-critic architecture that
can be used for search decisions. Second, we suggested a genetic algorithm,
SANE-TA, suitable for learning time allocation.

From the experiments comparing the two learning algorithms, we conclude
that with the SANE-TA algorithm a better time-allocation mechanism can be
obtained than with the MAC algorithm, but the time required for training by
the former is significantly higher, and even prohibitive. From the experiments
comparing semi-dynamic and dynamic time allocation, we conclude that the
MAC algorithm is suitable for dynamic time allocation, and is not hampered
by multiple search decisions during one search process. Although the rather
shallow search depths used in the experiments do not allow us to have a definite
conclusion on the performance, we observe that the time-allocation mechanism
resulting from the two learning algorithms achieved comparable or better results
than the existing time-allocation mechanisms investigated (including a well-
tuned time-allocation mechanism).

Chapter 6

Conclusions and future research

In this thesis we addressed the research problem of improving search decisions by learning algorithms. In subsection 1.2.1, we introduced a classification of search decisions in class-P, class-S, and class-G search decisions. Subsequently, we chose an instance for each class: move ordering for class-P search decisions, forward pruning for class-S search decisions, and time allocation for class-S search decisions. The research question formulated in section 1.4 was addressed by developing learning algorithms for each of the three instances. The experimental results showed that the learning algorithms are capable of improving the particular instance of the search decisions, and are competitive with the existing algorithms for the search decisions.

The main conclusions for the three instances of search decisions were given in the sections 3.5, 4.6, and 5.4. In this chapter we summarise these conclusions (in section 6.1 to section 6.3) and provide promising directions for future research (in section 6.4).

6.1 Move ordering

We noted that move ordering is the main instance of the class-P search decisions. We developed a new move ordering heuristic, the Neural MoveMap heuristic. The heuristic uses a neural network to estimate the likelihood of a move being the best in a certain position. The moves considered more likely to be the best are examined first.

Concerning the details of the heuristic, we concluded that the best choice to encode the moves in the neural network is to use one output unit for each possible move of the game. The neural network should be trained with large learning sets. The learning sets can be labelled either with the game program or using the moves from a game database. The best approach to use the neural network during the search is the weighted-combination approach, which orders

the move according to a weighted sum of the neural-network and the history-heuristic score of the respective moves.

We have shown that the neural network predicts the best move in chess almost as often as a 7-ply search of CRAFTY, and more important, by using the Neural MoveMap heuristic, the search expands smaller trees than the existing heuristics (specifically the history heuristic) both in the games of LOA and of chess.

6.2   Forward pruning

Forward pruning is an instance of class-S search decisions. We defined two types of representations for forward pruning: (1) FPVs, which is a discrete representation, and (2) FPFs, which is a continuous representation. We developed two new learning algorithms, designed for maximising the quality of the move selected by the search while keeping the size of the search tree below a specified limit: the TS-FPV algorithm and the RL-FPF algorithm. The TS-FPV algorithm uses a tabu-search algorithm to explore the space of the FPVs focusing on the most promising solutions: the critical FPVs. The RL-FPF algorithm is a reinforcement-learning algorithm for tuning FPFs.

The experimental results showed that the two algorithms are able to tune a forward-pruning scheme that has a better overall performance than the full-width search. The FPFs obtained from RL-FPF outperformed the best FPVs resulting from TS-FPV.

6.3   Time allocation

We remarked that time allocation is a class-G search decision. We proposed two learning algorithms for time allocation, viz. the Meta-Actor-Critic algorithm and the SANE-TA algorithm. The Meta-Actor-Critic algorithm (MAC) is a gradient-descent actor-critic algorithm. The SANE-TA algorithm is an application of a genetic algorithm, SANE.

We concluded that SANE-TA algorithm resulted in a better time-allocation scheme than the MAC algorithm, but the time required for training by the former is significantly higher, and even prohibitive. Both algorithms resulted in a comparable or better time allocation than the existing time allocation algorithms tested in our experiments. Finally, we concluded that the Meta-Actor-Critic algorithm is able to use the additional freedom in the dynamic time allocation, and it is not hampered by multiple search decisions during one search process.

6.4   Future research

There are two main directions to extend the work described in this thesis: (1) improving the learning algorithms developed, and (2) extending the learning

algorithms developed so far to other instances of the same class of search decisions or even to an other class. Below we discuss both directions for each of the learning algorithms proposed in the thesis.

Improving the Neural MoveMap heuristic. Most of the details of the heuristic (subsection 3.2.1 to subsection 3.2.3) were studied already, and these details were well tuned experimentally. Nevertheless, there are a two modifications that may be beneficial for the heuristic. The first consists of using more knowledgeable features instead of (or additionally to) encoding the raw board position in the input of the neural network. The features of the evaluation function are reasonable candidates for this purpose. The second modification is related to the learning algorithm. As it was described in the thesis, the heuristic relies on off-line training. A combination of off-line and on-line learning may benefit both from the general knowledge of the game and from the specifics of the position on the board. In the heuristic, the weighted-combination approach is already adding on-line knowledge (resulting from the history heuristic scores) to the off-line knowledge of the neural network, but further training the neural network during the search may be a better way to incorporate on-line knowledge in the heuristic.

Extending the Neural MoveMap heuristic. It is unlikely that the heuristic can be used for search decisions other than move ordering.

Improving the TS-FPV algorithm. The algorithm may benefit from extending the FPV variants. All three variants investigated were rather simple, and by designing more complex ones the result should be beneficial for the forward-pruning algorithm. The disadvantage that may result from the more complex FPV variants is the increase of the cardinality of FPV space, and as a consequence the increase in time in the optimisation phase. The increase in time can be diminished if faster ways to evaluate the performance of FPVs are developed.

Extending the TS-FPV algorithm. The algorithm is easy to be modified for other class-S search decisions with a discrete representation. It would be interesting to test the algorithm for choosing the sequence of search depths in iterative deepening or some of the extension schemes.

Improving the RL-FPF algorithm. The first source for improvement can be the use of additional search parameters. This was discussed in more detail in subsection 4.5.2. A second source could be the faster evaluation of the performance of the FPFs.

Extending the RL-FPF algorithm. Modifying the algorithm for class-S search decisions with continuous representation is possible, but not so straightforward as modifying the TS-FPV algorithm. This is because, the algorithm depends partially on reward terms that are dependent on the search decisions. For every search decision these reward terms have to be designed. Nevertheless,

modifying the algorithm for learning search extensions is a promising direction for further research. The RL-FPF algorithm can be modified for class-P search decisions, such as move ordering. However, tuning the algorithm for move ordering in such a way that it results in a move ordering competitive with the existing techniques is a rather difficult task.

Improving the Meta-Actor-Critic algorithm. Since the class-G search decisions have to be evaluated on complete move sequences, the time required by the learning algorithms for this class is usually high. Although, the Meta-Actor-Critic algorithm is faster than the genetic algorithms, the training time is a serious problem. Similarly as for the forward pruning algorithms, speed up of the algorithm can result only from shortening the time used for evaluation.

Extending the Meta-Actor-Critic algorithm. The algorithm may be extended for any search decision with continuous representation, independently of the class it belongs to. However, there are two difficulties. First, the algorithm should be improved with the specifics of the search decision. Second, the algorithm should be sped up. Despite of these difficulties, it would be challenging to modify the Meta-Actor-Critic algorithm for learning forward pruning or search extensions.

Improving the SANE-TA algorithm. The SANE algorithm is continuously improved by the authors of the algorithm. The improvement of the SANE-TA algorithm can result either from the improvement of the SANE algorithm, or an improvement in the way SANE is applied for time allocation. The latter can be achieved by speeding up the evaluation.

Extending the SANE-TA algorithm. The SANE algorithm is a general purpose genetic algorithm, and thus extending SANE-TA for other search decisions is rather simple. SANE was applied already for forward pruning in the game of Othello [77].

References

[1] 2000 FIDE handbook, E.I. Laws of chess, http://handbook.fide.com/.

[2] S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *1977 ACM Annual Conference Proceedings*, pages 466–473. ACM, Seattle, 1977.

[3] R. Aler, D. Borrajo, and P. Isasi. Using genetic programming to learn and improve control knowledge. *Artificial Intelligence*, 141:29–56, 2002.

[4] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. thesis, University of Limburg, The Netherlands, 1994.

[5] T.S. Anantharaman. Extension heuristics. *ICCA Journal*, 14(2):47–65, 1991.

[6] T.S. Anantharaman. Evaluation tuning for computer chess: Linear discriminant methods. *ICCA Journal*, 20(4):224–242, 1997.

[7] E.B. Baum and W.D. Smith. A bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1-2):195–242, 1997.

[8] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *ICCA Journal*, 21(2):84–99, 1998.

[9] D.F. Beal and M.C. Smith. Learning piece values using temporal difference learning. *ICCA Journal*, 20(3):147–151, 1997.

[10] A. Bernstein, M. de V. Roberts, T. Arbuckle, and M. S. Belsky. A chess playing program for the IBM 704. In *Proceedings of the 1958 Western Joint Computer Conference*, pages 157–159. Los Angeles, California, 1958.

[11] Y. Björnsson. *Selective Depth-First Game-Tree Search.* Ph.D. thesis, University of Alberta, Alberta, Canada, 2002.

[12] Y. Björnsson and T.A. Marsland. Multi-cut pruning in alpha-beta search. In H.J. van den Herik and H. Iida, editors, *Computers and Games, Proceedings of the 1st International Conference CG'98, Lecture Notes in Computers Science 1558*, pages 15–24. Springer-Verlag, Berlin, 1999.

[13] Y. Björnsson and T.A. Marsland. Learning search control in adversary games. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 157–174. IKAT, Maastricht, The Netherlands, 2001.

[14] Y. Björnsson and T.A. Marsland. Learning control of search extensions. In *Proceedings of the 6th Joint Conference on Information Sciences (JCIS 2002)*, pages 446–449, 2002.

[15] Y. Björnsson and M.H.M. Winands. YL wins Lines of Action tournament. *ICGA Journal*, 24(3):180–181, 2001.

[16] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes and two-level tables. *ICCA Journal*, 19(3):175–180, 1996.

[17] M. Buro. Statistical feature combination for the evaluation of game positions. *Journal of Artificial Intelligence Research*, 3:373–382, 1995.

[18] M. Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In H. J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 77–96. Universiteit Maastricht, The Netherlands, 1999.

[19] M. Buro. Toward opening book learning. *ICCA Journal*, 22(2):98–102, 1999.

[20] M.S. Campbell. Knowledge discovery in Deep Blue. *Communications of the ACM*, 42(11):65–67, November 1999.

[21] K. Chellapilla and D.B. Fogel. Co-evolving checkers playing programs using only win, lose, or draw. In *Proceedings of SPIE's AeroSense'99: Applications and Science of Computational Intelligence II*, 1999.

[22] S. Chinchalkar. An upper bound for the number of reachable positions. *ICCA Journal*, 19(4):181–183, 1996.

[23] F.A. Dahl. Honte, a go-playing program using neural nets. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 10, pages 205–223. Nova Science Publishers, 2001.

[24] C. Donninger. Null move and deep search: Selective-search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, 1993.

[25] D. Dyer. Lines of action homepage. http://www.andromeda.com/people /ddyer/loa/loa.html.

[26] H.D. Enderton. The Golem Go program. Technical Report CMU-CS-92-101, School of Computer Science, Carnegie-Mellon University, 1991.

[27] S.L. Epstein. Learning to play expertly: A tutorial on Hoyle. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 8, pages 153–178. Nova Scientific Publishers, 2001.

[28] T.A. Estlin and R.J. Mooney. Learning to improve both efficiency and quality for planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1227–1233. Morgan Kaufmann, San Mateo, CA, 1997.

[29] T.E. Fawcett and P.E. Utgoff. Automatic feature generation for problem solving systems. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Conference on Machine Learning*, pages 144–153. Morgan Kaufmann, San Mateo, CA, 1992.

[30] P.W. Frey. Algorithmic strategies for improving the performance of game playing programs. *Physica D*, 22:355–365, 1986.

[31] J. Fürnkranz. Machine learning in games: A survey. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 2, pages 11–59. Nova Scientific Publishers, 2001.

[32] R. Gasser. Applying retrograde analysis to nine men's morris. In D.N.L. Levy and D.B. Beal, editors, *Heuristic Programming in Artificial Intelligence 2: the Second Computer Olympiad*, pages 161–173. Ellis Horwood Ltd., Chichester, UK, 1991.

[33] F. Glover. Tabu search – Part I. *ORSA J. of Computing*, 1(3):190–206, 1989.

[34] K.R.C. Greer. Computer chess move-ordering schemes using move influence. *Artificial Intelligence*, 120:235–250, 2000.

[35] K.R.C. Greer, P.C. Ojha, and D.A. Bell. A pattern-oriented approach to move ordering: the chessmaps heuristic. *ICCA Journal*, 22(1):13–21, 1999.

[36] R. Grimbergen. The 13th CSA world computer-shogi championship. *ICGA Journal*, 26(2):120–124, 2003.

[37] S. Hanafi and A. Freville. An efficient tabu search approach for the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 41(106):659–675, 1998.

[38] D. Harada and S. Russell. Extended abstract: Learning search strategies. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*. Stanford University, Palo Alto, California, 1999.

[39] G. Haworth and M. Velliste. Chess endgames and neural networks. *ICCA Journal*, 21(4):211–227, 1998.

[40] E.A. Heinz. Extended futility pruning. *ICCA Journal*, 21(2):75–83, 1998.

[41] E.A. Heinz. Adaptive null-move pruning. *ICCA Journal*, 22(3):123–132, 1999.

[42] E.A. Heinz. New self-play results in computer chess. In T.A. Marsland and
     I. Frank, editors, *Computers and Games, Proceedings of the 2nd Interna-
     tional Conference CG 2000, Lecture Notes in Computers Science 2063*,
     pages 262–276. Springer-Verlag, Berlin, 2001.

[43] E.A. Heinz. Self-play experiments in computer chess revisited. In H.J.
     van den Herik and B. Monien, editors, *Advances in Computer Games 9*,
     pages 73–91. IKAT, Maastricht, The Netherlands, 2001.

[44] H.J. van den Herik. *Computerschaak, Schaakwereld en Kunstmatige In-
     telligentie*. Ph.D. thesis, Delft University of Technology, The Netherlands,
     1983.

[45] H.J. van den Herik and I.S. Herschberg. Omniscience, the rulegiver? In
     B. Pernici and M. Somalvico, editors, *Proceedings of III Covegno Inter-
     nazionale L'Intelligenza Artificiale ed il Gioco Degli Scacchi*, pages 1–18,
     1986.

[46] H.J. van den Herik, J.W.H.M. Uiterwijk, and J. van Rijswijck. Games
     solved: Now and in the future. *Artificial Intelligence*, 134(1–2):277–311,
     2002.

[47] J.M. Hsu. A strategic game program that learns from mistakes. M.Sc.
     thesis, Northwestern University, Evanston, IL, 1985.

[48] R.M. Hyatt. Using time wisely. *ICCA Journal*, 7(1):4–9, 1984.

[49] R.M. Hyatt. Book learning – a methodology to tune an opening book
     automatically. *ICCA Journal*, 22(1):3–12, 1999.

[50] R.M. Hyatt. Rotated bitmaps, a new twist on an old idea. *ICCA Journal*,
     22(4):213–222, 1999.

[51] R.M. Hyatt and M. Newborn. CRAFTY goes deep. *ICCA Journal*,
     20(2):79–86, 1997.

[52] N. Inuzuka, H. Fujimoto, T. Nakano, and H. Itoh. Pruning nodes in the
     alpha-beta method using inductive logic programming. In J. Fürnkranz
     and M. Kubat, editors, *Workshop Notes: Machine Learning in Game
     Playing*. 16th International Conference on Machine Learning (ICML-99),
     1999.

[53] T. Jaakkola, S. Singh, and M. Jordan. Reinforcement learning algorithm
     for partially observable markov problems. In *Advances in Neural Infor-
     mation Processing Systems 7*, pages 345–352, 1994.

[54] Y.-F. Ke and T.-M. Parng. The guard heuristic: Legal move ordering
     with forward game-tree pruning. *ICCA Journal*, 16(2):76–85, 1993.

[55] H. Kimura and S. Kobayashi. An analysis of actor/critic algorithms using eligibility traces: Reinforcement learning with imperfect value function. In *Proceedings of the 15th International Conference on Machine Learning*, pages 278–286, 1998.

[56] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[57] L. Kocsis, H.J. van den Herik, and J.W.H.M. Uiterwijk. Two learning algorithms for forward pruning. *ICGA Journal*, 26(3):165–181, 2003.

[58] L. Kocsis and J.W.H.M. Uiterwijk. Learning move ordering in chess. In J.W.H.M. Uiterwijk, editor, *Proceedings of the 6th Computer Olympiad Computer-Games Workshop*. Technical Report CS 01-04, IKAT, Maastricht, The Netherlands, 2001.

[59] L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Learning time allocation using neural networks. In T.A. Marsland and I. Frank, editors, *Computers and Games, Proceedings of the 2nd International Conference CG 2000, Lecture Notes in Computers Science 2063*, pages 170–185. Springer-Verlag, Berlin, 2001.

[60] L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Move ordering using neural networks. In L. Monostori, J. Váncza, and M. Ali, editors, *Engineering of Intelligent Systems, Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2001), Lecture Notes in Artificial Intelligence, Vol. 2070*, pages 45–50. Springer-Verlag, Berlin, 2001.

[61] L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Search-independent forward pruning. In B. Kröse, M. de Rijke, G. Schreiber, and M. van Someren, editors, *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*, pages 159–166, 2001.

[62] L. Kocsis, J.W.H.M. Uiterwijk, E.O. Postma, and H.J. van den Herik. The Neural MoveMap heuristic in chess. In *Proceedings of the 3nd International Conference on Computers and Games (CG 2002), To appear in the "Lecture Notes in Computer Science" series*, 2002.

[63] V.R. Konda and J.N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems 12*, 2000.

[64] K.F. Lee and S. Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36(1):1–25, 1988.

[65] R.A Levinson. A self-learning, pattern-oriented chess program. *ICCA Journal*, 12(4):207–215, 1989.

[66] D. Levy, D. Broughton, and M. Taylor. The SEX algorithm in computer chess. *ICCA Journal*, 12(1):10–21, 1989.

[67] T.R. Lincke. Strategies for the automatic construction of opening books. In T.A. Marsland and I. Frank, editors, *Computers and Games, Proceedings of the 2nd International Conference CG 2000, Lecture Notes in Computers Science 2063*, pages 74–86. Springer-Verlag, Berlin, 2001.

[68] A. Lokketangen and F. Glover. Solving zero-one mixed integer programming problems using tabu search. *European Journal of Operational Research*, 41(106):624–658, 1998.

[69] S. Markovitch and Y. Sella. Learning of resource allocation strategies for game playing. *Computational Intelligence*, 12(1):88–105, 1996.

[70] T.A. Marsland. A review of game-tree pruning. *ICCA Journal*, 9(1):3–19, 1986.

[71] T.A. Marsland and M.S. Campbell. Parallel search on strongly ordered game trees. *Computing Surveys*, 14(4):533–551, 1982.

[72] A. Matanović. *Encyclopaedia of Chess Openings*, volume A–E. Chess Informant, Beograd.

[73] D.A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, 1988.

[74] S. Minton. *Learning Search Control Knowledge*. Kluwer Academic Publishers, 1988.

[75] T.M. Mitchell, R.M. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[76] E. Morales. Learning patterns for playing strategies. *ICCA Journal*, 17(1):15–26, 1994.

[77] D.E. Moriarty and R. Miikkulainen. Evolving neural networks to focus minimax search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1371–1377, 1994.

[78] D.E. Moriarty and R. Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–209, 1995.

[79] D.E. Moriarty and R. Miikkulainen. Learning sequential decision tasks. Technical Report AI95-229, Department of Computer Sciences, University of Texas, 1995.

[80] D.E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1–3):11–32, 1996.

[81] D.E. Moriarty and R. Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, 1997.

[82] D.E. Moriarty and R. Miikkulainen. Hierarchical evolution of neural networks. In *Proceedings of the 1998 IEEE Conference on Evolutionary Computation*, pages 428–433, 1998.

[83] S.H. Muggleton. *Inductive Logic Programming*. Academic Press, New York, NY, 1992.

[84] E.V. Nalimov, G.M^{c}C. Haworth, and E.A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000.

[85] M.A. Pérez. Representing and learning quality-improving search control knowledge. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 382–390. Morgan Kaufmann, San Francisco, CA, 1996.

[86] D. Pisinger. *Algorithms for Knapsack Problems*. Ph.D. thesis, University of Copenhagen, Copenhaga, Denmark, 1995.

[87] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1–2):255–293, 1996.

[88] M. Riedmiller. Advanced supervised learning in multilayer perceptrons - from backpropagation to adaptive learning algorithms. *Int. Journal of Computer Standards and Interfaces, Special Isuue on Neural Networks*, 16:265–278, 1994.

[89] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In H. Rusini, editor, *Proceedings of the IEEE International Conference on Neural Networks 1993 (ICNN 93)*, pages 586–591, 1993.

[90] M. Riedmiller and H. Braun. RPROP – description and implementation details. Technical report, Universitat Karlsruhe, 1994.

[91] J. van Rijswijck. Learning from perfection (a data mining approach to evaluation function learning in awari). In T.A. Marsland and I. Frank, editors, *Computers and Games, Proceedings of the 2nd International Conference CG 2000, Lecture Notes in Computers Science 2063*, pages 115–132. Springer-Verlag, Berlin, 2001.

[92] S. Russell and E.H. Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.

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

[94] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

[95] J. Schaeffer. The history heuristic. *ICCA Journal*, 6(3):16–19, 1983.

[96] J. Schaeffer. The history heuristic and alpha-beta search enhancements in
      practice. *IEEE Pattern Analysis and Machine Intelligence*, 11(11):1203–
      1212, 1989.

[97] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy at Check-
      ers*. Springer-Verlag, New York, NY, 1997.

[98] J. Schaeffer. The games computers (and people) play. *Advances in Com-
      puters*, 50:189–266, 2000.

[99] J. Schaeffer and A. Plaat. Kasparov versus DEEP BLUE: The rematch.
      ICCA Journal, 20(2):95–101, 1997.

[100] T. Scherzer, L. Scherzer, and D. Tjaden. Learning in Bebe. In T. A.
      Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*,
      chapter 12, pages 197–216. Springer-Verlag, New York, NY, 1990.

[101] Y. Seirawan. The Kasparov - DEEP BLUE games. *ICCA Journal*,
      20(2):102–125, 1997.

[102] B. Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*,
      134(1–2):241–275, 2002.

[103] M. Slate and L. Atkin. Chess 4.5 – The Northwestern University chess
      program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages
      82–118. Springer-Verlag, New York, NY, 1977.

[104] S.J.J. Smith and D.S. Nau. An analysis of forward pruning. In *Proceedings
      of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*,
      volume 2, pages 1386–1391, 1994.

[105] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*.
      MIT Press, 1998.

[106] R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient
      methods for reinforcement learning with function approximation. In *Ad-
      vances in Neural Information Processing Systems 12*, pages 1057–1063,
      2000.

[107] G. Tesauro. Connectionist learning of expert preferences by comparison
      training. In D. Touretzky, editor, *Advances in Neural Information Process-
      ing Systems 1 (NIPS-88)*, pages 99–106. Morgan Kaufmann, San Mateo,
      CA, 1989.

[108] G.J. Tesauro. Practical issues in temporal difference learning. *Machine
      Learning*, 8:257–277, 1992.

[109] K. Tournavitis. MOUSE(μ): A self-teaching algorithm that achieved
      master-strength at othello. In *Proceedings of the 3nd International Con-
      ference on Computers and Games (CG 2002), To appear in the "Lecture
      Notes in Computer Science" series*, 2002.

[110] Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):145–152, 2002.

[111] Y. Tsuruoka, D. Yokoyama, T. Maruyama, and T. Chikayama. Game-tree search algorithm based on realization probabilty. In *Proceedings of the 6th Game Programming Workshop 2001*, pages 17–24, 2001.

[112] J.W.H.M. Uiterwijk. The countermove heuristic. *ICCA Journal*, 15(1):8–15, 1992.

[113] P.E. Utgoff and J. Clouse. Two kinds of training information for evaluation function learning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 596–600, 1991.

[114] P.E. Utgoff and P.S. Heitman. Learning and generalizing move selection preferences. In H. Berliner, editor, *Proceedings of the AAAI Spring Symposium on Computer Game Playing*, pages 36–40, Stanford University, 1988.

[115] P.E. Utgoff and D. Precup. Constructive function approximation. In H. Liu and H. Motoda, editors, *Feature Extraction, Construction and Selection: A Data Mining Perspective*, volume 453 of *The Kluwer International Series in Engineering and Computer Science*, chapter 14. Kluwer Academic Publishers, 1998.

[116] E.C.D. van der Werf, J.W.H.M. Uiterwijk, E.O. Postma, and H.J. van den Herik. Local move prediction in Go. In *Proceedings of the 3rd International Conference on Computers and Games (CG 2002), To appear in the "Lecture Notes in Computer Science" series*, 2002.

[117] D.E Wilkins. Using knowledge to control tree searching. *Artificial Intelligence*, 18(1):1–51, 1982.

[118] M.H.M. Winands. Analysis and implementation of Lines of Action. M.Sc. thesis, Department of Computer Science, Universiteit Maastricht, The Netherlands, 2000.

[119] M.H.M. Winands, H.J. van den Herik, and J.W.H.M. Uiterwijk. An evaluation function for Lines of Action. In H.J. van den Herik and H. Iida, editors, *Advances in Computer Games 10 (to appear)*, 2003.

[120] M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Learning in Lines of Action. In J.W.H.M. Uiterwijk, editor, *Proceedings of the 7th Computer Olympiad Computer-Games Workshop*. Technical Report CS 02-03, IKAT, Maastricht, The Netherlands, 2002.

[121] M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Learning in Lines of Action. In H. Blockeel and M. Denecker, editors, *Proceedings of the 14th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2002)*, pages 371–378, 2002.

[122] M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Temporal difference learning and the Neural MoveMap heuristic in the game of Lines of Action. In Q. Mehdi, N. Gough, and M. Cavazza, editors, *Proceedings of the 3rd International Conference on Intelligent Games and Simulation (GAME-ON 2002)*, pages 99–103, 2002.

[123] M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. Combining proof-number search with alpha-beta search. In B. Kröse, M. de Rijke, G. Schreiber, and M. van Someren, editors, *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*, pages 299–306, 2001.

[124] M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. The quad heuristic in Lines of Action. *ICGA Journal*, 24(1):3–15, 2001.

[125] T. Wolf. Forward pruning and other heuristic search techniques in tsumo go. *Information Sciences*, 122:59–76, 2000.

Appendix A

Learning algorithms

Below we provide three learning algorithms. Appendix A.1 describes temporal-difference learning, appendix A.2 describes SANE, and appendix A.3 describes RPROP.

A.1 Temporal-difference learning

An attractive approach to tune an evaluation function is temporal-difference (TD) learning (see, e.g., [105]). In TD learning, a learning agent interacts with its environment to achieve a goal. The agent senses the state of the environment, and performs actions that affect that state. In the domain of games, the state is represented by a position, and the actions by moves. The goal of the agent (i.e., player) is to win the game. Each state s has associated a value $V(s)$, representing the estimation of the likelihood of winning the game from that state. In a game program, the state value can be used as an evaluation function in the search tree.

In the learning phase, the state values are updated so that they approach a target value. Let us consider a sequence of game positions $s_0, s_1, \ldots, s_T$. The target value for the final position, s_T, is given by

$$V^{target}(s_T) = \begin{cases} 1, & \text{if } s_T \text{ is a win for White} \\ 0, & \text{if } s_T \text{ is a draw} \\ -1, & \text{if } s_T \text{ is a win for Black} \end{cases} \qquad (A.1)$$

The target values for the non-terminal positions $s_0, s_1, \ldots, s_{T-1}$ are given by

$$V^{target}(s_t) = \gamma V(s_{t+1}) \qquad (\text{with } 0 < \gamma \leq 1) \qquad (A.2)$$

The *discount factor* γ is used to favour early over late success.

Occasionally, TD learning can be slow. To speed up the learning, we can

use TD(λ), which averages toward future target values:

$$V^{target}(s_t) = (1 - \lambda) \sum_{k=1}^{T-t-1} \lambda^{k-1} V(s_{t+k}) + \lambda^{T-t-1} V(s_T) \qquad (\text{with } 0 \leq \lambda \leq 1)$$

$$(A.3)$$

In equation A.3, for simplicity we omitted the discount factor γ. The precise combination of γ and λ is described in [105].

In game programs using TD learning, V is typically represented by a parametrised function. To tune the weights of this function, we minimise the mean square of the TD error (i.e., $V^{target}(s_t) - V(s_t)$) with the following gradient updating rule:

$$\Delta w_t = \alpha(V_{t+1} - V_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial V(s_k)}{\partial w_i} \qquad (A.4)$$

The gradient updating rule given above suggests a 'plain' back-propagation-like adaptation, but some of the improvements developed for supervised learning (e.g., [88]) are likely to work for TD learning also.

To employ TD learning, we need to generate sequences of positions. Game sequences can be generated using game databases or games played by the learning program itself. In the latter case, a further choice can be made on the opponent. Possible options include using an already existing game program, playing against players with similar strength on the Internet, playing against itself, or using more learning players which improve their skill by playing together.

A.2   SANE

SANE[1] (**S**ymbiotic, **A**daptive **N**euro-**E**volution) [80, 81, 82] is a genetic algorithm that evolves a population of neural networks.

SANE maintains and evolves two populations: a population of neurons and a population of networks. Each individual in the neuron population represents a hidden neuron in a 2-layer feed-forward network. The chromosome associated to a neuron specifies to which input or output neurons it is connected, and the weight of the connections. Each individual in the network population specifies a set of neurons to include in a neural network.

The genetic algorithm of SANE has two phases: evaluation and recombination. In the evaluation phase each individual in the neuron population and in the network population is assigned a fitness value. The fitness of a neuron is the sum of the fitnesses of the best five networks in which it is included. The fitness for a neural network is provided from exterior by the 'user', depending on how the network is performing in the target environment.

In the recombination phase, the neuron population is ranked based on fitness values. The neurons are selected for recombination depending on their

---

[1]The source code of SANE is available from the University of Texas Computer Sciences, Neural Networks' home page http://www.cs.utexas.edu/user/nn/.

rank, and combined using a one-point crossover operation. The fittest neurons are preserved to the next generation. A similar recombination is used for the network population.

SANE was found to form 'good' neural networks fast, while maintaining the diversity in the population. To employ SANE, one has to specify mainly the topology of the neural network (i.e., the sizes of the input layer, the output layer and the hidden layer), and to implement the fitness function for the network.

A.3 RPROP

RPROP (**R**esilient back**prop**agation) [88, 89, 90] is a weight update rule that performs a direct adaptation of the weight based on local gradient information. As other update rules used in supervised learning for neural networks, RPROP employs the first-order partial derivative of each weight with respect to the over-all network error. In most cases the error is the mean square error between the target and actual output of the network (typical supervised training), summed over all training instances (i.e., batch learning). The form of the update rule, however, is not dependent on how the error function or the partial derivative is computed.

The basic principle of RPROP is to eliminate the (potentially) harmful influence of the size of the partial derivative on the weight step. Only the sign of the derivative is considered to indicate the direction of the weight update. The weight change is determined by a weight-specific, so-called 'update-value' Δw_i:

$$\Delta w_i(t) = \begin{cases} -\Delta_i(t), & \text{if} \quad \frac{\partial E}{\partial w_i}(t) > 0 \\ +\Delta_i(t), & \text{if} \quad \frac{\partial E}{\partial w_i}(t) < 0 \\ 0, & \text{otherwise} \end{cases} \tag{A.5}$$

where w_i stands for individual weights, E is the error function, and t is the moment of the update. The new update-values $\Delta_i(t)$ are computed with the following formula:

$$\Delta_i(t) = \begin{cases} \eta^+ \times \Delta_i(t), & \text{if} \quad \frac{\partial E}{\partial w_i}(t-1) \times \frac{\partial E}{\partial w_i}(t) > 0 \\ \eta^- \times \Delta_i(t), & \text{if} \quad \frac{\partial E}{\partial w_i}(t-1) \times \frac{\partial E}{\partial w_i}(t) < 0 \\ \Delta_i(t-1), & \text{otherwise} \end{cases} \tag{A.6}$$

with $0 < \eta^- < 1 < \eta+$

In words, equation A.6 works as follows. If the partial derivative of the corresponding weight retains its sign, the update-value is increased by the factor η^+ in order to accelerate convergence toward a local minimum. If the partial derivative changes its sign, which indicates that the last update was too large and the local minimum was jumped over, the update-value is decreased by the factor η^-.

At the beginning, all update-values are set to an initial value Δ_0. In order to prevent the weights from becoming too large or to 'freeze', the update-values are constrained between Δ_{min} and Δ_{max} (typically $1e^{-6}$ and 50.0, respectively).

Thus, RPROP has a relatively small number of parameters (η^+, η^-, Δ_0, Δ_{min} and Δ_{max}), and appears to be robust with respect to them. It was found having fast convergence, and a good generalisation ability on unknown inputs [88].

Appendix B

Generating the NN population

The set of evaluation functions included in the NN population consists of 100 neural networks. These 100 neural networks were generated by tuning a larger number of neural networks using a specific genetic algorithm (*SANE*) and the TD-learning algorithm, and selecting the best 100 out of these.

Tuning the evaluation functions with SANE

An evaluation function is tuned with a genetic algorithm by evolving the weights of the neural network. To evolve the weights of the evaluation functions with genetic algorithms, we used a method called SANE [81] (for a short description see appendix A.2). SANE is a fairly generic program for evolving neural networks. It requires the definition of two problem-dependent details: the topology of the neural network and the fitness function. The topology of the neural network specifies the size of the input, output, and hidden layer. These are described in section 2.1.2. The fitness function specifies how good a neural network is for a specific problem. In our case the fitness is defined as the tournament performance when the neural network is included in a game program as evaluation function. The performance is evaluated in a swiss tournament[1] with 15 rounds that included the neural networks from one generation. The best neural network from each generation is saved (50, in total).

Tuning the evaluation functions with TD learning

To tune evaluation functions through TD learning, we used the equations from appendix A.1, with $\lambda = 0.8$ and $\gamma = 0.9$. In order to use TD learning we have to generate game sequences. In general, game sequences can be generated using

---

[1]In the swiss pairinings system the players are paired in every round with opponents with approximately the same amount of points obtained so far. The swiss system is used in tournaments with a large number of participants.

game databases or games played by the learning program itself. In the latter case, a further choice can be made on the opponent. Possible options include using an already existing game program, playing against players with similar strength on the Internet, playing against itself, or using more learning players which improve their skill by playing together. In LOA, game databases and an Internet server are not available. The remaining option is to use games played by the program which use the evaluation function to be trained. Three training strategies were tested: *STD* (Single TD player), *MTD* (Multiple TD player) and *SANE-TD*. In the first strategy (*STD*) the evaluation function is trained using games played against itself. The neural network is saved after every 100 games. During the learning 10,000 games were played. From the resulting 100 neural networks the top 50 were selected using a 20-round swiss tournament. The second strategy (*MTD*) used 10 TD players each being trained by playing against the rest. For this strategy, the top 50 networks were selecteded in a similar way as for STD. The third strategy (*SANE-TD*) used three TD players that played against SANE opponents. The TD players were included in the tournament evaluating the fitness of the SANE neural networks (see above). After the 15-round tournament has been finished, SANE, through recombination, generated a different set of neural networks (a new generation), and the training of the TD players continued with these. After each tournament the there TD neural networks are saved. From the resulting 150 networks (3 networks × 50 generations) the top 50 were selected using a 20-round swiss tournament.

Selecting the NN population

We compared the four methods (*SANE, STD, MTD, SANE-TD*) in a tournament with 50 players for each method (a total of 200 players). The tournament used the swiss pairing system and lasted 50 rounds. The results are summarised in Table B.1. The result for each method is given as the ratio between the total amount of points obtained by the players representing the method, and the total number of points in the tournament. The standard deviation reflects the deviation in performance inside a method. The tournament resulted in a clear superiority of the TD players over the genetic ones. Out of the three TD training strategies, *SANE-TD* had the worst result, due to the weaker opponents (provided by the genetic algorithm). The values of the standard deviation indicate that the advantage of *STD* and *MTD* compared to the other two methods is significant, whereas the difference between *STD* and *MTD* was not significant. To obtain a better set of evaluation functions, we generated 60 more players using the two 'pure' TD methods (30 with *STD* and 30 with *MTD*). These 60 new players together with 60 top players (30 *STD* and 30 *MTD*) from the first tournament are included in a second tournament (Table B.2).

The final set of evaluation functions were selected as the top 100 players from the second tournament based on their performances (see last column of Table B.2).

method	# of players	result	std. dev.
sane	50	0.199	0.027
std1	50	0.276	0.024
mtd1	50	0.288	0.025
sane-td	50	0.237	0.011

Table B.1: Selecting the evaluation function population: 1st tournament. The players for *sane* are generated using *SANE*, for *std1* using *STD*, for *mtd1* using *MTD*, and for *sane-td* using *SANE-TD*.

method	# of players	result	std. dev.	final set
std1	30	0.228	0.049	20
mtd1	30	0.261	0.037	28
std2	30	0.238	0.038	24
mtd2	30	0.273	0.051	28

Table B.2: Selecting the evaluation function population: 2nd tournament. The players for *std1* and *mtd1* are from the 1st tournament, and the players for *std2* and *mtd2* originate from a new set generated using *STD* and *MTD*.

Appendix C

Examples of move ordering

Appendix C presents the move ordering performed by the Neural MoveMap heuristic in test positions originating from the four test sets described in subsection 3.3. The positions are labelled with the moves played in the game were it occurred. The neural network used for move ordering was trained with the learning set *ALL* (see subsection 3.4.4). The positions were selected randomly from the four test files: #1–25 is from A30, #26–50 is from B84, #51-75 is from D85 and #75–100 from E97 [72].

#1 WTM: Nd6

#2 WTM: Rd3

#3 WTM: Nc3
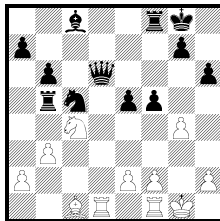
Rd6 Nd6 gf5 Ba3 f3 Ne5 g5 e3
b4 Na3 Be3 Bf4 Bb2 h4 Rfe1
Kg2 Nb6 Ne3 f4 Rd4 Bd2 Rd5
Nd2 a3 a4 Na5 h3 Rd3 Rde1
Nb2 Kh1 e4 Bh6 Rd2 Bg5
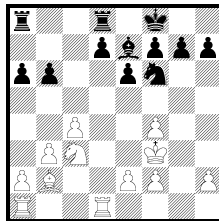
Rac1 Kg2 e3 e4 Rd4 Kg3 Ke3
Rg1 Na4 a3 a4 Rd2 Rd3 Rd7
Ne4 Nd5 Re1 Nb5 Rh1 b4 h3
Ba3 Rdc1 Rd6 f5 Rf1 c5 Rd5
Nb1 Bc1 Rab1 h4 Rdb1
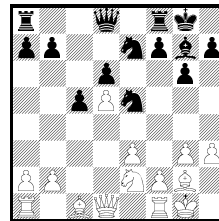
Nc3 e4 Nf4 b3 f4 a4 Rb1 Kh2
Bd2 Qb3 g4 Nd4 Qc2 Be4 Qd2
a3 h4 Qa4 f3 b4 Bh1 Re1 Qd3
Qe1 Kh1 Bf3 Qd4

#4 WTM: Rad1

#5 WTM: Bb7

#6 WTM: dc6
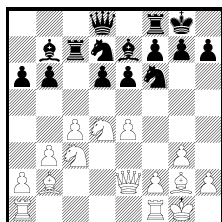
Rad1 Rfd1 f4 Rac1 f3 a4 Rfe1
Nc2 Rae1 h3 e5 a3 Na4 Nf5 h4
Nf3 Nd1 Kh1 Bh3 Ncb5 Qd2
Nb1 Nd5 Nc6 Qd3 Qe3 g4 Qe1
b4 Ne6 Bc1 Ba3 Qf3 Rfc1 Ndb5
Bf3 Bh1 Rfb1 Qg4 Qc2 Qh5
Rab1 Qd1 c5
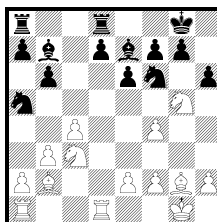
Nf3 Bb7 Nge4 Nh3 Rac1 e3 Nf7
Rd4 e4 Nb5 f5 Nce4 Rd7 Kf1
Ne6 Kh1 b4 Re1 Nd5 a3 Rd3
Ba3 Bh3 Na4 Bh1 Rd2 h3 Nb1
h4 Rd6 Rdc1 Nh7 Rdb1 c5 Bc1
Bf3 Bd5 Bf1 f3 Rab1 Rf1 a4
Be4 Bc6 Rd5
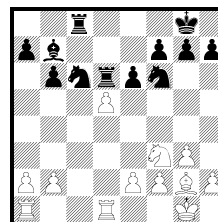
dc6 Rac1 de6 e4 a3 e3 Rd2 Nd2
b3 Rdc1 Ng5 Ne5 Rd4 Re1 Bh3
a4 h3 Ne1 b4 Kf1 Bf1 Rab1
Nd4 Rd3 h4 Nh4 Rf1 Rdb1
Bh1 g4 Kh1

#7 WTM: Qe2

#8 WTM: Ne4

#9 WTM: Rb1
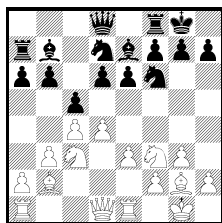
Qe2 Rc1 Qc2 d5 Qd2 e4 Nd2
a4 dc5 Ne5 h3 a3 Qd3 Ne4 Na4
Re2 Nh4 h4 Ne2 Bf1 Ng5 Bh3
Ba3 g4 Bh1 Qb1 Nb1 Nd5 Qc1
Rf1 Kf1 Bc1 Rb1 Nb5 b4 Kh1
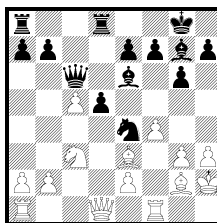
Ne4 Bd4 Rc1 Be4 f5 Nd5 Qc2
Qd4 Qb3 Qd2 Qc1 b3 Qa4 Rb1
Bg1 g4 Bf2 Nb5 a3 Qe1 Na4 b4
Rf2 Bc1 Qd3 Qb1 Nb1 a4 Rf3
h4 Bh1 Bf3 Bd2 Kg1 Rg1 Qd5
Rh1 Re1 Kh1
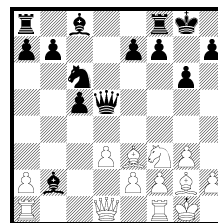
Rb1 Rc1 Qd2 Qa4 Bh6 Bc5
Qb3 Qc2 Qc1 Bd2 d4 Nd2 a4
a3 Qb1 Ng5 h4 h3 Bg5 Bf4 Nh4
Ne1 Bh3 g4 Qe1 Re1 Bh1 Bc1
Ne5 Kh1 Bd4 Nd4

#10 WTM: Nb6

#11 WTM: Nd5

#12 WTM: Nd4
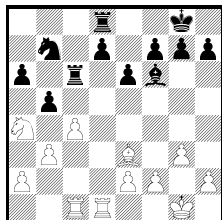
cb5 Nc5 c5 Nc3 Nb6 Rd7 Bb6
Rd2 Nb2 a3 Bd2 Bd4 b4 Ba7
Rc2 Bf4 Bc5 Kf1 Rd3 Rd5 Ra1
f3 Rf1 Bg5 h4 Rc3 g4 Re1 Rd6
h3 Rd4 Kg2 Rb1 f4 Bh6 Kh1
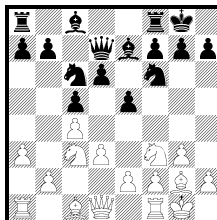
b4 e4 Be3 Nd5 Bg5 h3 Rb1 Qc2
Bf4 Nd2 Bd2 Re1 e3 b3 Nh4
Ne1 Ne4 Qb3 Ng5 Qd2 Bh6
Qa4 Nb5 Ne5 a4 g4 Qe1 Na4
Bh3 Bh1 Kh1 h4 Nb1 Ra2 Na2
d4 Nd4
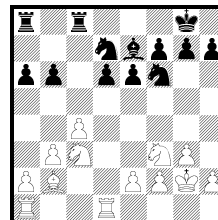
Rac1 e4 Nd4 Rd4 Rd2 a3 Rd6
h3 Rd3 Nd5 e3 Nd2 a4 Na4
Ne1 Ne4 Rh1 Kh1 h4 Rdc1 b4
Ba3 Re1 g4 Ng1 Nb1 Nb5 Bc1
Nh4 Rdb1 Kf1 Kh3 Rg1 Ng5
Rab1 Rf1 Rd5 Ne5 Kg1 c5

#13 WTM: Qd4

#14 BTM: Bf8

#15 BTM: Bf6
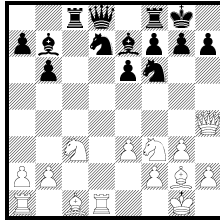
Nd4 e4 Qd4 Ng5 b3 Qh6 Qc4
Qh3 Bd2 Qg5 Qb4 a3 Ne5 Ne4
a4 Qg4 b4 Rd4 Nb5 Ne1 g4
Rd7 h3 Qf6 Bh3 Nd2 Bf1 Ne2
Qe4 Bh1 Qf4 Rb1 Kf1 Qa4 Nb1
Rf1 Na4 Re1 Rd3 Kh1 Rd6
Qh5 Qh7 Nd5 Rd2 Rd5
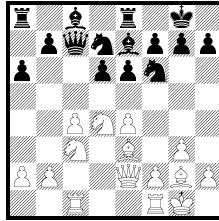
Nc5 e5 Ne5 Bf8 b5 Nf8 b6 Qb8
Qc6 Rb8 Nb6 Qd8 Qc4 Qa5
Ng4 h6 Qc5 Bd8 g6 Rd8 Qb6
Nb8 h5 g5 a5 Ne4 Nh5 Ra7 Kf8
d5 Kh8 Rf8 Nd5
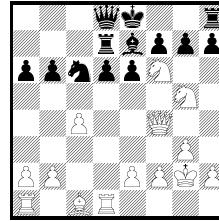
Bf6 gf6 Kf8

#16 BTM: Nf6

#17 BTM: b5

#18 BTM: Ne8
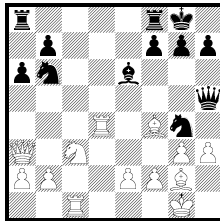
Nf6 Ne5 Rac8 Nc4 Nh6 Rfc8
Rfe8 Rfd8 Nd5 Qe5 Bd5 Ne3
Rad8 f6 Nf2 Bd7 Qc5 g5 Bc4
a5 Qg6 Qf5 Rae8 Nh2 Ra7 Qh4
Qh6 Ba2 Qg5 f5 Nd7 Bf5 Rab8
Nc8 h6 Bc8 Na4 Rfb8 Kh8 Bb3
Qb5 Qh3 g6 Qa5 Qd5
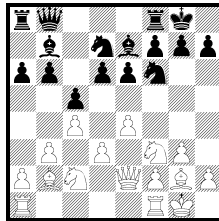
Qc7 a5 b5 Re8 e5 Rd8 Rc8 Ne8
Qa7 h6 Bd8 Ra7 Qe8 d5 Bc6
Qc8 Ne5 Be4 Nd5 g6 Nh5 g5
Ne4 Ng4 Kh8 h5 Bc8 Qd8 Bd5
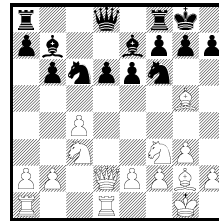
Rc8 h6 Qc7 Qd7 a6 Na5 Re8
d5 Qc8 a5 Nb4 Qe8 Rb8 Ne5
Ne4 Qb8 e5 Nd7 Nb8 Bc8 Ng4
Kh8 h5 Ba6 b5 g6 Nd5 Nd4
Ne8 Nh5

#19 BTM: Be6

#20 BTM: Nfe4
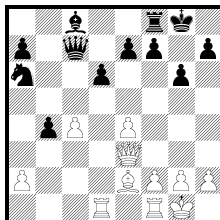
#21 BTM: Nd5

Nc5 e5 Bb7 Bd7 Be6 Qa5 Nb8
Rd8 Qc5 Kg7 Qb8 Bg4 Qd8
Qc6 f5 e6 f6 Bf5 Re8 Qb7 b3
Qd7 Kh8 Bh3 g5 h5 Qb6 h6
Qc4 d5
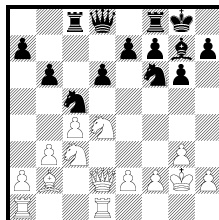
Qc7 Qd7 a6 Nce4 a5 Ne6 Nfe4
Re8 Ng4 e5 e6 Ncd7 Nb7 h5
d5 Rc7 Qe8 Nfd7 Nh5 Rb8 Na6
Rc6 Bh6 g5 h6 Na4 Nb3 Ra8
Ne8 Bh8 Nd3 Kh8 b5 Nd5
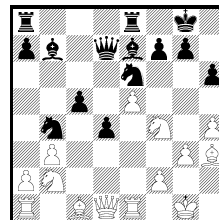
Nf4 Rad8 Nd5 Bf6 Bf3 Bd6 d3
Qc7 Bf8 a5 Nd3 Red8 Na2 Nf8
Bg5 g5 Ba6 Bh1 a6 Bd8 g6 f5
Qb5 h5 Nc2 Bh4 Bc8 Qc6 Rf8
Rac8 Rab8 Bc6 c4 Be4 Kh8
Qd5 Qd6 Kf8 Reb8 Ng5 Kh7
Na6 Qa4 Qc8 f6 Nc7 Rec8 Bg2
Nc6 Bd5 Qd8 Nd8

#22 BTM: Be7
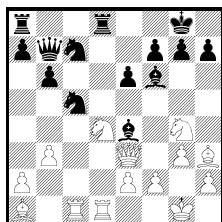
Rd5 Bg6 Be7 Nd5 Nb5 Rac8
Ne8 N7a6 a6 a5 Qd5 Bf3 Bd4
Re8 Rd7 Na4 Be5 Nd7 Bc6
N5a6 Rd4 b5 Rf8 Bb1 Bd5 Bg2
Kf8 g6 Nd3 Bg5 Rdc8 Qa6 Bf5
Qc8 h6 Nb3 Rdb8 Rd6 Bc2 h5
g5 Bh1 Kh8 Qc6 e5 Bh4 Rab8
Bd3 Qb8

#23 BTM: Nbd7
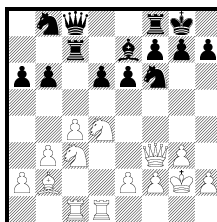
Qb7 Nbd7 Nc6 Rd8 Rb7 Qd7
Rc4 Rc5 d5 Re8 Rc6 Nfd7 h6
b5 e5 h5 Ne8 Rd7 g6 Qd8 g5
Nh5 Ne4 Qe8 Bd8 Nd5 Ng4 a5
Kh8 Ra7

#24 BTM: Qb6
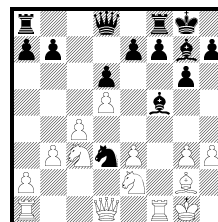
Qa5 Bc3 Be5 Qd7 e5 Qb6 Rc8
Bd7 a6 Nb4 Nf4 Rb8 Nb2 Nf2
Be4 a5 Bc8 Bg4 Bh6 Bh3 Qc8
b5 Nc1 Nc5 h5 g5 Qb8 h6 Bf6
Kh8 e6 f6 Qc7 Ne1 Bh8 Be6
Qe8 Bd4 Ne5 b6 Re8

#25 BTM: Nb7
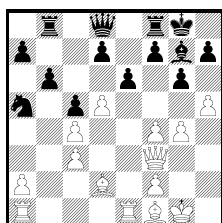
ed5 Nb7 Qh4 Qc7 Nc4 f5 d6
Qf6 Bd4 Bc3 a6 h6 b5 f6 Nc6
Rc8 Rb7 Bf6 Qc8 g5 Kh8 Ra8
Re8 Bh6 Bh8 gh5 Qe8 Qe7 e5
Be5 Qg5 Nb3

#26 WTM: Nd2
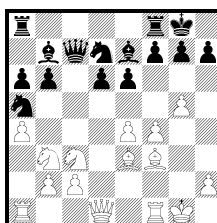
Na5 Bg2 Qe2 Kh1 f5 Qd2 Nd5
Qd4 Ne2 Nd2 Bg4 Bd4 Rc1 Rf2
Qe1 Qd3 Be2 Nd4 Bh5 g6 Bf2
Re1 Bc1 h4 e5 Bb6 Bd2 Nb5
Na2 Ra2 Nb1 Qb1 Qd6 Qd5
Kf2 Ra3 Qc1 h3 Kg2 Nc1 Bh1
Rb1 Bc5 Nc5

#27 WTM: Nd4
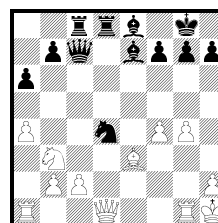
Nd4 Bd4 Qd4 g5 Rg3 c3 Qe2
Qf3 f5 a5 Qe1 Rg2 Re1 Qd2
Nc5 h4 Qd3 h3 Qf1 Na5 Nd2
Qc1 Rf1 Rc1 Ra3 c4 Ra2 Qb1
Kg2 Bd2 Bc1 Nc1 Bf2 Rb1

#28 WTM: e5
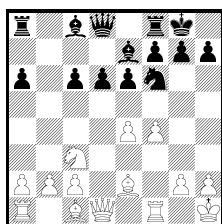
Be3 Bf3 e5 a4 Bd3 Na4 Qe1 f5
b3 a3 Qd4 Bc4 Qd3 h3 g4 Bd2
Rb1 Ba6 b4 Nb1 g3 Rf3 Kg1
Rg1 Bh5 Qd6 Bg4 Rf2 Qd2
Re1 h4 Nb5 Bb5 Nd5 Qd5

#29 WTM: Rae1
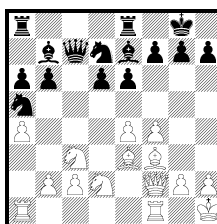
Bd4 Qg3 Rae1 Rad1 Bb6 b3 f5
g4 e5 Bg4 Be2 Rac1 Ne2 Nd5
b4 Rfe1 Bh5 Nd1 Nc4 Rfd1
Rg1 Ra3 Qg1 g3 Na2 Nb5 Qh4
Qe2 Rfc1 Nb3 Ra2 Rab1 h4
Ncb1 Bd1 Kg1 Rfb1 h3 Ndb1
Qe1 Bc5

#30 WTM: Bb7
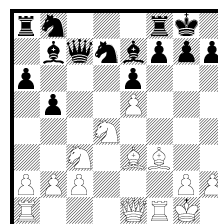
Bb7 Qg3 Rd1 Bf4 Kh1 Ne4 Bf2
Qf2 Rc1 a3 Ne6 a4 Bh5 Nf5
Nce2 g4 Qe2 Ncb5 Qc1 Bh6
Bc1 b3 Nd5 h4 Bd2 Nde2 Bg4
Be4 Qd1 Qd2 Be2 Bg5 Na4
Ndb5 Rf2 Bd5 Nb3 g3 Nb1
Rb1 Qb1 Qh4 Nd1 b4 Kf2 h3
Nc6 Bc6 Bd1

#31 WTM: g5
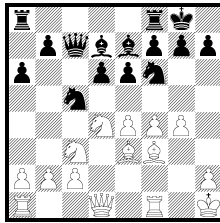
g5 Qe2 e5 Qd2 f5 Nb3 Qe1 Rc1
h4 Rg1 Nde2 h3 a3 Bg2 Kg1 a4
Bc1 Nce2 Rb1 b4 Nf5 b3 Re1
Bg1 Rf2 Qd3 Qb1 Nd5 Be2
Na4 Kg2 Qc1 Nb1 Ne6 Bd2
Ncb5 Bf2 Ndb5 Nc6

#32 WTM: Nd5
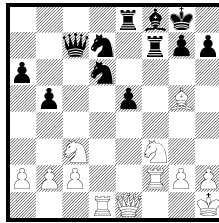
Nd5 Rd5 Rd6 Qg1 Qf1 Re2 Bf4
a3 Nh4 Ne4 Ne5 Bd2 b3 Bf6 a4
Nb5 h3 Bc1 Bd8 Rd3 Nd2 Qe2
Bh4 Be3 Kg1 g4 Nb1 Qd2 Ng1
Rfd2 Rb1 g3 h4 Na4 Nd4 Rc1
Ra1 Ne2 b4 Bh6 Qe4 Rdd2 Be7
Qe3 Qe5 Rd4 Rf1

#33 WTM: Ng5
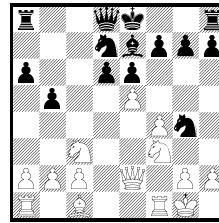
ed6 h3 Nd4 Ne4 Kh1 Nd1 Qe4
Bd2 Be3 a4 Ng5 f5 Rd1 a3 b3
Qd3 Nd5 Qb5 Nb5 g3 Re1 Rb1
Qd1 Nb1 Ne1 Nd2 b4 Nh4 h4
Qe3 Qd2 Rf2 Na4 Qf2 Qe1 Qc4

#34 WTM: Bd4
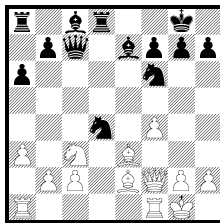
Bd4 Rad1 Bf3 Rfd1 Bc4 Nd5
Bd3 b4 Bd1 Rae1 Na4 Qh4 g4
Nd1 Rac1 f5 Kh1 h3 Rab1 g3
a4 Ba6 Ra2 Bh5 Qg3 Nb1 Ne4
Qf3 Bg4 Nb5 Qe1 Na2 Rfb1
Bd2 Bc1 Bb5 Rfe1 b3 Rfc1 h4

#35 WTM: a3
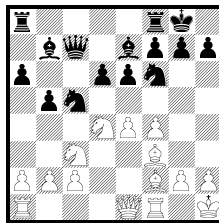
Rd1 e5 Bg3 Bh4 a3 Nb3 Rc1 a4
Bg1 g4 b3 Qe3 f5 Nce2 Nde2
Rg1 b4 Be2 Nf5 g3 Rb1 Qe2
Ncb5 Kg1 Ne6 h3 Qd2 Qc1
Nd1 Ndb5 Be3 h4 Nc6 Qd1
Qb1 Bh5 Nb1 Na4 Nd5 Bd1
Bg4

#36 WTM: Qc2
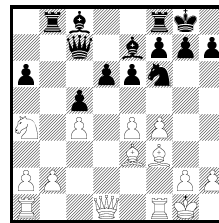
Rc1 e5 b3 Nb6 Nc3 Qe2 Qd2
Qc2 Kh1 Qd3 Bc5 g4 Be2 Qe1
a3 b4 Rf2 f5 g3 h3 Bd4 Bc1
Rb1 Qc1 Bd2 Re1 Bf2 Bh5
Qd4 Kf2 Qb1 Qb3 Nc5 Bg4 h4
Qd5 Qd6

#37 WTM: Rae1
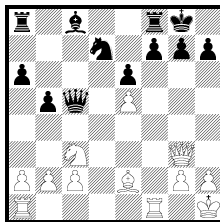
Ne4 Rad1 Bd3 Qf4 Qf2 a3 Rf4
Bf3 Bh5 a4 Rae1 Bg4 Nd1 Qh4
Qg5 Rac1 Qg7 b4 Rf2 Rfd1
Nb5 h4 Rf3 Qh3 Na4 Qg4 Qg6
b3 Nb1 Bb5 Rab1 Rg1 Bd1
Nd5 Rfb1 Qe3 Rf7 Qe1 Qf3 Rf6
Rfc1 Qd3 h3 Rfe1 Rf5 Bc4

#38 WTM: Qf3
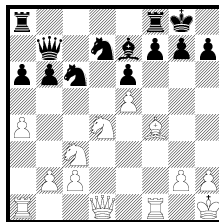
Nc6 Qe2 Qf3 Qg4 Qh5 Ne4
Qd2 Qe1 Bg3 Nb3 Nf5 Nf3 Rf3
Qd3 Re1 Nce2 Ne6 Nd5 b3 Bh6
Bg5 g4 Kg1 h4 Be3 Ncb5 a5
Rf2 Rg1 h3 b4 Qb1 Ra2 Nb1
Na2 Rc1 Rb1 Ra3 Nde2 Ndb5
g3 Qc1 Bc1 Bd2

#39 BTM: Bg5
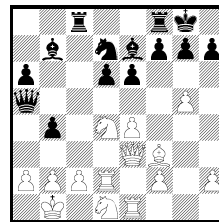
Nc5 Ne5 Bg5 Rfe8 e5 Rc4 Rfd8
Qc5 Nb6 Rc7 d5 Rc5 Nf6 Qc7
g6 Qg5 Bc6 Bd5 Qa4 Qb5 Rb8
Rc3 Bf6 Kh8 Nb8 f5 b3 Qa2
Bd8 Be4 Ra8 Rcd8 Rce8 h6
Rc6 Qb6 Qe5 Ba8 Qa3 f6 h5
Rc2 Qd5 Qd8 Qf5

#40 BTM: Qe5
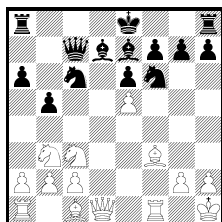
0-0 Ne5 Nd5 b4 Rc8 Rd8 Qe5
Kc8 Rb8 Nb4 Na5 Ng4 h5 Bc8
g5 Bd6 Qb6 h6 Ra7 Qb8 Bc5
Qa7 Qb7 Ng8 Qd8 Bb4 Ne4
Na7 g6 a5 Bf8 Kf8 Ba3 Bd8
Rf8 Rg8 Kd8 Nb8 Nd8 Qd6
Nd4 Qa5 Nh5 Qc8

#41 BTM: Be7
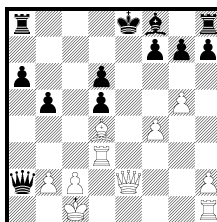
Be7 Kd7 Kd8

#42 BTM: Nd4
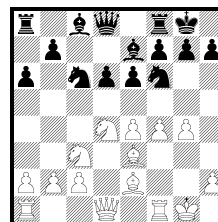
Nd4 Qc7 b5 Bd7 Nd7 d5 e5
Ng4 Re8 Ne8 Ne4 Nd5 Qa5
Rb8 h6 b6 g6 Nb4 Na5 h5 Qb6
Kh8 Nb8 Ne5 g5 Ra7 Qe8 Na7
a5 Qd7 Nh5

#43 BTM: Rf8
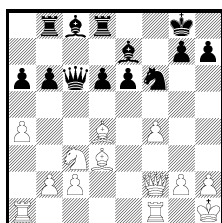
Bb7 d5 e5 Qc5 Rb7 Bd7 Qc7
b5 Nd5 Rf8 Qe8 Nd7 Ne8 Ng4
Re8 Ra8 Kh8 g6 h6 Qa8 a5 Kf7
Ne4 Qg2 Qc4 Kf8 Bf8 Qb7 Nh5
Qa4 Qe4 Qd7 Rd7 g5 Qf3 h5
Qd5 Qc3 Qb5

#44 BTM: Qb4
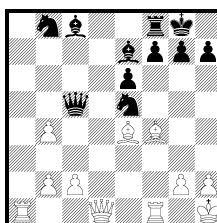
Nbc6 Qc7 Qb4 Qb6 Bb7 Ng6
f5 Rd8 Ba6 Nc4 Qb5 Qa7 Bf6
Qd5 g6 Ng4 Nec6 Bd8 Qd6
Bh4 Nd3 g5 Qd4 Qc3 Qc6 h6
Qc4 Kh8 Bd6 Qa5 Qg1 Bg5
Bd7 Nbd7 Nf3 h5 f6 Qc2 Qf2
Na6 Ned7 Re8 Qe3

#45 BTM: Qc7
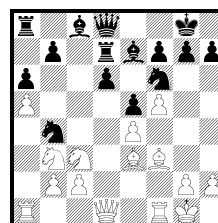
Qc7 d5 Bf8 Nc6 Rc7 Nc2 b5
Nbd5 b6 Ne8 Rb8 h6 Qf8 Ng4
h5 Qe8 Nfd5 Na2 Ra7 Kh8 Kf8
g5 Nd3 g6 Nh5 Qb6 Ne4 Qa5

#46 BTM: Qb2
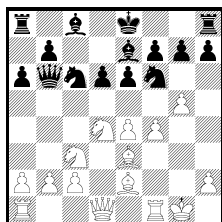
0-0 Nd7 Nd4 Bd7 Qb2 Qc7
Qd4 Qa5 Qd8 Ng4 Nh5 Bf8 d5
Ng8 Nd5 Qb4 Qc5 Ne4 e5 g6
Nb4 Rg8 Na5 Bd8 h6 Qb5 Nb8
Rb8 Qa7 Nd8 h5 Kf8 Rf8 Kd8
Na7 Ra7 Qb3 Ne5 Kd7 a5

#47 BTM: Nd5
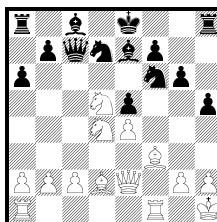
Nd5 ed4 0-0 Qd8 Bd8 h4 Bd6
Qc6 Nh7 Nc5 Qd6 b5 Bc5 Nb6
Bf8 Ng4 Kf8 Ne4 Qg6 Qb8 Rh7
Ng8 Kd8 b6 Nb8 Rb8 Ra7 Rg8
g5 a5 Nf8 Rf8 Bb4 Qc4 Ba3
Qc2 Rh6 Qa5 Qc5 Qc3

#48 BTM: Be7
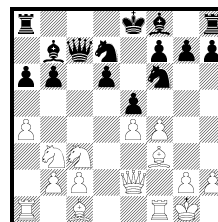
Be7 ef4 Rc8 Nc5 Kc8 g6 Bc6 b5
h6 h5 a5 d5 Rb8 Ke7 g5 Nb8
Qc6 Rd8 Qb8 Qc3 Ng8 Ra7
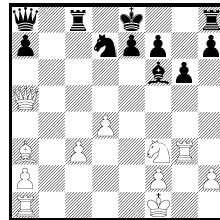Kd8 Qc5 Bc8 Qd8 Ne4 Rg8
Qc4 Be4 Nh5 Bd5 Ng4 Nd5
Qc8

#49 BTM: Nc5
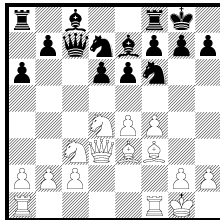
Nc5 b5 e5 Re8 Ne5 Nb6 Rd8
Rb8 b6 g6 d5 Ne8 Ng4 Nd5
Qc5 Qc4 Qc6 h6 Qb8 Nb8 Bd8
Qd8 Kh8 Qb6 h5 Ra7 g5 a5
Qa5 Nh5 Ne4 Qc3

#50 BTM: de5
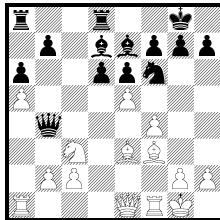
de5 Bc6 Qa5 Rac8 Qb2 d5 Nd5
Qc3 b5 Be8 Ne8 Bb5 Rab8 Ng4
Qc4 Re8 Qb3 Qc5 Ba4 Rdc8
Bc8 Qa3 Rdb8 Ne4 g6 h6 Kf8
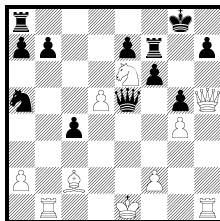h5 Qf4 Qd4 Rf8 Ra7 g5 Bf8
Kh8 b6 Nh5 Qb6 Qa4 Qe4 Qb5

#51 WTM: Re1

Rg2 Qa7 Kg1 Re1 Rd1 Rc1
Ke1 Rb1 Kg2 Qf5 Ke2 Qa6
Nd2 Ne1 h4 Ne5 Bc1 Qb5 Qd5
c4 Qa4 Ng5 Qb6 Ng1 Nh4 Bb2
Bc5 Qh5 Qe5 d5 h3 Qc5 Rh3
Bd6 Rg4 Rg6 Be7 Qb4 Rg5
Qc7 Bb4 Qd8 Qg5 Rg1

#52 WTM: c4
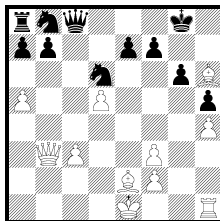
c4 Qb2 f4 Bd2 Qb4 Rg1 Bg5
Kd2 Qd1 Bf4 Qa3 Bd3 Kf1 Qc2
Kd1 Bb5 Bc1 Bg7 Qb1 Bd1
Qb5 Rh2 Bf8 Rh3 Bc4 Be3 Qa4
Qa2 0-0 a6 Qc4 Bf1 Qb6 Ba6
Rf1 Qb7

#53 WTM: Kf1

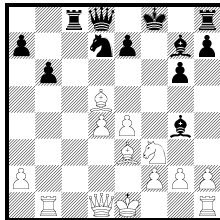Kf1 Kd2 Kd1 Be4

#54 WTM: ba3
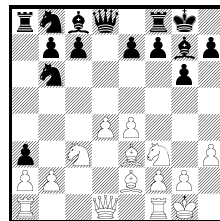
ba3 e5 Qd2 d5 Qb3 Rc1 Qc2
b3 Ne5 Nd2 Bb5 Nb5 Bh6 Nd5
Qc1 Re1 Bd3 Na4 Nh2 g4 Ng5
Nb1 Bf4 h4 Bc4 Qb1 Bg5 Ne1
Qd3 Kh2 Qe1 Bd2 Bc1 Rb1
Kh1 Nh4 g3 Qa4 b4 Ba6
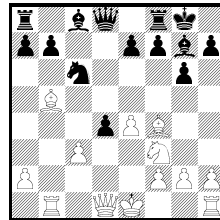
#55 WTM: Qb3
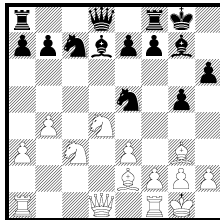
Be5 Rc1 Qb3 Rb1 f4 Qd2 Nf5
Qc2 f3 Bh5 b5 Nb3 h4 Nf3 h3
a4 Ra2 Bg4 e4 Ndb5 Bc4 Qc1
Ncb5 Qb1 Qd3 Bd3 Nd5 Kh1
Nc2 Qa4 Ba6 Bf3 Qe1 Na4 Bb5
Ne6 Na2 Bh4 Ne4 Re1 Nb1
Nc6 Bf4

#56 WTM: h3

0-0 Ba8 Bb7 h3 Qb3 Qd2 Qa4
Qd3 e5 Ng5 Rb3 Qe2 Bb3 Rb4
h4 Qc1 Rc1 a4 Kd2 Kf1 Bc4
Be6 Bd2 Bf7 Bc6 Bh6 Ng1 Bc1
Ke2 Nd2 Ra1 a3 Bg5 Ne5 g3
Nh4 Qc2 Rf1 Rg1 Rb6 Bf4 Rb5
Rb2 Bg8

#57 WTM: Bc6

cd4 0-0 Bc6 Nd4 Ba4 Be3 Bc4
Qa4 Bd3 Qe2 Be2 Qd2 Bd2
Qd4 h4 Bh6 Qd3 Bg3 h3 e5
Ke2 c4 Kf1 Qc1 Bg5 Bf1 Bd6
Bb8 Ng5 a4 Rc1 Bc7 a3 Rg1
Qc2 Qb3 Bc1 g4 Ba6 g3 Ng1
Rb3 Ra1 Rb4 Nd2 Rb2 Rf1
Nh4 Kd2 Be5 Ne5

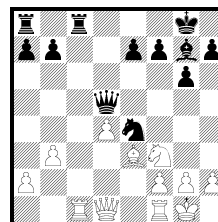**#58 WTM:**   Bb4

#59 WTM: Bc3

#60 WTM: Rc2
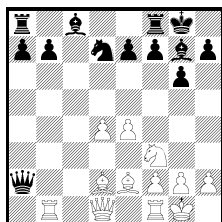
Bb4 Bg5 Ra1 Rc1 Bc3 d5 Qc1
Be3 Re1 e5 Bd3 Bb5 Qc2 Rb7
Rb3 Ng5 Bc4 Ne1 Rb5 Rb4 h3
Ba5 Bf4 Ne5 Be1 Kh1 Nh4 Qe1
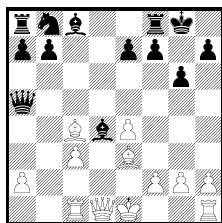Qb3 Ba6 h4 g3 Rb2 Bc1 g4 Bh6
Rb6 Qa4
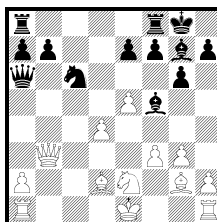
0-0 Kf2 f4 Rc1 Nf4 Bc3 d5 g4
Bf1 Rb1 Bf4 Qb5 h4 Bb4 Kc1
a4 Nc3 Be3 Qa3 Qb2 Qb7 Kf1
Qe3 a3 Bc1 Bh3 Rf1 Qd1 Qc3
Qd3 h3 Ng1 Qd5 Bg5 Qb1 Kd1
Bh6 Qb6 Nc1 Qb4 e6 Qf7 Ba5
Rd1 Rg1 Qc2 Qa4 Qc4 Qe6

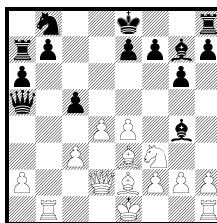Rc8 Ne5 Re1 Qe2 Rc5 Rc2 Qd2
Nd2 Ne1 h3 Rb1 Rc7 Qd3 Rc4
Qc2 b4 Bd2 Ng5 a4 Qe1 g3 a3
Rc3 Bh6 h4 Nh4 Bf4 Bg5 Ra1
g4 Kh1 Rc6

#61 WTM: Qd4
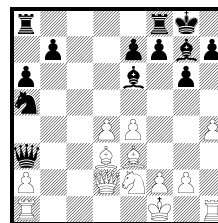
#62 WTM: 0-0

#63 WTM: Rb1

Bd4 Qd4 0-0 Qb3 Bb3 Qf3 Bd3
Bb5 Bd2 Qd2 Qe2 Bd5 f3 Be2
Bh6 Bf7 Qd3 Kf1 h4 f4 Qg4
e5 Bf4 Be6 Ke2 a4 Ra1 Bf1 h3
Qh5 Rc2 Qc2 g3 Ba6 Rb1 Rg1
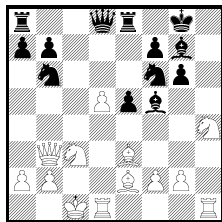g4 Rf1 Qa4 a3 Kd2 Bg5

0-0 d5 dc5 h3 e5 c4 a4 Bh6 Rb7
Ne5 h4 Rb2 Rb5 Bf4 Kf1 Rb6
Rc1 Ng5 Qc1 Ng1 a3 Qc2 Bd1
Rb3 Qb2 Qd3 Rg1 Bc4 Nh4
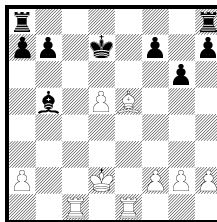Ra1 Bg5 Rb4 Bd3 Kd1 g3 Bb5
Rf1 Rd1 Bf1 Ba6 Qd1

h5 Rc1 Rb1 Bh6 Qa5 d5 f3 e5
g4 Bc2 Nc3 Nf4 Rh3 f4 Kg1
Qc2 Qb2 Nc1 Bg5 g3 Qc1 Rh2
Ng1 Bb1 Bf4 Qb4 Qe1 Rd1
Rg1 Ng3 Ba6 Re1 Qc3 Bb5 Bc4
Ke1 Qd1

#64 BTM: Bd7

#65 BTM: Rhc8

#66 BTM: Na5
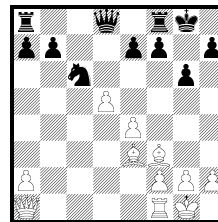
Be6 Bd7 Nbd5 Bg4 Ng4 Nbd7
e4 Rc8 Nfd5 g5 Qd7 Qc7 Nc8
Qe7 Ne4 Nh5 Bd3 Bc8 Bf8 a6
Qc8 Kh7 Nh7 Qd6 a5 Rf8 Nc4
Na4 Qb8 Bh6 Qd5 Kh8 Bc2
Bh8 Be4 Kf8 Re6 Bb1 Rb8 Re7
Nfd7 Bh3
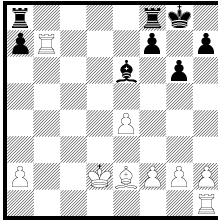
Rhe8 Rhc8 f6 Rac8 Bc4 b6
Rae8 Ba6 Rhg8 Rhf8 Rad8
Rhd8 h5 f5 a5 Ba4 g5 Ke7
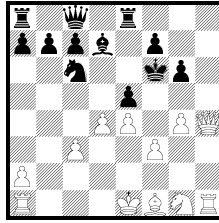Rhb8 Ke8 Rab8 Bd3 Bf1 h6
Bc6 Rag8 a6 Raf8 Be2 Kd8

Ne5 Rc8 Nd4 Na5 Qa5 f6 Nb4
e5 Qd7 e6 b6 Rb8 Qc7 a6 Qd6
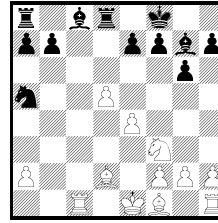Qd5 f5 Nb8 Re8 b5 a5 Qc8 g5
Qb6 Qb8 h5 Qe8 h6

#67 BTM: Rfd8

#68 BTM: Kg7

#69 BTM: b6

Rfd8 Ba2 Rfc8 Rad8 Rac8 Bc8
Bd5 Rab8 Bd7 Bc4 a5 Rfe8 f6
Kg7 Rfb8 f5 a6 g5 h5 Bb3 Bg4
Kh8 Rae8 Bh3 h6 Bf5

Kg7 Ke6 g5

Nc6 b6 Bf6 Bg4 Bb2 Bd7 b5 e5
e6 Nc4 a6 Rd5 f6 Rb8 Bd4 Nb3
Be6 Be5 Bh6 Bf5 Bc3 Re8 Kg8
h6 h5 g5 Ba1 Bh8 f5 Rd6 Rd7
Ke8 Bh3

#70 BTM: Ba6
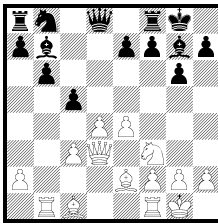
#71 BTM: cd4

#72 BTM: ba5

Nd7 cd4 Nc6 e6 Qc7 Ba6 c4
Be4 Qd6 a6 Qc8 Na6 Qd7 f5
Re8 a5 Bc8 f6 Bc6 b5 h6 Kh8
Qd5 Bd5 e5 Bd4 Bf6 h5 Bh8
Bh6 Qe8 g5 Be5 Qd4
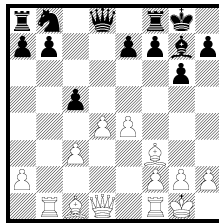
cd4 Nc6 Nd7 e6 Qc7 b6 Qa5
Bd4 c4 e5 b5 Qd7 Qd4 Qd6
Qc8 a6 Be5 a5 Na6 Re8 Kh8
Bh6 Qb6 f5 h5 Bf6 h6 f6 g5
Bh8 Qe8 Qd5
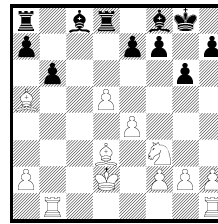
ba5 Bb7 Rd5 Bh6 e6 e5 Bg7
Bd7 Bg4 Rb8 Ba6 Re8 f6 Kg7
Rd7 Rd6 Be6 h5 b5 h6 a6 Kh8
g5 Bf5 f5 Bh3

#73 BTM: Rc8

#74 BTM: 0-0

#75 BTM: e5
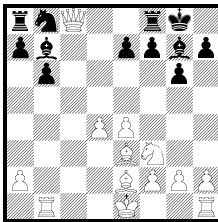
Rc8 Nd7 Be4 Nc6 e6 Na6 Rd8
a5 Bd5 a6 Bc6 Bf6 Ba6 b5 Bd4
Kh8 f6 Bh6 f5 Bh8 Re8 h6 h5
g5 e5 Be5 Bc8
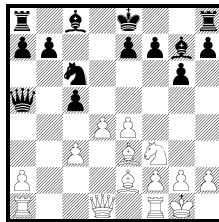
0-0 cd4 Bg4 Bd7 b6 Bf6 Qc3
Be6 c4 Qc7 e6 Qa4 Qa2 b5 Bf5
Qd8 Nd8 a6 f6 h6 h5 Rb8 Qb6
Qa3 e5 Bh6 Kd8 Qa6 Kf8 Qb5
Be5 Nb8 Bf8 Bh3 g5 Rf8 Ne5
Rg8 f5 Bd4 Nd4 Kd7 Nb4 Qb4
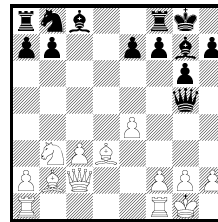
Nc6 Nd7 Bg4 b6 Bh3 Be5 Rd8
Qh6 Be6 Qh5 e5 Bh6 e6 Bf6
Na6 Qf4 h5 Qg4 a5 f5 b5 a6
Bf5 Bc3 Qe5 Bd7 Qe3 Kh8 Qa5
f6 Re8 Qd2 Qg2 Bd4 Qg3 Qd5
Qf5 Qh4 Qb5 Bh8 Qc1 Qf6 h6
Qc5

#76 WTM: c5
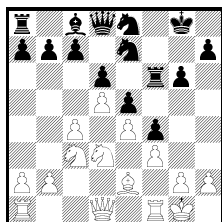
c5 Qd2 b4 Rc1 Kh1 Qb3 a4 Nf2
Nf4 g4 Re1 Nb5 Qe1 Rf2 Kf2
Qc2 a3 Qc1 Qa4 h4 b3 g3 Rb1
Na4 Nb4 Nb1 Ne5 Ne1 h3 Nc1
Qb1 Nc5

#77 WTM: Qc2
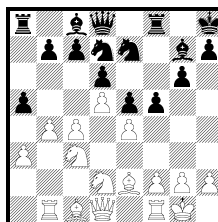
f3 ba5 Bb2 c5 ef5 b5 f4 Nb3
Qc2 Nf3 Nb5 a4 Kh1 Bd3 Re1
h3 g3 Bh5 Bg4 Qb3 Rb3 Bf3
g4 h4 Qe1 Ra1 Rb2 Na2 Qa4
Na4

#78 WTM: Rc2
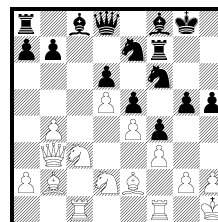
Nc4 a4 Nb5 Rfd1 Bc4 b5 a3
Qc2 Rg1 Qc4 Bb5 Qd1 Rc2 h3
Ba6 Rcd1 Kg1 h4 g4 g3 Bd3
Ba3 Ndb1 Na4 Ra1 Qa3 Rfe1
Rb1 Bd1 Qa4 Rf2 Ba1 Ncb1
Rce1 Nd1

#79 WTM: Ne4
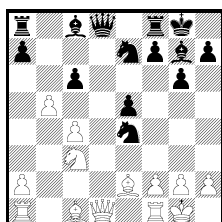
Ne4 Qd8 bc6 Be3 Na4 Ba3 f3
Rb1 Qa4 c5 Qc2 a4 Nb1 Bg4
Qd3 Bb2 Bg5 Bf3 Bd3 Qb3
Qd6 Re1 h3 Kh1 g3 Bh6 Nd5
b6 Qe1 Bh5 Qd2 Bf4 f4 g4 a3
Bd2 h4 Qd4 Qd7 Qd5

#80 WTM: Be3
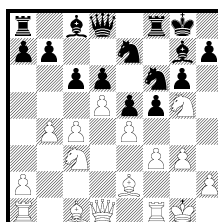
dc6 Be3 Ne6 c5 a4 Kg2 Qb3
Nh3 ef5 Rb1 Ba3 Kh1 f4 Qc2
Bb2 b5 Qd2 a3 Bd3 Qd3 Rf2
Bd2 Re1 h4 g4 Qe1 Nf7 h3 Kf2
Qa4 Bf4 Nb5 Nh7 Na4 Nb1
Qd4

#81 WTM: a5
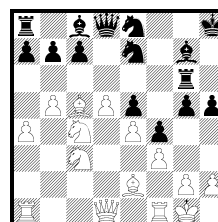
b6 d6 a5 Ba3 Qd2 Bd4 Qb3
Ba7 Kh1 Nb2 Be3 Rb1 Ra3
Qd3 Rc1 Nb1 g3 Qc2 h4 h3
Rf2 Bd3 Ne5 Be7 Na5 Bb4 Na2
Ra2 g4 Kf2 Qc1 Qb1 Bf2 Nd6
Qe1 Na3 Qd4 Re1 Nd2 Ne3
Bd6 Bb6 Nb6

#82 WTM: b6
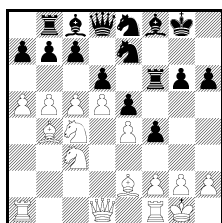
cd6 f3 Qd2 b6 Na4 Qa4 Nb2
Qb3 Qd4 Rb1 a6 c6 Bg4 Nb1
Rc1 Ra3 Re1 Nb6 g3 Ba3 Kh1
Qd3 Qb1 Ra2 h4 h3 Qe1 Qc1
Bh5 Ra4 Qc2 g4 Nd6 Na3 Nd2
Bf3 Ne5 Ne3 Bd3 Na2

#83 WTM: a4
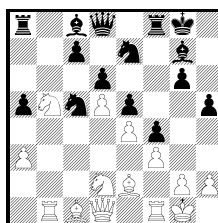
Nc4 Qc2 Nb3 a4 Bc4 Kh1 Nc7
Nc3 Ra1 Na7 h4 h3 Qa4 Qe1
g4 g3 Rf2 Rb3 Rb2 Bd3 Re1
Nd6 Nd4 Kf2 Bb2 Qb3 Rb4

#84 WTM: Bg5
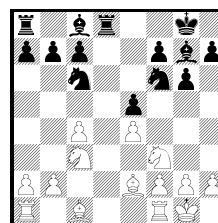
Rd1 Bg5 Be3 h3 Nd5 Re1 b4 b3
a3 c5 Bf4 Nb5 Rb1 Ng5 Bd2 a4
Ne1 Nd2 Bd3 Kh1 Nd4 Bd1 g3
Bh6 Na4 Nh4 Ne5 Nb1 h4 Nd1
g4

#85 WTM: Ne3
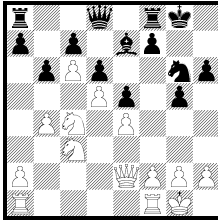
Ne3 Nb5 b5 Rad1 a4 Rac1
Rfd1 g3 f3 f4 Qg4 Qe3 Qh5
Nb2 Qb2 Kh1 Rab1 Rfc1 Nd2
a3 h4 Qd2 Qe1 Na5 Ne5 Qf3
Na3 g4 Qd3 Rfb1 Nb6 Qd1 Qc2
h3 Rae1 Rfe1 Nd6 Nb1 Na4
Nd1

#86 WTM: f4
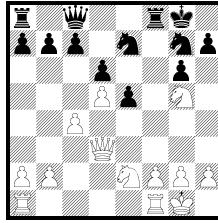
Ne4 Ng3 Qh3 Nc3 Ne6 f3 Qc3
f4 b4 Rae1 Rac1 Qe4 Qd2 c5
Qb3 b3 Nh7 g4 Rfe1 Rab1
Rad1 Qd4 Qe3 Qg6 g3 Nd4 h3
Qg3 h4 a4 a3 Nc1 Qa3 Qf3 Kh1
Nf4 Rfc1 Nf3 Qc2 Rfb1 Qd1
Nf7 Rfd1 Qb1 Nh3 Qf5

#87 WTM: Bc5
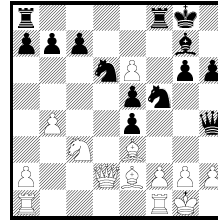
Nd5 Qd5 g3 Bc5 e7 f3 Ne4 a4
Rac1 Bg4 b5 Rab1 Rae1 Bd3
Ba7 Bc4 Rad1 Qe1 Qd1 g4 Nb5
Bb5 Bf4 Rfc1 Rfe1 Bf3 h3 Qc2
Rfb1 Qd6 Rfd1 Bd1 Ba6 a3
Bg5 Bb6 Bh6 Qc1 f4 Kh1 Na4
Qb2 Qd3 Bd4 Nb1 Bh5 Nd1
Qd4

#88 BTM: ab6
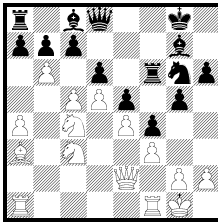
ab6 cb6 dc5 Bf8 Be6 Bd7 Rf7
a6 Bh3 Nh4 Rb8 h5 Re6 a5 c6
g4 Kh8 Qf8 Qe8 Rf5 Bf5 Kh7
Qd7 Nh8 Rf8 Bh8 Qe7 Kf8 Kf7
Bg4 Ne7 Nf8

#89 BTM: c5
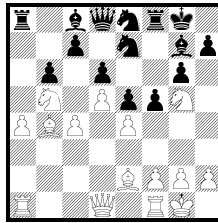
h6 Nf6 f4 fe4 c6 Rf7 Qd7 Rf6
Kh8 Bh6 Bf6 c5 Ra4 Bd7 Rb8
Nc6 Ra7 Ba6 Ra6 Ra5 h5 Bh8
Bb7 Be6 Nd5

#90 BTM: Rdg8
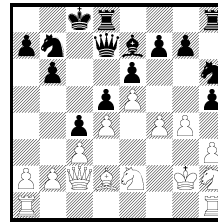
hg4 Rdg8 Ng8 Kb8 f6 Bh4 g6
f5 b5 Bc5 g5 Bd6 Qb5 Bg5
Rdf8 Qa4 Nc5 h4 Qc6 Nf5
Rde8 Kc7 Qe8 Rhg8 Nd6 Bf6
Ng4 Qc7 a5 a6 Bb4 Rhf8 Rhe8
Rh7 Ba3 Bf8 Qd6 Na5

#91 BTM: Bg4
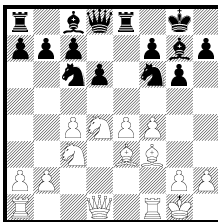
Nd4 a6 Nd7 Bg4 Ne5 Bd7 Qe7
Rb8 Na5 Ng4 Ne7 b6 Be6 a5
Ne4 h6 Qd7 Nh5 d5 Re4 h5
Bh8 Nb8 Bf8 Bh6 Bh3 Kh8
Nb4 Re7 Rf8 Kf8 Bf5 Re5 g5
Re6 b5 Nd5

#92 BTM: Nh4
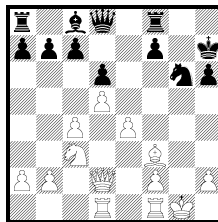
Qh4 Ne5 Bh3 Nh4 f5 Kg7 Qg5
Rg8 Qe7 Nf4 Bg4 Qf6 h5 Be6
c6 Rb8 Re8 a6 Bd7 b6 Qe8 Rh8
a5 Bf5 f6 Qd7 c5 Nh8 Ne7 b5
Kg8 Kh8

#93 BTM: d5
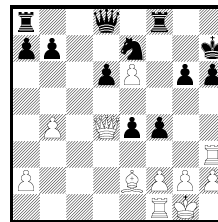
Nf5 d5 g5 Ng8 Qd7 Re8 Nc6
Rf6 Rf7 f3 Qe8 a6 Nd5 Rg8
Rc8 Qb6 e3 b6 Qa5 Rh8 Rf5 h5
Qb8 a5 Rb8 Qc8 Nc8 b5 Kg8
Qc7

#94 BTM: h6
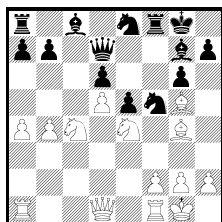
Nf6 h6 Nd4 b6 Qf7 Bf6 a6 Qc7
Nh6 Qe7 Kh8 Rf7 Ne7 Qa4 a5
Rb8 h5 Qe6 Ne3 Bh6 Qc6 Qb5
b5 Kf7 Nc7 Nh4 Bh8 Ng3 Rf6
Qd8

#95 BTM: e4
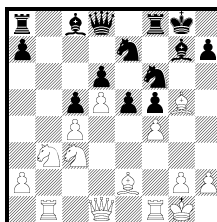
e4 ef4 h6 Ng6 Rb8 Qc7 Ne8
Bb7 Kh8 a6 Ne4 a5 Qe8 Qb6
Be6 Ng4 Qd7 Bd7 Re8 Bh6 Rf7
Kf7 Bh8 Nd7 Ned5 h5 Nh5 Qa5
Ba6 Nfd5 Nc6

#96 BTM: f4
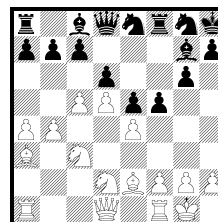
Ngf6 Nh6 f4 a6 Rf7 Nef6 a5 b6
Bh6 Ne7 g5 c6 Bd7 dc5 Qh4
Qg5 Be6 fe4 Rb8 h5 Rf6 h6 Qf6
Qd7 Bf6 Qe7 b5

#97 BTM: dc5
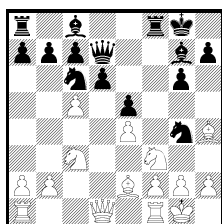
dc5 Nf6 g5 Nh6 Ne7 Rf4 Qf7
Bf6 Kh8 Nd8 Bh6 Nd4 a6 Re8
Rf3 Qf5 h6 Rf7 d5 Qe7 a5 Nb4
Nh2 Nb8 b6 h5 Qe8 Bh8 Ne3
Nf2 Na5 Qe6 Rb8 b5 Rf6 Kf7
Rd8 Rf5 Qd8

#98 BTM: fe5
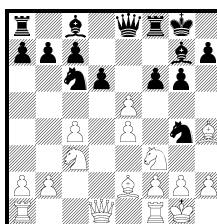
de5 Nge5 fe5 Nh6 Nce5 g5 Qe5
Bh6 f5 Be6 Kh8 Bd7 a6 Nd4
Bf5 Ne7 Qf7 Nd8 h6 b6 Rb8
Qe7 Qd7 a5 Ne3 Nb4 Rf7 h5
Nh2 Na5 Qd8 Bh8 Nb8 Qe6
Kf7 d5 Nf2 b5

#99 BTM: Ra8
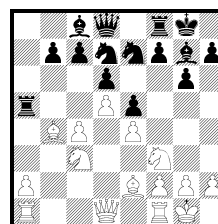
Ra8 f5 Nc5 Nf6 b6 Re8 c5 h6
c6 Rc5 g5 f6 Ra3 Nf5 Bh6 Nc6
b5 Ra6 Nb6 Qe8 Bh8 h5 Kh8
Rd5 Ra7 Bf6 Nb8 Ra4 Nd5
Rb5 Ra2

#100 BTM: f4
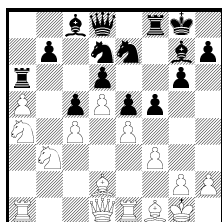
f4 Nf6 fe4 b6 g5 Qc7 b5 Rf6
Kh8 Qe8 Bh6 Ra7 Ra5 h6 Rc6
Bf6 Kf7 Nc6 Rf7 Ra8 h5 Re8
Bh8 Nb8 Nb6 Rb6 Qa5 Nd5
Qb6

Appendix D

Forward-Pruning Vectors

Appendix D presents the best FPVs with their performances in LOA (D.1) and in chess (D.2).

D.1   FPVs in LOA

The performance of the best FPVs in LOA is given in Table D.1 (see subsection 4.4 for details). The table has five parts corresponding to five search depths for the opponent. In each of the five parts of the table the first line represents the result for the full-width searcher, and the second line the +1 ply full-width searcher (both in italics). The three best FPVs obtained by the optimisation algorithm with each opponent depth and each explored FPV length are given. The performance of the overall-best FPV corresponding to a certain opponent depth is given in bold.

D.2   FPVs in chess

The best FPVs are given in Table D.2 for FPV-d, in Table D.3 for FPV-it and in Table D.4 for FPV-l (see subsection 4.4 for details). In each table and for each reference search depth, the first entry specifies the performance of the 1-ply deeper full-width searcher (in italics), and the next five entries the performance of the top five FPVs.

| opponent search depth = 1 |||
depth	FPV values	performance
1	*all*	*0.5675*
2	*all all*	*0.9275*
2	7 20	0.8200
2	10 7	0.8175
2	3 all	0.7275
3	10 2 1	**0.9050**
3	4 4 4	0.8925
3	7 6 1	0.8900
4	4 1 2 6	0.8500
4	5 1 2 5	0.8450
4	4 2 2 2	0.8400
5	3 1 2 1 3	0.8625
5	3 1 1 6 4	0.8575
5	3 1 2 3 2	0.8525
6	4 1 2 1 1 2	0.8450
6	4 1 1 1 1 10	0.8225
6	3 1 2 1 1 7	0.8025

| opponent search depth = 2 |||
depth	FPV values	performance
2	*all all*	*0.4775*
3	*all all all*	*0.8400*
3	10 all 6	**0.8050**
3	10 20 7	0.7850
3	20 10 5	0.7500
4	10 10 3 2	0.6225
4	10 7 4 2	0.6075
4	7 7 6 2	0.5950
5	7 4 3 2 3	0.6000
5	5 7 3 4 1	0.5925
5	7 6 2 3 2	0.5800
6	6 3 4 2 1 2	0.4900
6	5 3 5 2 1 2	0.4725
6	5 5 4 2 1 1	0.4700

| opponent search depth = 3 |||
depth	FPV values	performance
3	*all all all*	*0.5300*
4	*all all all all*	*0.8175*
4	all all 10 10	0.6650
4	all 20 20 5	0.6625
4	20 all 20 6	0.6200
5	20 all 5 7 4	**0.7750**
5	20 all 7 7 3	0.7500
5	20 10 7 20 2	0.7350
6	10 10 6 7 4 7	0.5575
6	7 all 7 5 2 2	0.5525
6	10 20 7 6 3 1	0.5500
7	6 10 6 7 4 1 1	0.5575
7	10 10 6 5 3 1 1	0.5450
7	7 4 6 4 6 3 1	0.5225

| opponent search depth = 4 |||
depth	FPV values	performance
4	*all all all all*	*0.4700*
5	*all all all all all*	*0.8000*
5	all all 20 all 4	**0.6675**
5	all all all 10 6	0.6650
5	all all 10 all 10	0.6550
6	20 all 10 6 7 4	0.5050
6	all all 7 4 7 3	0.4900
6	20 all 7 7 6 5	0.4825
7	20 all 7 7 4 2 4	0.5200
7	20 20 7 6 3 6 3	0.4700
7	10 20 6 10 6 1 4	0.4600

| opponent search depth = 5 |||
depth	FPV values	performance
5	*all all all all all*	*0.5025*
6	*all all all all all all*	*0.7375*
6	all all 10 all 10 all	0.5725
6	all all all 20 20 6	0.537500
6	all all all all 10 6	0.502500
7	20 all 20 all 10 7 4	**0.6750**
7	20 all 10 all 10 7 6	0.6675
7	all all 10 20 10 7 6	0.6500

Table D.1: FPV values and performances in LOA.

reference search depth = 4	
FPV values	performance
all all all all all	3.8
3 2 all 20 20	0.32
5 5 all 10 10	0.17
5 5 20 10 20	0.11
4 5 20 10 all	-0.09
3 2 20 20 all	-0.15

reference search depth = 5	
FPV values	performance
all all all all all all	2.8
4 10 40 10 10 40	0.19
5 30 all 3 5 10	-0.17
5 5 all 10 20 all	-0.33
5 5 100 30 5 10	-0.70
3 30 20 3 30 4	-0.74

reference search depth = 6	
FPV values	performance
all all all all all all all	2.2
10 30 20 4 all 10 20	-0.09
5 5 20 30 all 20 20	-0.49
5 10 30 5 all 20 all	-0.77
3 40 20 5 30 30 30	-0.81
20 30 all 2 5 10 all	-0.90

reference search depth = 7	
FPV values	performance
all all all all all all all all	1.2
all 5 20 all all 4 10 20	-0.96
all 20 20 40 10 10 20 5	-1.06
4 5 all 10 30 all all all	-1.12
all 4 all 30 all 10 5 20	-1.24
3 10 all 20 20 20 all all	-1.24

reference search depth = 8	
FPV values	performance
all all all all all all all all all	1.3
10 10 all 10 10 all 30 all 20	-1.00
all all all 10 20 5 10 10 all	-1.07
30 10 30 10 10 all 20 all 20	-1.09
20 30 50 3 30 10 3 all all	-1.66
3 all 30 5 10 all all all all	-1.66

Table D.2: FPV-d values and performances in chess.

reference search depth = 4	
FPV values	performance
all all all all all	3.8
all all 30 20 5	0.02
10 3 30 3 10	-0.11
all 5 20 4 10	-0.56
all all 20 3 10	-0.73
all all 5 10 10	-0.96

reference search depth = 5	
FPV values	performance
all all all all all all	2.8
all all 20 3 20 10	0.49
all 5 all 3 20 10	0.44
all 20 30 5 20 10	0.32
all 20 50 3 20 10	0.23
all all 20 10 20 10	0.20

reference search depth = 6	
FPV values	performance
all all all all all all all	2.2
all all all 20 5 30 10	-0.18
all 40 10 40 4 30 10	-0.42
all 5 all 40 10 30 10	-0.49
all all all all all 20 10	-0.56
all 20 40 20 10 30 10	-0.55

reference search depth = 7	
FPV values	performance
all all all all all all all all	1.2
all 20 all 3 all 10 10 20	0.18
10 30 5 40 40 5 50 10	-0.68
all all all all all 30 30 10	-1.11
10 2 2 4 4 30 all 5	-2.96
all all 20 2 10 30 all 5	-3.08

reference search depth = 8	
FPV values	performance
all all all all all all all all all	1.3
all all 40 all all all 10 10 20	-0.26
all all all all all 50 10 10 20	-0.32
all all all all all all 3 10 20	-0.42
all 30 2 20 20 10 all 30 10	-1.08
all all 5 all 10 10 all 30 10	-1.23

Table D.3: FPV-it values and performances in chess.

reference search depth = 4	
FPV values	performance
all all all all all	3.8
all 2 5 20 10	1.04
5 4 20 10 10	0.82
5 2 5 all 10	0.79
20 3 3 30 10	0.08
10 2 4 all 10	-0.09

reference search depth = 5	
FPV values	performance
all all all all all all	2.8
30 2 5 30 20 10	0.74
10 10 30 5 5 20	0.46
5 2 10 all 20 10	0.45
3 5 20 40 10 5	0.44
all 3 10 30 10 5	0.17

reference search depth = 6	
FPV values	performance
all all all all all all all	2.2
5 10 all 10 10 all 20	1.08
10 10 all 5 10 all 20	1.02
40 5 20 5 10 all 20	0.76
5 5 all 20 10 10 all	0.69
10 10 all 10 5 all 20	0.62

reference search depth = 7	
FPV values	performance
all all all all all all all all	1.2
all 3 all 30 10 10 all 20	0.25
10 10 10 all 20 10 all 20	0.09
10 4 10 10 all all all 20	0.09
4 10 20 30 20 10 all 20	0.06
20 5 all 30 10 10 all 20	0.00

reference search depth = 8	
FPV values	performance
all all all all all all all all all	1.3
5 2 10 20 30 20 all all 20	-0.16
all 3 20 40 10 10 all all 10	-0.19
5 2 10 all 20 30 all 20 20	-0.23
all 20 30 3 5 30 all all 10	-0.30
4 2 10 all 20 30 all 30 20	-0.31

Table D.4: FPV-l values and performances in chess.

Summary

This thesis presents research on learning algorithms suitable for improving program decisions that may influence the game-tree search. Such decisions are called search decisions. In Chapter 1, we provide some general background information. We introduce a classification of search decisions based on the difference in the way search decisions are evaluated (more precisely how their efficacy is evaluated). Three classes are identified: (1) Class-P search decisions, (2) Class-S search decisions, and (3) Class-G search decisions. Class-P search decisions are evaluated by using Positions only, Class-S search decisions are evaluated by using Search trees, Class-G search decisions are evaluated by using complete move sequences (as they occur in Game trees).

We would like to investigate how learning can be employed for search decisions. This leads to the formulation of our research question: which learning algorithms can improve search decisions? The question is addressed in the following chapters by choosing an instance for each of the three classes of search decisions, and by designing new learning algorithms suitable for these instances. The learning algorithms are tested experimentally.

Chapter 2 describes the environments employed for testing the learning algorithms. A test environment consists of a game and a game program. Two games are employed: Lines of Action (LOA) and chess. Both games are two-person zero-sum games with perfect information. LOA is a game with a smaller complexity, a shorter game length, and three relatively uniform game phases (i.e., the number of pieces and the strategies involved are not changing drastically during the game). These features make LOA an attractive domain for generating new ideas, since they can be tested fast. In LOA, we use two test environments: (1) the NN-population environment, which includes a set of 100 neural-network evaluation functions, combined with a simple search engine, and (2) the MIA environment adopted from the tournament program MIA. Chess has a larger complexity, and so far received most attention from the game-research community. As a consequence, the available chess programs are fine-tuned, and use state-of-the-art techniques. Thus, chess is a suitable domain for evaluating the algorithms developed. In chess, we use one test environment adopted from the tournament program CRAFTY.

Chapter 3 deals with move ordering. This is considered to be the main instance of class-P search decisions. We introduce a new move-ordering heuristic, the Neural MoveMap heuristic. The heuristic uses a neural network to estimate

the likelihood of a move being the best in a certain position. The moves considered more likely to be the best are examined first. We describe the details of the heuristic, and we formulate five questions. The first three questions concern details of the heuristic; the next two questions deal with the performance of the heuristic. All questions are answered by experiments. From the experiments on the details of the heuristic we conclude that (1) the best choice for move encoding is using one output unit for each possible move, (2) the neural network should be trained with large learning sets, which are labelled either by a game program or by using moves from a game database, and (3) the best approach to use the neural network during the search is a weighted-combination approach, which orders the moves according to a weighted sum of the neural-network scores and the history-heuristic scores. From the experiments evaluating the performance of the heuristic, we conclude that (1) by using the Neural MoveMap heuristic the search expands smaller trees than the existing heuristics (specifically the history heuristic) both in the games of LOA and chess, and (2) the neural network predicts the best move correctly in chess almost as often as a 7-ply search of CRAFTY.

Chapter 4 deals with forward pruning. This is an instance of class-S search decisions. Forward pruning is evaluated by the size of the search tree and by the quality of the move. The measures are usually averaged over a set of positions. We define two types of representation for forward pruning: (1) forward-pruning vector (FPV), which is a discrete representation, and (2) forward-pruning function (FPF), which is a continuous representation. For each of the two types of representation a learning algorithm is developed: the TS-FPV algorithm for FPVs, and the RL-FPF algorithm for FPFs (TS stands for Tabu Search and RL for Reinforcement Learning). Both algorithms are designed for maximising the quality of the move selected by the search while keeping the size of the search tree below a specified limit. The algorithms rely on two assumptions: (1) if at some depth(s) the game-tree search is wider while at no depth it is narrower, the number of nodes explored does not decrease; and (2) if at some depth(s) the search is wider while at no depth it is narrower, the performance does not decrease. The critical forward-pruning solutions (FPVs or FPFs) are those that are feasible (i.e., the size of the search tree is below the limit), and cannot be widened at any depth without becoming infeasible. If the two assumptions hold, the optimal solution is among the critical ones. The TS-FPV algorithm uses a tabu-search algorithm to explore the space of the FPVs focusing on the critical FPVs. The RL-FPF algorithm is a reinforcement-learning algorithm for FPFs. It uses a gradient-descent update rule, which combines four reward terms corresponding to (1) the size of the search tree, (2) the quality of the move in the root, (3) the moves in an internal node that change the value of the node, and (4) the moves in an internal node that do not change the value of the node. The experimental results show that the two algorithms are able to tune a forward-pruning scheme that has a better overall performance than the full-width search. The FPFs obtained from RL-FPF outperformed the best FPVs resulting from TS-FPV.

Chapter 5 deals with time allocation. This is an instance of class-G search

decisions. We propose two learning algorithms for time allocation, viz. the Meta-Actor-Critic algorithm and the SANE-TA algorithm. The Meta-Actor-Critic algorithm (MAC) is a gradient-descent actor-critic algorithm. It uses the framework of a temporal-difference algorithm for learning the state-action values of the search decisions, with the change in the value of the evaluation function (the critic) as intermediary reward (the evaluation function being an estimation of the game result). The SANE-TA algorithm is an application to time allocation of an existing genetic algorithm, SANE. From the experiments we conclude that the SANE-TA algorithm results in a better time-allocation scheme than the MAC algorithm, but the time required for training by the SANE-TA algorithm is significantly higher, and even prohibitive. Both algorithms result in a comparable or better time allocation than the existing time allocation algorithms tested in the experiments. Finally, we may conclude that the MAC algorithm is able to use the additional freedom in dynamic time allocation, and that it is in no way hampered by multiple search decisions during one search process.

In Chapter 6, we revisit the research question and summarise the main conclusions of the preceding chapters. Moreover, we provide promising directions for future research. For each learning algorithm described in the thesis, we discuss how these algorithms can be improved and/or extended for other instances of search decisions.

Samenvatting

Dit proefschrift beschrijft een onderzoek op het gebied van leeralgoritmen. Het gaat om het verbeteren van programma-beslissingen, die het zoekproces in de spelboom beïnvloeden. Zulke beslissingen noemen we in deze samenvatting kortheidshalve *zoekbeslissingen*. Hoofdstuk 1 geeft enige algemene achtergrond-informatie. We introduceren een classificatie van zoekbeslissingen, die gebaseerd is op het verschil in de manier waarop zoekbeslissingen worden geëvalueerd (beter gezegd hoe hun effect wordt geëvalueerd). We identificeren drie klassen: (1) Klasse-P zoekbeslissingen, (2) Klasse-S zoekbeslissingen, en (3) Klasse-G zoekbeslissingen. Klasse-P zoekbeslissingen worden geëvalueerd met alleen spel-posities (Eng.: Positions). Klasse-S zoekbeslissingen worden geëvalueerd met zoekbomen (Eng.: Search trees), Klasse-G zoekbeslissingen worden geëvalueerd met volledige partijen (Eng.: Games).

We onderzoeken hoe diverse verschillende leermethoden gebruikt kunnen worden voor het nemen van zoekbeslissingen. Dit leidt tot de formulering van onze onderzoeksvraag: welke leeralgoritmen kunnen zoekbeslissingen ver-beteren? De vraag wordt in de volgende hoofdstukken nader bezien. We kiezen voor elk van de drie klassen zoekbeslissingen een voorbeeld en ontwerpen ver-volgens nieuwe leeralgoritmen die geschikt zijn voor deze voorbeelden. De leer-algoritmen worden experimenteel getest.

Hoofdstuk 2 beschrijft de omgevingen die gebruikt zijn om de leeralgorit-men te testen. Een test omgeving bestaat uit een spel en een spelprogramma. Er is gebruik gemaakt van twee spelen: Lines of Action (LOA) en schaken. Beide spelen zijn twee-persoons nul-som spelen met perfecte informatie. LOA is een spel met een kleinere complexiteit, een kortere spellengte en drie relatief uniforme spelfasen (d.w.z. dat het aantal stukken en de gebruikte strategieën niet drastisch veranderen tijdens het spel). Deze kenmerken maken LOA tot een aantrekkelijk domein voor het genereren van nieuwe ideeën, vooral omdat ze snel kunnen worden getest. In LOA gebruiken we twee test-omgevingen: (1) de NN-populatie omgeving, met een verzameling van 100 neurale-netwerk-evaluatiefuncties die verbonden worden met een eenvoudige zoekmachine, en (2) de MIA omgeving, die gebruik maakt van het toernooiprogramma MIA. Schaken heeft een grotere complexiteit en heeft tot nu toe de meeste aandacht van de spel-onderzoekgemeenschap gehad. Dientengevolge zijn de beschikbare schaakprogramma's zeer verfijnd en maken ze gebruik van state-of-the-art tech-nieken. Daarom is schaken een geschikt domein om de ontworpen algoritmen te

testen. Bij het schaken gebruiken we één test-omgeving, die gebaseerd is op het toernooi programma CRAFTY.

Hoofdstuk 3 gaat over het ordenen van zetten. Dit is het belangrijkste voorbeeld van een klasse-P zoekbeslissing. We introduceren een nieuwe heuristiek voor het ordenen van zetten, de *Neural MoveMap* heuristiek. De heuristiek gebruikt een neuraal netwerk om een schatting te geven van de waarschijnlijkheid dat een zet de beste is in een bepaalde positie. De zetten die waarschijnlijker geacht worden de beste te zijn worden eerst onderzocht. We beschrijven de details van de heuristiek en we formuleren vijf vragen. De eerste drie vragen hebben betrekking op de details van de heuristiek en de overige twee vragen op de prestaties van de heuristiek. De vragen worden beantwoord met behulp van experimenten. Op basis van de experimenten betreffende de details van de heuristiek concluderen wij (1) dat de beste keuze om de zetten te coderen is het gebruik van één output-unit voor iedere mogelijke zet, (2) dat het neurale netwerk getraind moet worden met grote leerverzamelingen die van een waarde voorzien zijn ofwel met behulp van een spelprogramma ofwel met behulp van de zetten uit een spel-database, en (3) dat de beste benadering om het neurale netwerk tijdens het zoekproces te gebruiken een benadering met gewogen combinaties is: de benadering ordent de zetten volgens een gewogen som van de scores van het neurale netwerk en de scores van de *history* heuristiek van de zetten. Op basis van de experimenten die de prestaties van de heuristiek evalueren concluderen we (1) dat met behulp van de Neural MoveMap heuristiek het zoekproces kleinere bomen ontwikkelt dan de bestaande heuristieken (met name de history heuristiek), zowel bij LOA als bij schaken, en (2) dat het neurale netwerk bij schaken de beste zet bijna net zo vaak goed voorspelt als een 7-ply zoekproces van het programma CRAFTY.

Hoofdstuk 4 handelt over voorwaarts snoeien. Dit is een voorbeeld van een klasse-S zoekbeslissing. Voorwaarts snoeien wordt geëvalueerd door middel van de grootte van de zoekboom en de kwaliteit van de zet. De maten worden gewoonlijk gemiddeld over de verzameling posities. Wij definiëren twee typen representaties voor voorwaarts snoeien: (1) FPVs (*Forward-Pruning Vectors*), een discrete representatie, en (2) FPFs (*Forward-Pruning Functions*), een continue representatie. Voor ieder van de twee representaties is een leeralgoritme ontwikkeld: het TS-FPV algoritme voor FPVs, en het RL-FPF algoritme voor FPFs (TS staat voor *Tabu Search* en RL voor *Reinforcement Learning*). Beide algoritmen zijn ontworpen om de kwaliteit van de zet die geselecteerd is op basis van het zoekproces te maximaliseren met inachtneming van een bovengrens voor de grootte van de zoekboom. De algoritmen zijn gebaseerd op twee aannamen: (1) dat als de zoekboom op een of meer diepten breder is en op geen enkele diepte smaller, dan is het aantal onderzochte knopen niet minder, en (2) dat als de zoekboom op een of meer diepten breder is en op geen enkele diepte smaller, dan is de prestatie niet minder. De kritische voorwaarts-snoei oplossingen (FPVs of FPFs) zijn die oplossingen die mogelijk zijn (dat wil zeggen, waarvoor de grootte van de zoekboom kleiner dan de bovengrens is) en niet op willekeurige diepte breder kunnen worden gemaakt zonder dan onmogelijk te worden. Als de twee aannamen gelden, dan moet de optimale oplossing zich

bevinden tussen de kritische oplossingen. Het TS-FPV algoritme gebruikt een *tabu-search* algoritme om de ruimte te exploreren van de FPVs, met een focus op kritische FPVs. Het RL-FPF algoritme is een *reinforcement-learning* algoritme voor FPFs. Het maakt gebruik van een *gradient-descent update* regel die vier beloningstermen combineert overeenkomstig (1) de grootte van de zoekboom, (2) de kwaliteit van de zet in de wortel van de boom, (3) de zetten in een interne knoop die de waarde van de knoop veranderen, en (4) de zetten in een interne knoop die de waarde van de knoop niet veranderen. De experimentele resultaten tonen aan dat beide algoritmen een voorwaarts-snoeiproces kunnen ontwikkelen dat een beter resultaat oplevert (kleinere boom) dan een volledig zoekproces. De FPFs die verkregen zijn met behulp van het RL-FPF algoritme waren beter dan de beste FPVs die verkregen zijn met het TS-FPV algoritme.

Hoofdstuk 5 gaat over tijdstoewijzing. Dit is een voorbeeld van een klasse-G zoekbeslissing. We stellen twee leeralgoritmen voor tijdstoewijzing voor, namelijk het Meta-Actor-Critic algoritme en het SANE-TA algoritme. Het Meta-Actor-Critic algoritme (MAC) is een *gradient-descent actor-critic* algoritme. Het maakt gebruik van een *temporal-difference* algoritme om de waarden te leren van toestand-actie paren voor zoekbeslissingen, waarbij de verandering in de waarde van de evaluatiefunctie (de critic) als tussentijdse beloning optreedt. (De evaluatiefunctie wordt dan gebruikt als een schatter van het resultaat van het spel.) Het SANE-TA algoritme is de toepassing van een bestaand genetisch algoritme, SANE, op het gebied van tijdstoewijzing. Op basis van de experimenten concluderen wij dat het SANE-TA algoritme een beter tijdstoewijzingsschema oplevert dan het MAC algoritme, maar dat de tijd die nodig is om het eerstgenoemde algoritme te trainen is beduidend langer, en dikwijls zelfs te lang. Beide algoritmen resulteren in vergelijkbare of betere tijdstoewijzing dan de bestaande tijdstoewijzingsalgoritmen die in de experimenten getest zijn. Ten slotte concluderen wij dat het MAC algoritme in staat is om de additionele vrijheid in de dynamische tijdstoewijzing te benutten, en niet belemmerd wordt door meervoudige zoekbeslissingen tijdens een zoekproces.

In Hoofdstuk 6 komen we terug op de onderzoeksvraag en vatten de belangrijkste conclusies van de voorgaande hoofdstukken samen. Verder geven we enige veelbelovende richtingen voor vervolg-onderzoek aan. Voor ieder leeralgoritme dat is beschreven in dit proefschrift gaan we na hoe deze algoritmen verbeterd kunnen worden voor en/of uitgebreid naar andere voorbeelden van zoekbeslissingen.

Curriculum Vitae

Levente Kocsis was born in Oradea, Romania, on September 10, 1972. In 1991 he started his undergraduate study of Computer Science at the "Politehnica" University of Timişoara. In 1996 he successfully defended his thesis in the field of Artificial Intelligence, receiving his bachelor degree in engineering. In 1997 he received his M.Sc. degree in Computer Science at the same university. After graduation, he worked as teaching assistant at the Computer Science Department of the "Politehnica" University of Timişoara.

Between 1999 and 2003 he worked as research assistant (AIO) at the Department of Computer Science (Institute for Knowledge and Agent Technology - IKAT), Universiteit Maastricht, The Netherlands. The research presented in this thesis was performed during this period.

His research in the field of games was facilitated by his chess expertise. In chess, he holds the title of FIDE master, with a rating around 2400 ELO points.

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 (TUE) *Prototyping of CMS Storage Management*

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

---

[1] Abbreviations: SIKS – Ph.D. School for Information and Knowledge Systems; CWI – Centrum voor Wiskunde en Informatica, Amsterdam; EUR – Erasmus Universiteit, Rotterdam; KUB – Katholieke Universiteit Brabant, Tilburg; TUE – Technische Universiteit Eindhoven; TUD – Technische Universiteit Delft; RUL – Universiteit Leiden; UM – Universiteit Maastricht; UT – Universiteit Twente, Enschede; UU – Universiteit Utrecht; UvA – Universiteit van Amsterdam; 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 (TUE) *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 Warehouses*

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

18 Levente Kocsis (UM) *Learning Search Decisions*